Udacity CS101: Building a Search Engine
Unit 5: How Programs Run- Making Things Fast

Introduction

In the last unit you built a search index that could respond to queries by going through each entry one at a time. The search index checked to see if the keyword matched the word you were looking for and then responding with a result.

However, with a large index and lots of queries, this method will be too slow. A typical search engine should respond in under a second and often much faster.

In this unit you will learn how to make your search index much faster.

Making Things Fast

The main goal for this unit is to develop an understanding of the cost of running programs. So far, you haven't worried about this and have been happy to write code that gives the correct result. Once you start to make programs bigger, make them do more things or run on larger inputs, you have to start thinking about the cost of running them. This question of what it costs to evaluate an execution is a very important, and it is a fundamental problem in computer science. Some people spend their whole careers working on this. It's called **algorithm analysis**.

You may not be aware of this but you've already written many algorithms. An **algorithm** is a procedure that always finishes and produces the correct result. A **procedure** is a well defined sequence of steps that it can be executed mechanically. We're mostly interested in procedures which can be executed by a computer, but the important part about what makes it a procedure is that the steps are precisely defined and require no thought to execute.

To be an algorithm, it has to always finish. You've already seen that it isn't an easy problem to determine if an algorithm always finishes. It isn't possible to answer that question in general, but it can be answered for many specific programs.

So, once you have an algorithm, you have well a defined sequence of steps that will always finish and produce the right results. This means you can reason about its cost.

What is Cost?

The way computer scientists think about cost is quite different from how most people think about cost.

When you think about the cost of things, you know specific things such as the red car costs $25000 and the green car $10000 – the red car costs more than the green car. You just have to compare

those costs. This is thinking in terms of very specific things with specific costs. It's different with algorithms. We don't usually have a specific execution in mind. The cost depends on the inputs.

Suppose algorithms Algo 1 and Algo 2 both solve the same problem.

Inputs	Algo 1	Output
Inputs	Algo 2	Output

You can't put a fixed price on them like with the cars. For some inputs, it might be the case that Algo 1 is cheaper than Algo 2, but for others, Algo 2 might be cheaper. You don't want to have to work this out for every input because then you might as well run it for every input. You want to be able to predict the cost for every input without having to run every input.

The primary way that cost is talked about in computer science is in terms of the size of the input. Usually, the size of the input is the main factor that determines the speed of the algorithm, that is, the cost in computing is measured in terms of how the time increases as the size of the input increases. Sometimes, other properties of the input matter, which will be mentioned later. Ultimately, cost always comes down to money. What costs money when algorithms are executed?

- The time it takes to finish – if it finishes more quickly, you'll spend less time on it. You can rent computers by the time it takes to execute. There are various cloud computing services where you pay for a certain sized processor for the time you use it. It's just a few cents per hour. Time really is money and, although we don't need to turn the costs into money because we might not know the exact computing costs, understanding the time to execute will give a good sense of the cost.
- Memory – if a certain amount of memory is needed to execute an algorithm then you have an indication of the size and cost of computer required to run the program.

In summary, cost is talked about in terms of time and memory rather than money; although the actual implementation of these do convert to actual monetary costs. Time is often the most important aspect of cost, but memory is also another consideration.

Q5-1: Measuring Speed

Why do computer scientists focus on measuring how time scales with input size, instead of absolute time? (Check all correct answers.)

a. We want to predict how long it will take for a program to execute before running it.
b. We want to know how the time will change as computers get faster.
c. We want to understand fundamental properties of our algorithms, not things specific to a particular input or machine.
d. We want abstract answers, so they can never be wrong.

Throughout the entire history of computing, it's been the case that the computer you can buy in a year for the same amount of money will be faster than the computer you can buy today.

Answer to Q-1

Stopwatch

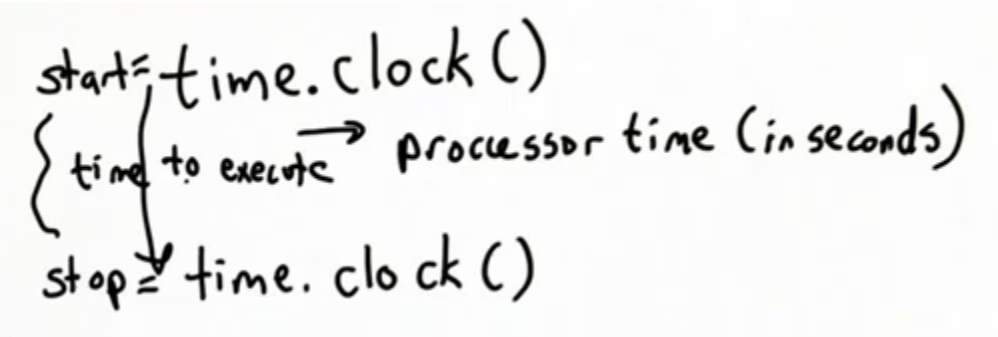
In this section, you'll see how you can check how long it takes for a piece of code to run. You may have seen people on the forums talking about bench marking code.

The procedure, **time_execution**, below, is a way to evaluate how long it takes for some **code** to execute. You could try to do this with a stopwatch but to be accurate, you'd have to run programs for a very long time. It's more accurate to use the built-in **time.clock** in the **time** library. An explanation of what is going on follows the code:

```
import time #this is a Python library

def time_execution(code):
   start = time.clock()  # start the clock
   result = eval(code)  # evaluate any string as if it is a Python
command
   run_time = time.clock() - start  # find difference in start and end
time
   return result, run_time  # return the result of the code and time
taken
```

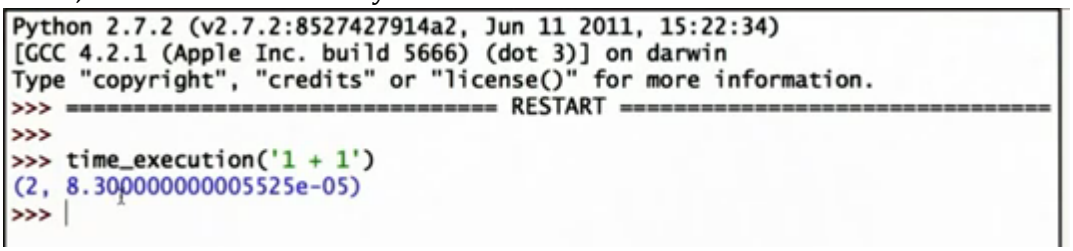
The time taken is calculated according to this diagram.



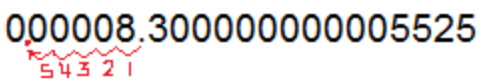
The clock is started. The time it reads when it starts is somewhat arbitrary, but it doesn't matter because you're only interested in the time difference between it starting and stoping and not an absolute start and end time. After the clock is started, the **code** you want to evaluate is executed. This is done by the rather exciting method **eval('<string>')**. It allows you to run any code input as a string! You put in a string and it runs it as Python code. For example, **eval('1 + 1')** runs the code **1 + 1**. After the code is executed, the clock is stopped and the difference between the start and stop times is calculated and stored as **run_time**. Finally, the procedure returns the result of the **code** in **eval** and its running time.

You can run the timing through the web interpreter, but it won't be accurate and there is a limit on how long your code is allowed to run there. If you have Python installed on your computer, you'll be able to run it on that. If you don't, then that's ok. Instructions will be available under the unit's Supplementary Information. You don't need to run it for yourself, but you should at least see it illustrated. The following outputs are all run in a Mac shell on Dave's desktop.

Recall that the input '**1 + 1**' to **time_execution**, shown in green in the image below, is sent as input to eval , which runs **1 + 1** as Python code.

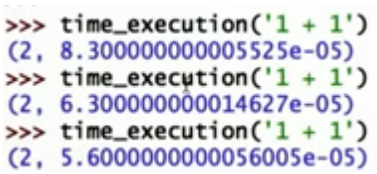


The run time is written in scientific notation: *8.300000000005525e-05* which is sometimes also written as 8.300000000005525 x 10^-5, or in Python code **8.300000000005525 * 10**-5**. In decimal terms, this can be written as 0.00008300000000005525, or rounded to 0.000083. You can see where this comes from by looking at the *-5* after the *e*. It tells you to move the decimal point 5 steps to the left like this:



The units are seconds, so this is only a fraction of a millisecond, that is, about 0.08 ms.

Trying the same instruction over and over, the result varies because other things are going on in the machine at the same time, but it's around the same value.



Instead, try larger numbers, the time is still very very small.

The actual processing time is even lower, because, for instance, starting and stopping the clock.

This doesn't tell you very much for short, fast executions, so next you'll see some longer ones.

Spin Loop

In order to get a better idea of how timing works, define the procedure **spin_loop**:

```
def spin_loop(n):
    i = 0
    while i < n:
        i = i + 1
```

This code will run for longer, and by picking the value **n** you can go through and loop any number

of times. The image below shows **spin_loop** running 1000, 10000, 100000 and 1000000 times, returning a result that is the time it takes to run the loop that number of times.

```
>>> time_execution('spin_loop(1000)')
(None, 0.00013299999999993872)
>>> time_execution('spin_loop(10000)')
(None, 0.0006189999999999252)
>>> time_execution('spin_loop(100000)')
(None, 0.005698000000000203)
>>> time_execution('spin_loop(1000000)')
(None, 0.05523699999999998)
```

It is important to notice that the time changes depending on the input – when you increase the input, the time (or the output) also increases accordingly.

Predicting Run Time

First you'll see the time taken for running some code, and then there will be a quiz. This will show if you understand execution time well enough to make some predictions.

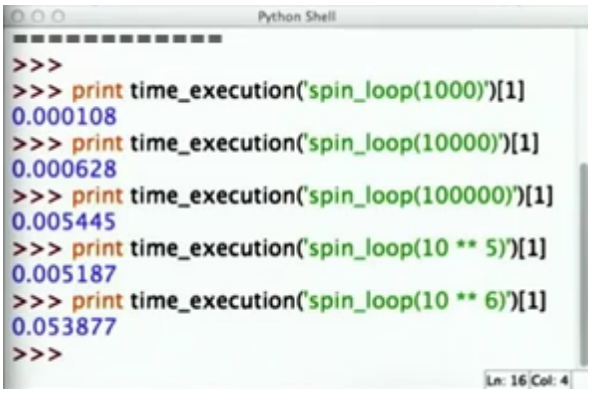
The code used to measure the time is the **time_execution** procedure from before which evaluates the time taken for the **code** passed in, and then a procedure **spin_loop** which just adds one to a variable as it loops through the numbers up to **n**.

```
import time

def time_execution(code):
   start = time.clock()
   result = eval(code)  # evaluate any string as if it is a python
command
   run_time = time.clock() - start
   return result, run_time

def spin_loop(n):
   i = 0
   while i < n:
       i = i + 1
```

The results for execution times, in seconds, for **spin_loop** are given below. Note the **[1]** is there to just print out the running time rather than the result of evaluating **spin_loop**. The first result is for running the loop 1000 times, then 10000 followed by 100000. The next is a repeat of the 100000 iterations (runs) through the loop, but written in Python's power notation for 10^5. The final time is the execution time for running through the loop 10^6, which is one million times. All the execution times are given in seconds.



Q5-2 Predicting Run Time

Given the execution times above, what is the expected execution time for **`spin_loop(10**9)`** (one billion) in seconds? You won't be able to guess the exact time, but the grader is looking for a multiple of 5.

Answer to Q-2

Make Big Index

The examples you've seen so far have shown the time taken to run short procedures. Next you'll see some test on longer code – the index code, which is important to the overall look up time of your search engine. In order to test this code, you'll first need to build a big index. You could do this by hand but it would take a very long time! The code to do it is as follows.

```
def make_big_index(size):
    index = []
    letters = ['a','a','a','a','a','a,','a','a']
    while len(index) < size:
        word = make_string(letters)
        add_to_index(index, word, 'fake')
        for i in range(len(letters) - 1, 0, -1):
            if letters[i] < 'z':
                letters[i] = chr(ord(letters[i])+ 1)
                break
        else:
                letters[i] = 'a'
    return index
```

First an empty index called **`index`** is created, and a list, **`letters`**, of eight letter a's. Next, there is a **`while`** loop that adds an entry to the index each time it iterates. (Iterates means going through the loop – one iteration is one pass through the loop.) The while loop continues to add to the index until it is of the required length, **`size`**. The details as to what happens in loop are as follows.

```
word = make_string(letters)
```

The procedure **make_string** takes as its input **letters** and returns a single string of the entries contained in the list. This means that, for example, **['a','a','a','a','a','a','a','a']** becomes **'aaaaaaaa'**. The code for this is:

```
def make_string(p):
   s=""
   for e in p: # for each element in the list p
       s = s + e  # add it to string s
   return s

add_to_index(index, word, 'fake')
```

This is the add_to_index code from before, and it adds an entry which looks like this:

['aaaaaaaa', ['fake']] to **index**.

```
for i in range(len(letters) - 1, 0, -1):
```

Although you've seen **range** before, here it is used differently.. If **len(letters)** is 8, then **range(len(letters) - 1, 0, -1)** becomes **range(8, 0, -1)**,which is a list which counts down from 4 to one greater than 0 in steps of -1, that is, *[8, 7, 6, 5, 4, 3, 2, 1]*. So, the for loop runs through the values of the list counting backwards from **len(letter)-1** down to 1. Discussion on Range

The **for** loop starts from the last letter in **letters**, and checks if it is a **'z'**. If it is, it changes it to **'a'**, and goes back to the top of the loop for the next iteration, which will check one position closer to the beginning of the loop. It continues until it finds a letter which is not a **'z'**.

Once it finds a letter which isn't a **'z'**, the line **letters[i] = chr(ord(letters[i])+ 1)** changes it to one letter later in the alphabet, **'a'**→**'b'**, **'b'**→**'c'** and so on . (Precisely what **chr** and **ord** do will be explained later.) After a letter has been changed, the **for**-loop stops and the code returns to the top of the **while**-loop .

During the first iteration of the **while**-loop, the **for**-loop it will change from:
['a','a','a,','a','a','a','a','a'] to **['a','a','a,','a','a','a','a','b']** and then break. The second time through, it will change from:
['a','a','a,','a','a','a','a','b'] to **['a','a','a,','a','a','a','a','c']** and break, and so on.

When it gets to:
`['a','a','a,','a','a','a','a','z'],` it will change it first to:
`['a','a','a,','a','a','a','a','a']`.

Then it will look at the last but one position, as it goes through the list backwards. It will change the **`'a'`** in that position to **`'b'`** which changes the list to: **`['a','a','a,','a','a','a','b','a']`**, after which it breaks.

Finally, when the **`while`** loop has finished, the index of length **`size`** is returned.

The complete code to try this is reproduced below, so you can try it for yourself if you'd like.

```
def add_to_index(index,keyword,url):
   for entry in index:
       if entry[0] == keyword:
           entry[1].append(url)
           return
   index.append([keyword,[url]])

def make_string(p):
   s=""
   for e in p:
       s = s + e
   return s

def make_big_index(size):
   index = []
   letters = ['a','a','a','a','a','a','a','a']
   while len(index) < size:
       word = make_string(letters)
       add_to_index(index, word, 'fake')
       for i in range(len(letters) - 1, 0, -1):
           if letters[i] < 'z':
               letters[i] = chr(ord(letters[i])+ 1)
               break
        else:
               letters[i] = 'a'
   return index
```

To see what it does, you can look at some small values.

```
print make_big_index(3) # index with 3 keywords
[['aaaaaaaa', ['fake']], ['aaaaaaab', ['fake']], ['aaaaaaac', ['fake']]]

print make_big_index(100) # index with 100 keywords
[['aaaaaaaa', ['fake']], ['aaaaaaab', ['fake']], ['aaaaaaac', ['fake']],
['aaaaaaad', ['fake']], ['aaaaaaae', ['fake']], ['aaaaaaaf', ['fake']],
... <snip> ...
['aaaaaadm', ['fake']], ['aaaaaadn', ['fake']], ['aaaaaado', ['fake']],
['aaaaaadp', ['fake']], ['aaaaaadq', ['fake']], ['aaaaaadr', ['fake']],
['aaaaaads', ['fake']], ['aaaaaadt', ['fake']], ['aaaaaadu', ['fake']],
['aaaaaadv', ['fake']]]
```

As you can see from the end of the list, the second to last letter has been changed as well as the last index.

To test the index, some larger indexes are needed. You'll see the results of tests run on one of length 10 000, and another of length 100 000. It takes over 5 minutes to construct the index of length 100 000. This doesn't matter as it is the **lookup** time that is important for your search engine.

To construct the smaller of the two indexes, you can use the line

```
index10000 = make_big_index(10000)
```

The time to run this **lookup** on **index10000** is checked several times, which you can see below. Note that the time varies, but is around the same value.

```
>>> time_execution('lookup(index10000, "udacity")')
(None, 0.0008559999999997459)
>>> time_execution('lookup(index10000, "udacity")')
(None, 0.000968000000000302)
>>> time_execution('lookup(index10000, "udacity")')
(None, 0.0009059999999863066)
```

After constructing the larger index of length 100 000 using:

```
index1000000 = make_big_index(100000)
```

you can see the element at the last position using **index1000000[99999]**, or the equivalent **index1000000[-1]**, just like with strings.

```
print index1000000[99999]
['aaaafryd',['fake']]
index100000[-1]
```

['aaaafryd',['fake']]

The timings for the longer index of length 100 000 are as follows:

```
>>> time_execution('lookup(index100000, "udacity")')
(None, 0.00859000000002652)
>>> time_execution('lookup(index100000, "udacity")')
(None, 0.00851799999998093)
```

If you compare these with the times for the index of length 10 000, you'll see that it takes approximately 10 times longer to do a look up in the longer index than in the shorter index.

Timings vary for many reasons. One reason is that lots of other things run on the computer, which means that the program does not have total control over the processor. Another reason is that where things are in memory can take a longer or shorter time to retrieve. What matters is that the run time is roughly the same each time the test is run, and that it depends on the input size.

Q5-3 Index Size Vs. Time

What is the largest size index that can do lookups in about one second?

a. 200 000 keywords
b. 1 000 000 keywords
c. 10 000 000 keywords
d. 100 000 000 keywords
e. 1 000 000 000 keywords

Answer to Q-3

Q5-4 Lookup Time

Predict the lookup time for a particular keyword in an index created as given below. Be careful, this is a bit of a trick question.

What is the expected time to execute:

```
lookup(index10M, 'aaaaaaaa')
```

where index10M is an index created by

```
index10M = make_big_index(10000000)
```

a. 0.0 s
b. 0.1 s
c. 1.0 s
d. 10 s

Answer to Q-4

Worst Case

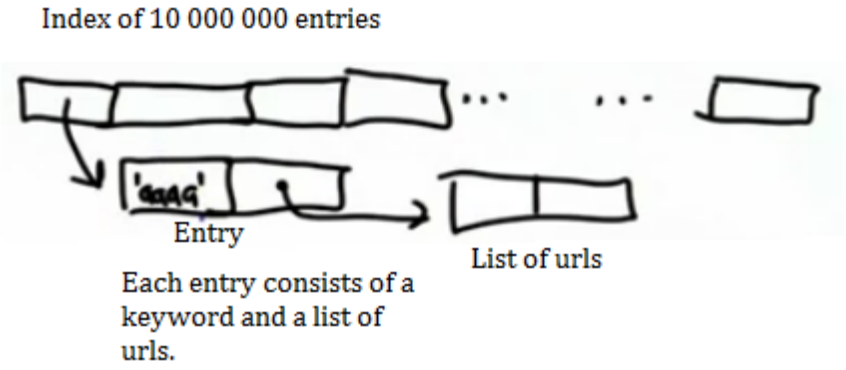
Usually, when analysing programs it's the worst-case execution time that is important. The **worst-case** execution time is the time it takes for the case where the input for a given size takes the longest to run. For **`lookup`**, it's when the **`keyword`** is the last entry in the index, or not in the index at all. Looking at the code will give you a better understanding of why the time scales as it does, and the worse-case and the average case running times.

This is the code from the last unit.

```
def add_to_index(index,keyword,url):
    for entry in index:
        if entry[0] == keyword:
            entry[1].append(url)
            return
    # not found, add new keyword to index
    index.append([keyword,[url]])

def lookup(index, keyword):
    for entry in index:
        if entry[0] == keyword:
            return entry[1]
    return None
```

What **`lookup`** does is go through the list. For each entry, it checks if it is equal to the keyword. Below is the structure of the index.



The number of iterations through the loop depends on the index. The length of the index is the maximum number of times the code goes through the loop. If the keyword is found early, the loop finishes sooner.

The other thing that is relevant is how **`add_to_index`** constructs the list. It loops through all the entries to see if the **keyword** exists, and if it doesn't, it adds the new entry to the end of the list. The first addition is added to the beginning of the **`index`** and the last to the end. That is

why '**aaaaaaaa**' is the first entry in the **index**.

Q5-5 Worst Case

Which keyword will have the worst case running time for a given index? (Choose one or more answers.)

a. **lookup(index, first word added)**
b. **lookup(index, word that is not in index)**
c. **lookup(index, last word added)**

Answer to Q-5

Q5-6 Fast Enough

This is a fairly subjective question.

Is our lookup fast enough?

a) Yes.
b) It depends on how many keywords there are.
c) It depends on how many urls there are.
d) It depends on how many lookups there are.
e) No.

Answer to Q-6

Making Lookup Faster

How can we make **lookup** faster? Why is it so slow?

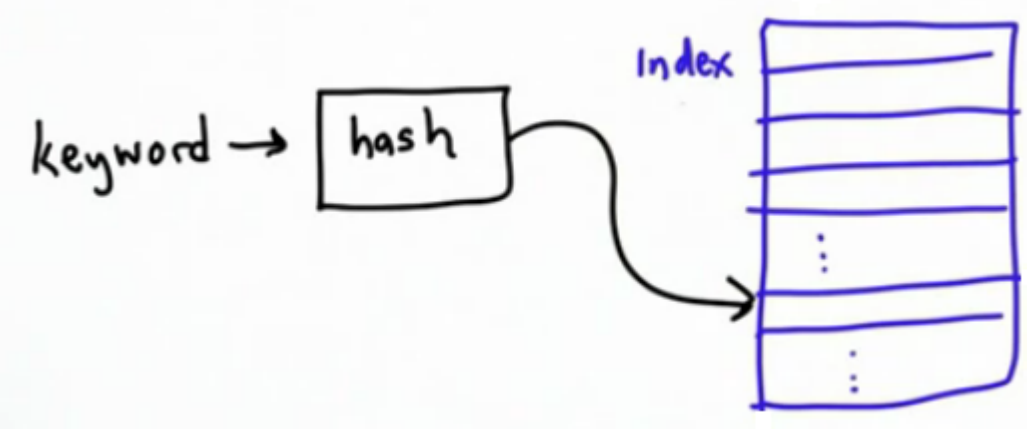
It's slow because it has to go through the whole of the **for** loop

```
for entry in index:
   if …
```

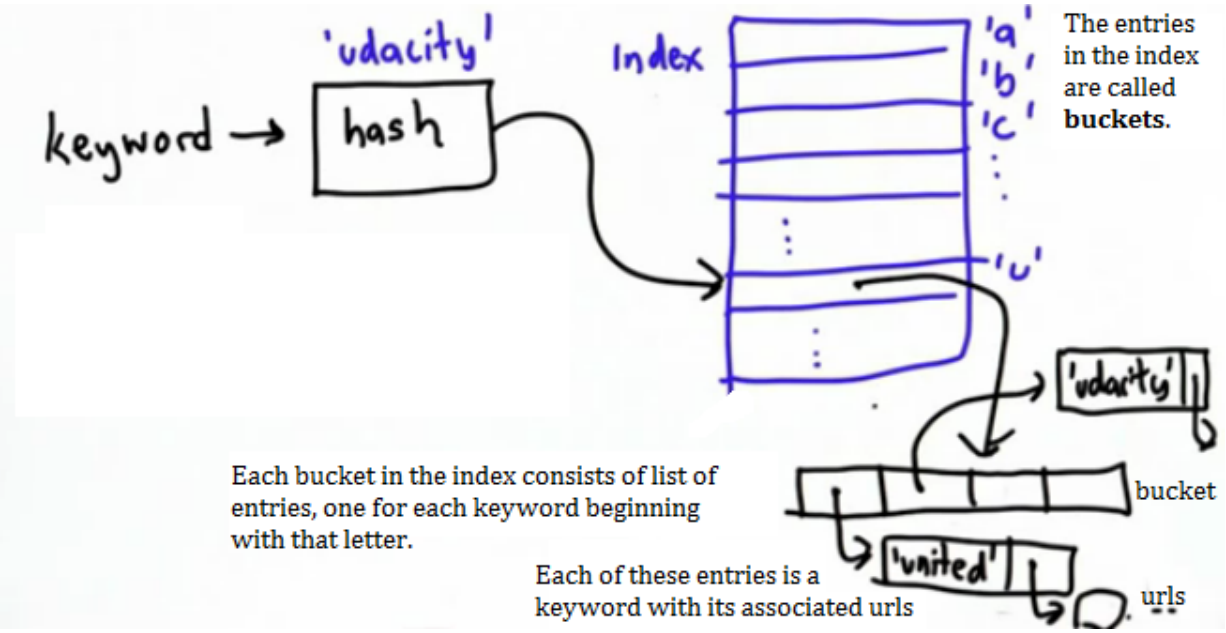
to find out if a keyword isn't there. This isn't how you use an index in real life. When you pick up a book, and look in the index, you don't start at A and work your way all the way through to Z to know if an entry isn't there. You go directly to where it should be. You can jump around the index because it is in sorted order. You know where the entry belongs because it is in alphabetic order. If it isn't where it belongs, it isn't in the index anywhere.

You could do something similar with the index in your code. If you had a sorted index, you could go directly to the entry you wanted. Sorting is an interesting problem, but it won't be covered in this course. Instead of having to keep the index in a specific order, you'll learn another way to know where to look for the entry you are interested in. This method will be a function, called a **hash function**. Given a keyword, the hash function will tell you where to look in the index. It will map the keyword to a number (this means it takes in a keyword and outputs a number) which is the

position in the index where you should look for the keyword. This means you don't have to start at the beginning and look all the way through the index to find the keyword you are looking for.



There are lots of different ways to do this. A simple way would be to base it on the first letter in each keyword. It would be like the way an index in a book works. Each entry in the index will correspond to a letter and will contain all the keywords beginning with that letter.



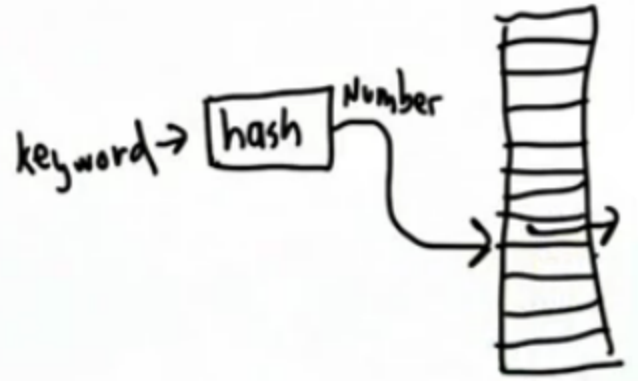
This isn't the best way to do it. It will reduce the time it takes to search through all the keywords as you'll only need to search through the keywords beginning with that letter, but it won't speed it up that much. The best it could do is to speed up the look up by a factor of 26, as there are 26 buckets, so each list would be 26 times smaller if all the buckets were the same size. It wouldn't be that good for English, since there are many more words beginning with S or T than there are beginning with X or Q, so the buckets are very different sizes. If you have millions of keywords, it will be faster, but not fast enough. There are two problems to fix:

- Make a function depending on the whole word
- Make the function distribute the keywords evenly between the buckets.

The table described is called a **hash table**. It's a very useful data structure. In fact, it is so useful that it's built into Python. There's a Python type (types are things like index and string) called the **dictionary**. It provides this functionality. At the end of this unit, you'll modify your search engine code to use the Python dictionary. However, before you learn about that, you'll learn to implement it yourself to make sure you understand how a hash table works.

Q5-7 Hash Table

Suppose we have b buckets and k keywords (k > b). Which of these properties should the hash function have? Recall that a hash function is a function which takes in a keyword and produces a number. That number gives the position in the table of the bucket where that input should be. (Choose all the properties the hash function should have.)

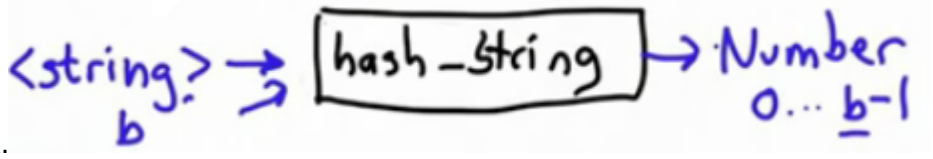


a. output a unique number between 0 and k-1.
b. output a unique number between 0 and b-1.
c. map approximately k/b keywords to bucket 0.
d. map approximately k/b keywords to bucket b-1.
e. map more keywords to bucket 0 than to bucket 1.

Answer to Q-7

Hash Function

Next, define a hash function which will satisfy those properties. It will be called **`hash_string`**. It takes as its inputs a string and the number of buckets, b, and outputs a number between 0 and b-1.



What you will need, which has not been discussed yet, is a way to turn a string into a number. You may recall that there were two operators in the code to generate the big index, **`make_big_index`**,

that were not explained. These were **ord** (for ordinal) and **chr** (for character). The operator **ord** turns a one-letter string into a number, and **chr** turns a number into a one-letter string.

```
ord(<one-letter string>) → Number
chr(<Number>) → <one-letter string>
```

Examples:

```
print ord('a')
97
print ord('A')
65
print ord('B')
66
print ord('b')
98
```

Note that upper and lower case letters are mapped to different letters. Also note that the number for **'B'** is higher than for **'A'**, and for **'b'** it is higher than for **'a'**.

A property of **ord** and **chr** is that they are **inverses**. This means that if you input a single letter string to **ord** and then input the answer to **chr**, you get back the original single letter. Similarly, if you input a number to **chr** and then input the answer to that into **ord**, you get back the original number (as long as that number is within a certain range.) This means that for any particular character, which we'll call alpha **α**, **chr(ord(α)) → α**. For example, in the code:

```
print chr(ord(u))
u

print ord(chr(117))
117
```

You can see below what happens if you enter a number which is too large.

```
print ord(chr(123456))
Traceback (most recent call last):
 File "C:\Users\Sarah\Documents\courses\python\testing.py", line 1, in
<module>
  print ord(chr(123456))
ValueError: chr() arg not in range(256)
```

From the last line of the error message, *ValueError: chr() arg not in range(256)*, you can see that **chr()** requires its input to be in the list of integers given by **range(256)**, which is a list of all the integers from 0 to 255 inclusive.

The numbers given by **ord** are based on ASCII character encoding. What these numbers are doesn't matter for the purpose of making a hash table. All that is important is that they are different for each letter. You'll be able to use **ord** to turn characters into numbers. The limitation of **ord** is that

it can only take one letter as input. If you try to input more, you'll get an error message.

```
print ord('udacity')
Traceback (most recent call last):
 File "C:\Users\Sarah\Documents\courses\python\testing.py", line 2, in
<module>
  print ord('udacity')
TypeError: ord() expected a character, but string of length 7 found
```

The last line tells you that you should just input a character, i.e. a single letter string, but instead you've input a string of length 7.

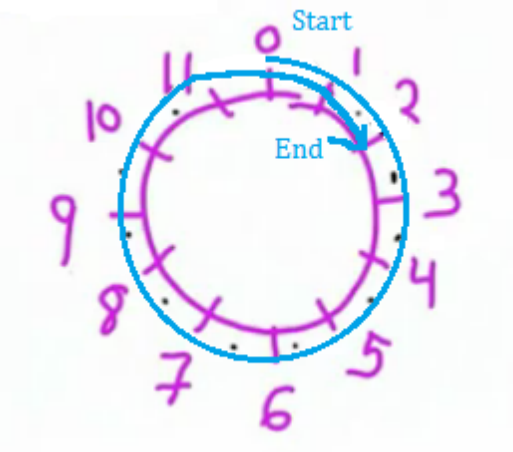
You now have a way to convert single character string to numbers which is **ord**. Next, you'll need a way to take care of the other property the hash function has to have which is to map the keywords to the range 0 to b-1.

Modulus Operator

The **modulus operator** (**%**) will be the tool used to change whatever values are calculated for strings into the range 0 to b-1. The modulus operator takes a number, divides it by the modulus and returns the remainder.

<*number*> % <*modulus*> → <*remainder*>

It's like clock arithmetic. The clock has **<*modulus*>** numbers on it. If you start counting from the top of a clock, and you count around **<*number*>** of steps, you arrive at the **<*remainder*>**. For example, if you want to work out **14 % 12**, you can think about a clock with 12 numbers. If you count 14 steps, you end up at 2, that is, **14 % 12 → 2**.



That's the same as the remainder when you divide 14 by 12. When you divide 14 by 12, you get a remainder of 2 since :

$$14 = 12 * 1 + 2$$

The remainder given by modulus is a number between **0** and ***<modulus> - 1***. This is good news when we're looking for something for our hash function to give us values between 0 and b-1, where b is the number of buckets.

In terms of the code:

```
print 14 % 12
2
```

Q5-8 Modulus Quiz

What is the value of each expression? Try to think of what the answer might be. You can also test it out in the Python interpreter.

a. **`12 % 3`**
b. **`ord('a') % ord('a')`** Try to do this without working out what the **`ord`** of **`'a'`** is.
c. **`(ord('z') + 3) % ord('z')`**

Answer to Q-8

Q5-9 Equivalent Expressions

Which of these expressions are always equivalent to x, where x is any integer between 0 and 10:

a. x % 7
b. x % 23
c. ord(chr(x))
d. chr(ord(x))

Answer to Q-9

Bad Hash

The **hash function** takes two inputs, the keyword and the number of buckets and outputs a number between zero and buckets minus one (b-1), which gives you the position where that string belongs.

You have seen that the function **`ord`** takes a one-letter string and maps that to a number.

And you have seen the **`modulus`** operator, which takes a number and a modulus and outputs the remainder that you get when you divide the number by the modulus.

You can use all of these to define a hash function. Here is an example of a bad hash function:

```
def bad_hash_string (keyword, buckets):
   return ord(keyword[0]) % buckets # output is the bucket based on the
```

```
first letter of the keyword
```

Q5-10: Bad Hash

Why is **bad_hash_string** a bad hash function?

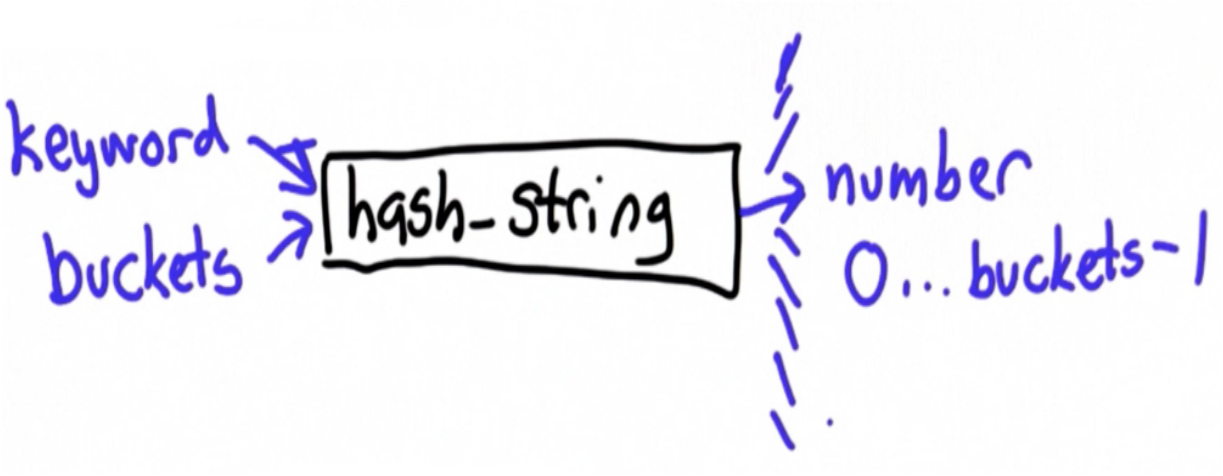
a. It takes too long to compute.
b. It produces an error for one input keyword.
c. If the keywords are distributed like words in English, some buckets will get too many words.
d. If the number of buckets is large, some buckets will not get any keywords.

Answer to Q-10

Better Hash Function

Now you know that looking at just the first letter does not work very well; it does not use enough buckets, nor does it distribute the keys well. Here is how you can make a better hash function:

Begin with the same properties you had before, a function with two inputs, the keyword and the number of buckets. The output is the hash value in the range of zero to number of buckets minus 1. The goal is for the keywords to be well distributed across the buckets and every time you hash the same keyword, you will get the same bucket to know quickly where to find it.



In order to make a more robust hash function, you are going to want to look at more than just one letter of the keyword. You want to look at all the letters of the keyword and based on all the letters you want to decide their bucket.

Recall that when you have a list of items you can use the **for** loop to go through the elements in the list:

```
p = ['a', 'b', …]
for e in p:
   <block>
```

This is also the same for strings:

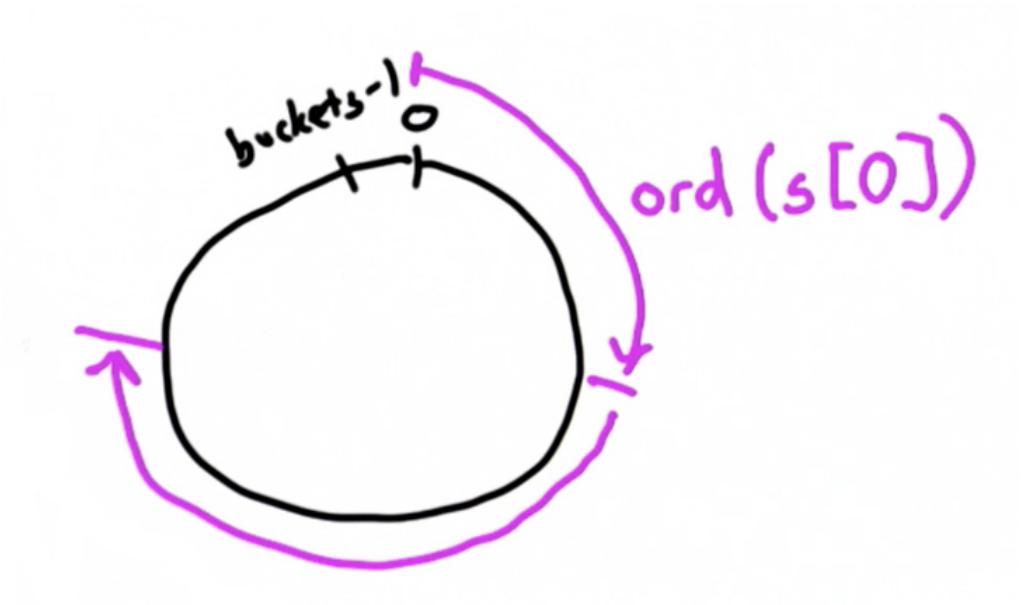
```
s = "abcd"
for c in s:
   <block>
```

This gives you a way to go through all of the elements in a string. Also recall how you turned single letter strings into numbers with an modulo arithmetic, then you know enough to define a much better hash function.

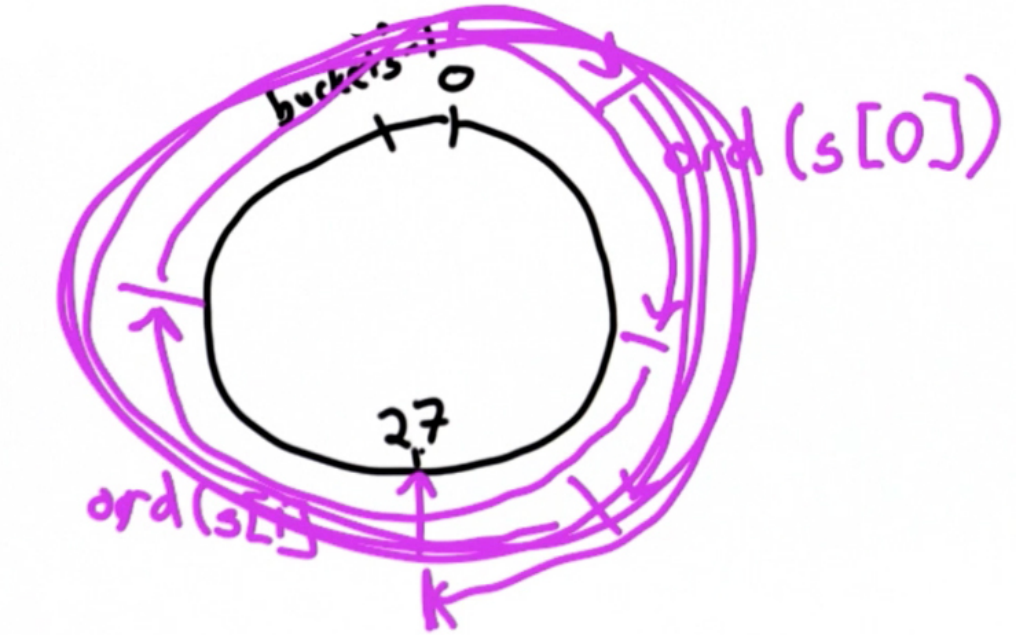
Q5-11: Better Hash Functions

Define a function, **`hash_string`**, that takes as its input a keyword (string) and a number of buckets, and outputs a number representing the bucket for that keyword. Do this in such a way that it depends on all the characters in the string, not just the first character. Since there are many ways to do this, here are some specifications:

Make the output of hash_string a function of all the characters. You can think about this in terms of modulo arithmetic:



The image shows a circle, which is the size of the number of buckets, and goes from zero to the number of buckets minus 1. For each character start at zero and go around the order of that character, **`ord(s[0])`**, distance around the circle. Then, keep going so that for each character you go some distance around the circle. The circle can be any size depending on the number of buckets, for example 27.



Examples:

Which bucket will **a** and **b** land in?

```
hash_string('a', 12) = 1 # 11 will be the last bucket
   ord('a') = 97 # go around the circle eight times, b/c 97 is 12 * 8 + 1


hash_string('b', 12) = 2
   ord('b') = 98
```

The size of the hash table matters as well as the string:

```
hash_string('a', 13) = 6 # this is because 97 = 13 * 7 + 6
```

What about multi-letter strings?

```
hash_string('au, 12') => 10
   ord('a') = 97
   ord('u') = 117 # add to 97 and modulo the sum, 214 to the number of
buckets, 12, which equals 10

hash_string('udacity', 12) => 11
```

Now try the quiz!

Answer to Q-11

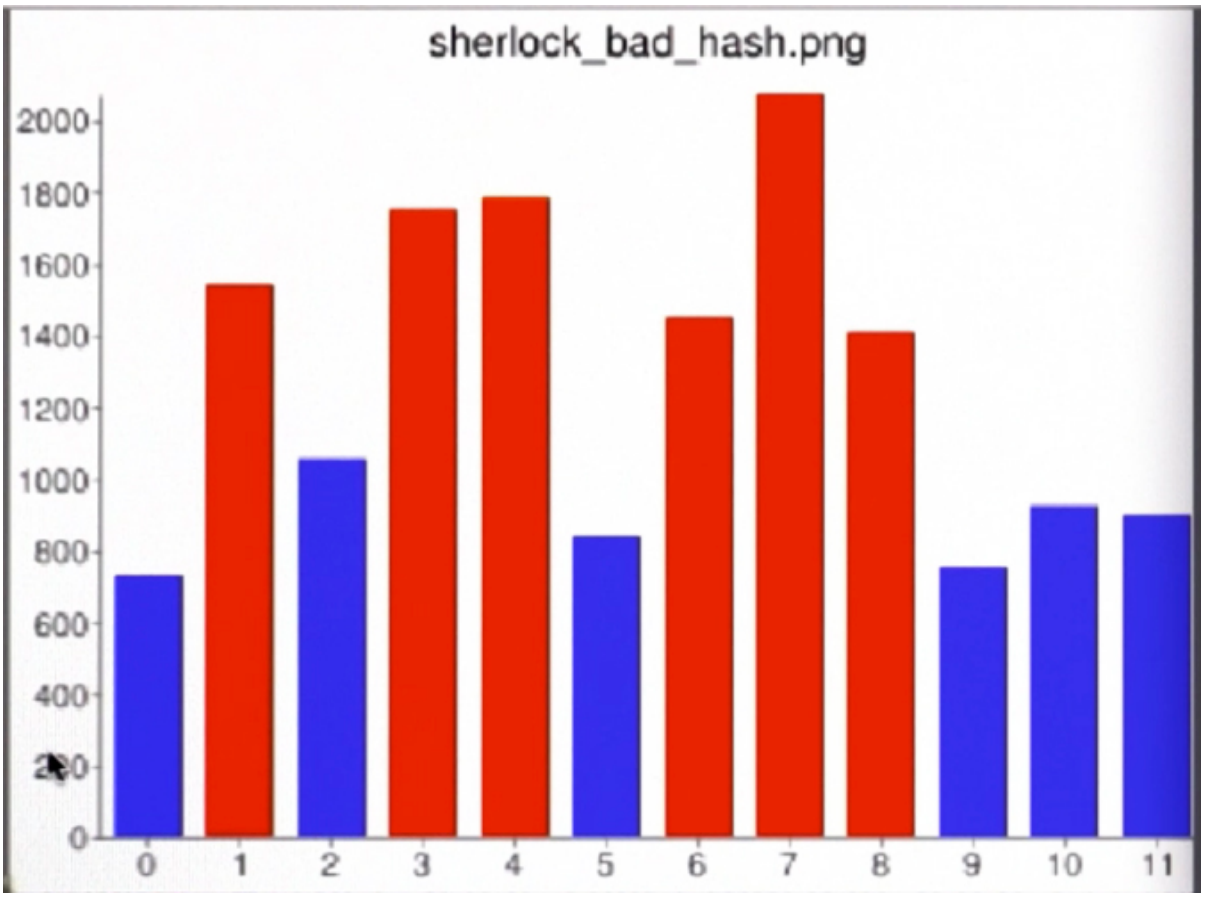
Testing Hash Functions

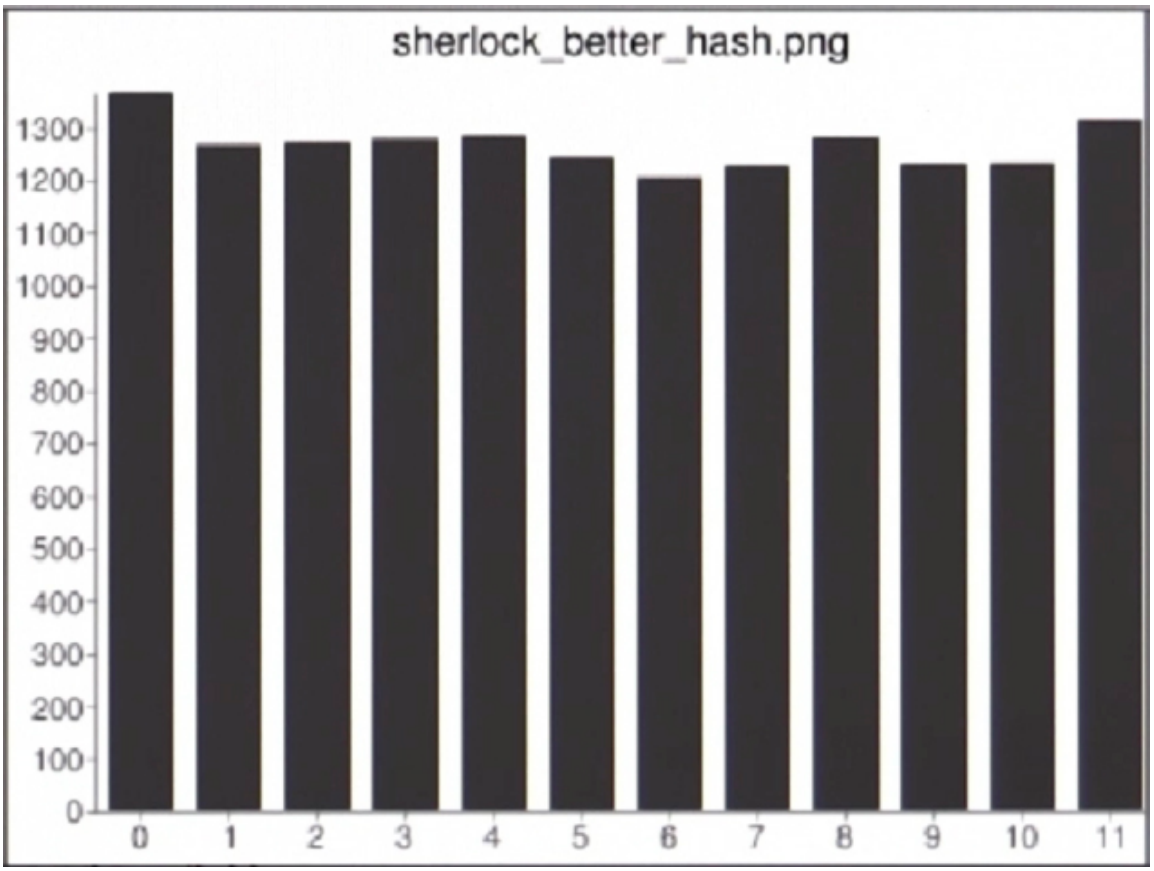
Test your new hash function to make sure it does better than the bad hash function, using the same url from the example before.

```
def test_hash_function(func, keys, size):
   results = [0] * size
   keys_used = []
   for w in keys:
       if w not in keys_used:
           hv = func(w, size)
           results[hv] += 1
           keys_used.append(w)
   return results

words = get_page('http://www.gutenberg.org/cache/epub/1661/
pg1661.txt').split() # initialize all the words from the
page 'Adventures of Sherlock Holmes'
counts = test_hash_function(bad_hash_string, words, 12) # obtain the
counts for the old string
print counts
[725, 1509, 1066, 1622, 1764, 834, 1457, 2065, 1398, 750, 1045, 935]
counts = test_hash_function(hash_string, words, 12) # find the
distribution for the new function
[1363, 1235, 1252, 1257, 1285, 1256, 1219, 1252, 1290, 1241, 1217, 1303]
```

Have a look at the distribution of the keywords into the buckets. Compare the first function, **bad_hash_string**, to the new function, **hash_string**:


sherlock_bad_hash.png
2000
1800
1600
1400
1200
1000
800
600
400
200
0
0
1
2
3
4
5
6
7
8
9
10
11



Now try changing the number of buckets:

```
>>> counts = test_hash_function(hash_string, words, 100)
>>> print counts
[152, 135, 113, 142, 145, 153, 123, 114, 125, 126, 146, 136, 147, 120, 141, 134, 142, 144, 1
40, 135, 126, 104, 136, 136, 131, 153, 142, 169, 136, 145, 158, 149, 175, 141, 142, 175, 145
, 157, 153, 153, 168, 148, 182, 154, 177, 163, 165, 138, 163, 157, 149, 154, 166, 173, 159,
162, 185, 158, 165, 172, 171, 159, 139, 152, 167, 150, 143, 151, 154, 174, 129, 184, 164, 17
6, 145, 159, 161, 149, 151, 163, 163, 151, 170, 156, 197, 160, 172, 142, 189, 141, 159, 155,
128, 139, 126, 164, 161, 156, 140, 163]
```

Building a good hash function is a very difficult problem. As your tables get larger it is very important to have an efficient hash function and while yours is not the most efficient, it is going to work for your purposes. Check out the website for more examples of how to build an even better hash function.

Q5-12: Keywords and Buckets

Assuming our hash function distributes keys perfectly evenly across the buckets, which of the following leaves the time to lookup a keyword essentially unchanged?

There may be more than one answer.

a. double the number of keywords, same # of buckets
b. same number of keywords, double # of buckets
c. double number of keywords, double # of buckets
d. halve number of keywords, same # of buckets
e. halve number of keywords, halve # of buckets

Answer to Q-12

Implementing Hash Tables

By now you should understand that the goal of a hash table is to map a keyword and a number of buckets using a **`hash_string`** function, to a particular bucket, which will contain all of the keywords that map to that location.

Now, try and write the code to do this! You can start with the index you wrote for the previous unit and try to figure out how to implement that with a hash table.

How is this going to change your data structure? Recall the data structure from the last class:

Q5-13: Implementing Hash Tables

What data structure should we use to implement our hash table index?

a. [[<word>, [<url>, ...]], ...]
b. [[<word>, [[<url>, ...], [<url>, ...]], ...]
c. [[[<word>, [<url>, ...]], [<word>, [<url>, ...]], ...], ...]
d. [[[<word>, <word>, ...], [<url>, ...]], ...]

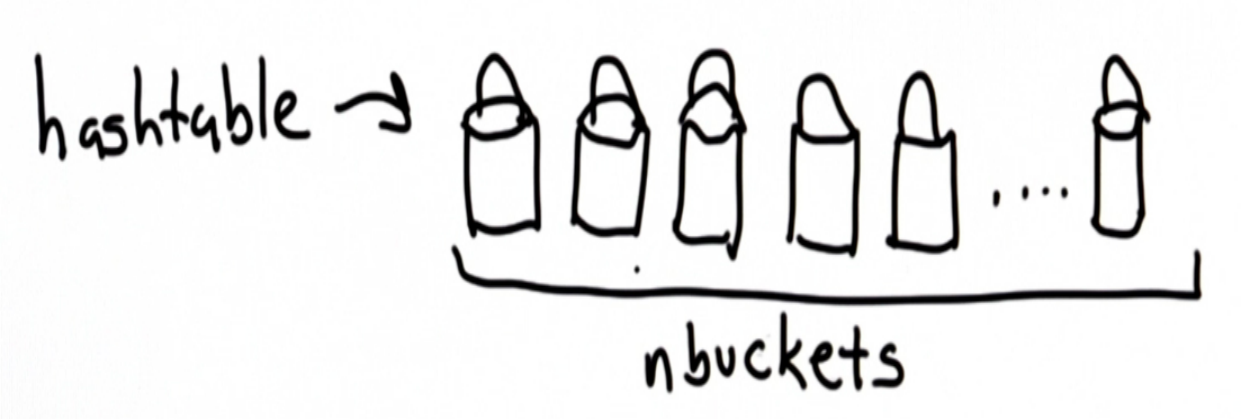
Answer to Q-13

Empty Hash Table

The first thing you want to do to implement your hash table index, is figure out how to create an empty hash table. With a simple index, this was really easy, just make an empty list and as you add elements to the list, just add them to the empty list.

```
index = []
```

Unfortunately, this will not work for an empty hash table because you need to start with all the buckets. The initial value for the hash table needs to be a set of empty buckets so that you can do lookups right away. You also want to be able to add elements to your hash table.



If you just started with an empty list, then the first time you looked up a keyword it would say that keyword belongs in bucket 27, but since you don't have a bucket for that you would have to figure out how to create that bucket. It makes more sense to start by making an empty hash table be a list of buckets, where initially all the buckets are empty, ready and waiting for keywords to be dropped in them. So, what you need is code to create the empty hash table.

Q5-14: Empty Hash Table

Define a procedure, **`make_hashtable`**, that takes as input a number, **`nbuckets`**, and outputs an empty hash table with **`nbuckets`** empty buckets.

Answer to Q-14

Q5-15: The Hard Way

Why does **`[[]] * nbuckets`** not work to produce our empty hash table?

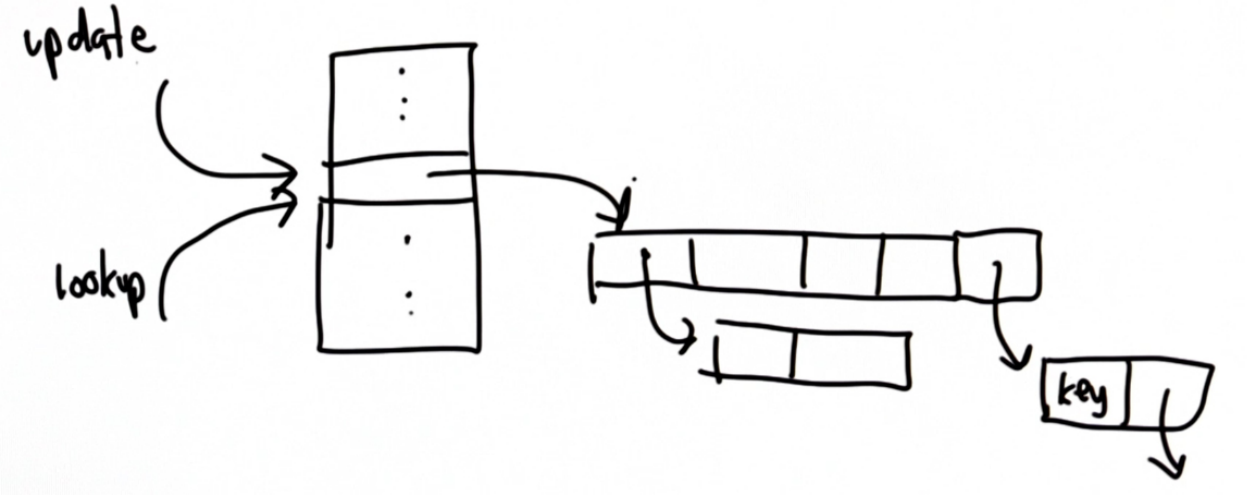
a. Because it is too easy and we like doing things the hard way.
b. Because each element in the output refers to the same empty list.
c. Because ***** for lists means something different than it does for strings.

Answer to Q-15

Finding Buckets

Both operations you will perform on a hash table, i.e. lookup (read) and add (write), depending on first being able to find the right bucket:

- **lookup:** In order to find desired value, you need to know, which bucket to look into.
- **add:** In order to add a value (or overwrite existing one associated with the key, already present in hash table), you need to know which bucket to add your value to (or where to look for the value being overwritten).



Q5-16: Finding Buckets

Define a procedure, **`hashtable_get_bucket`**, that takes two inputs – a hash table and a keyword – and outputs the bucket where the keyword could occur. Note that the bucket returned by the procedure may not contain the searched for keyword in case the keyword is not present in the hash table:

```
hashtable_get_bucket(hashtable, keyword) # => list: bucket
```

Function **`hash_string`**, defined earlier, may prove useful:

```
hash_string(keyword, nbuckets) # => number: index of bucket
```

Note that there is a little mismatch between those two functions. The function **`hashtable_get_bucket`** takes hash table and the searched for keyword as its arguments, while the function **`hash_string`** takes the keyword and **`nbuckets`**, which is a number (size of the hash table).

The former function does not have any input like this, which means that you will need to determine the size of hash table yourself.

Hint: whole implementation of **`hashtable_get_bucket`** function might be done on a single line.

Answer to Q-16

Q5-17: Adding Keywords

Define a procedure:

```
hashtable_add(htable, key, value)
```

that adds the key to the hash table (in the correct bucket). Create a new record in the hash table even if the given key already exists. Also note that records in the hash table are lists composed of

two elements, where the first one is the key and the second one is the value associated with that key (i.e. ***key-value pairs***):

```
[key, value]
```

Answer to Q-17

Q5-18: Lookup

Define a procedure:

```
hashtable_lookup(htable, key)
```

that takes two inputs, a hash table and a key (`string`), and outputs the value associated with that key. If the key is not in the table, output `None`.

Answer to Q-18

Q5-19: Update

Define a procedure:

```
hashtable_update(htable, key, vaue)
```

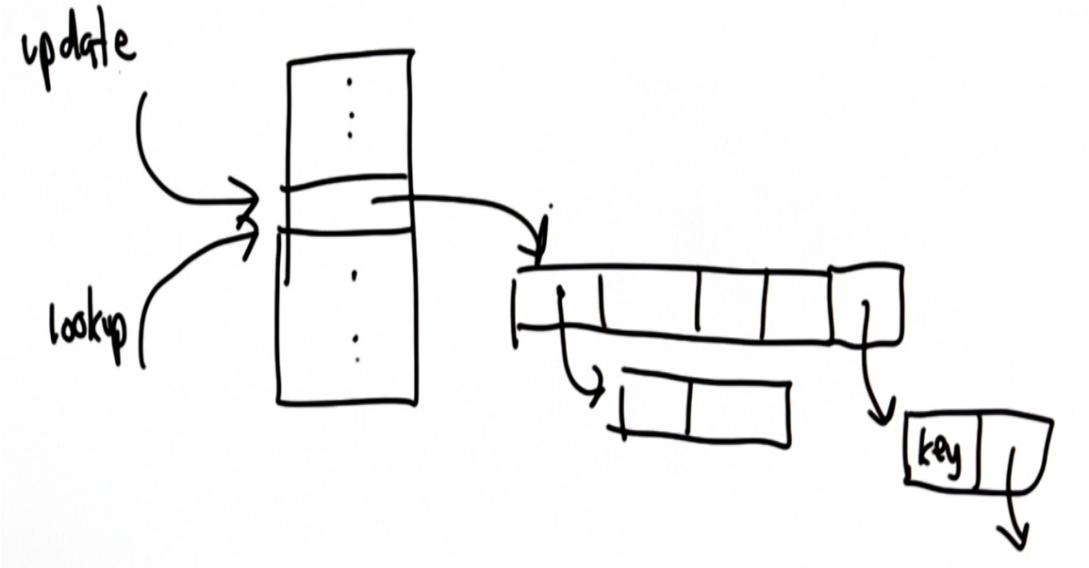
that updates the value associated with key. If key is already in the table, change the value to the new value. Otherwise add a new entry for the key and value.

Answer to Q-19

Operations on Hash Table

You will perform both operations on a hash table, i.e. lookup (read) and add (write), depend on first being able to find the right bucket:

- **lookup:** In order to find desired value, you need to know, which bucket to look into.
- **add:** In order to add a value (or overwrite existing one associated with the key, already present in hash table), you need to know which bucket to add your value to (or where to look for the value being overwritten).



Dictionaries

Now that you've built a hash table for yourself, you'll see an easier way to do it using the built-in Python type **dictionary**. It's an implementation of a hash table, and it's easier to use than defining your own hash table.

You've already seen two complex types in Python, **string** type and **list** type. The **dictionary** type is another. These three types have many things in common but other things are different.

To create a string, you have a sequence of characters inside quotes ''. To create a list, you use square brackets [] and have a sequence of elements inside the square brackets, separated by commas. The elements can be of any type, unlike with a string which has to be made up of characters. The dictionary type is created using curly brackets **`{}`**, and consists of **`key:value`** pairs. The keys can be any immutable type, such as a string or a number, and the values can be of any type.

Recall a strings is immutable which means that once it is created, the characters can not be changed. A list is mutable, which means it can be changed once it's been created. A dictionary is also mutable and its **`key:value`** pairs are updated like in your update function for hash tables. In other words, if the key isn't in the list, it's created, and if it is, it's changed to the new value. The differences, and similarities between a string, list and dictionary are summarised below.

String	List	Dictionary
`'hello'`	`['alpha', 23]`	`{'hydrogen':1, 'helium':2}`
sequence of characters	list of elements	set of <key:value> pairs
immutable	mutable	mutable
`s[i]` ithcharacter of s	**`p[i]`** ithelement of p	**`d[k]`** value associated with the key k

Characters can't be replaced as strings are immutable.	**p[i] = u** replaces the value of the ithelement of p with u.	**d[k] = v** updates the value associated with k to be v.

Using Dictionaries

You can create a dictionary using **{}**. For the example below, the **key:value** pairs are the chemical elements along with their atomic numbers.

```
elements = { 'hydrogen': 1, 'helium': 2, 'carbon': 6 }

print elements
{'helium': 2, 'hydrogen': 1, 'carbon': 6}
```

Unlike with a list, when the elements are printed out, the **key:value** pairs can be in a different order from the order they were entered into the dictionary. When you made a hash table and put elements into it, you saw that the position of the elements in the table was not necessarily the same order as they were entered as it depended on the key and the hash function. You see the same thing with the dictionary as with your hash table as the dictionary is implemented like a hash table.

When you look up a chemical element in the dictionary, the value associated with that element is returned.

```
print elements['hydrogen']
1
print elements['carbon']
6
```

What do you think will happen when you try to lookup an element which is not in the list?

```
print elements['lithium']
Traceback (most recent call last):
 File "C:\Users\Sarah\Documents\courses\python\dummy.py", line 5, in
<module>
  print elements['lithium']
KeyError: 'lithium'
```

You get a *KeyError* which tells you that *'lithium'* is not in the dictionary. Unlike in your lookup where you defined it to return **None** if the key is not there, you get an error if a key is not in a dictionary and you try to do a lookup on it.

To prevent this error, you can check if a **key** is in the dictionary using **in,** just like with lists. Just like for lists, it will return **True** if the key in in the list and **False** if it is not.

```
print 'lithium' in elements
False
```

As a dictionary is mutable, it can be added to and changed. Using **`elements['lithium']`** on the left hand side of an assignment does not cause an error even though **`'lithium'`** is not in the dictionary. Instead it adds the **`key:value`** pair **`'lithium':3`** to the dictionary.

```
elements['lithium'] = 3
elements['nitrogen'] = 8

print elements
{'helium': 2, 'lithium': 3, 'hydrogen': 1, 'nitrogen': 8, 'carbon': 6}

print element['nitrogen']
8
```

Although this gives the output expected, the atomic number of **`'nitrogen'`** is actually 7 and not 8, so it needs to be changed. As **`'nitrogen'`** is already in the dictionary, this time

```
elements['nitrogen'] = 7
```

doesn't create a new **`key:value`** pair, but instead it updates the value associated with **`'nitrogen'`**. To see that, you can see print the value

```
print element['nitrogen']
7
```

and the complete dictionary.

```
print elements
{'helium': 2, 'lithium': 3, 'hydrogen': 1, 'nitrogen': 7, 'carbon': 6}
```

Q5-20: Population

Define a dictionary, **`population`**, that provides information on the world's largest cities. The key is the name of a city (a string), and the associated value is its population in millions.

Shanghai	17.8
Istanbul	13.3
Karachi	13.0
Mumbai	12.5

If you don't happen to live in one of those cities, you might also like to add your hometown and its population or any other cities you might be interested in. If you define your dictionary correctly, you should be able to test it using **`print population['Mumbai']`** for which you should get the output *`12.5`*.

Answer to Q-20

A Noble Gas

Now to return to the elements dictionary, but this time, to make it more interesting. The chemical element dictionary from earlier just contains elements and their atomic numbers.

```
elements = { 'hydrogen': 1, 'helium': 2, 'carbon': 6 }

elements['lithium'] = 3
elements['nitrogen'] = 8

print elements['nitrogen']
elements['nitrogen'] = 7
print elements['nitrogen']
```

Values don't have to be numbers or strings. They can be anything you want. They can even be other dictionaries. The next example will have atomic symbols as keys with associated values which are dictionaries. First, an empty dictionary is created and then hydrogen, with key **'H'** and helium, with key **'He'** are added to it.

```
elements = {}
elements['H'] = {'name': 'Hydrogen', 'number': 1, 'weight': 1.00794}
elements['He'] = {'name': 'Helium', 'number': 2, 'weight': 4.002602,
             'noble gas': True}
```

The code:

```
elements['H'] = {'name': 'Hydrogen', 'number': 1, 'weight': 1.00794}
```

sets **'H'** as the key with associated value the dictionary **{'name': 'Hydrogen', 'number': 1, 'weight': 1.00794}**. This dictionary has three entries with the keys **'name'**, **'number'** and **'weight'**. For helium, **'He'** is the key and a dictionary containing the same keys as **'H'** but with an extra entry which has key **'noble gas'** and value **True**.

The dictionary of **key:value** pairs, **atomic_symbol:{dictionary}** is shown below.

```
print elements
{'H': {'name': 'Hydrogen', 'weight': 1.00794, 'number': 1},
'He': {'noble gas': True, 'name': 'Helium', 'weight':
4.002602, 'number': 2}}
```

To see the element hydrogen, look up its key which is **'H'**.

```
print elements['H']
{'name': 'Hydrogen', 'weight': 1.00794, 'number': 1}
```

Note that the elements appear in a different order from the order they were input as **elements['H']** is a dictionary.

To look up the name of the element with symbol H, use

```
print elements['H']['name']
Hydrogen
```

where **elements['H']** is itself a dictionary and **'name'** is one of its keys.

You could also lookup the weights of hydrogen and of helium, or check if helium is a noble gas.

```
print elements['H']['weight']
1.00794
print elements['He']['weight']
4.002602
print elements['He']['noble gas']
True
```

What happens if you try to check if hydrogen is a noble gas?

```
print elements['H']['noble gas']
Traceback (most recent call last):
 File "C:\Users\Sarah\Documents\courses\python\dummy.py", line 11, in
<module>
  print elements['H']['noble gas']
KeyError: 'noble gas'
```

It's the same error as for the attempted lookup of lithium in the dictionary of elements which didn't include it. It tries to look up **'noble gas'** but it doesn't exist in the dictionary which is associated with the key **'H'**.

Modifying the Search Engine

Modifying the search engine code from the previous unit to use dictionary indexes instead of list indexes has the advantage of doing lookups in constant time rather than linear time.

Q5-21: Modifying the Search Engine

Which of the procedures in your search engine do you need to change to make use of a dictionary index instead of a list index?

a. get_all_links
b. crawl_web
c. add_page_to_index
d. add_to_index
e. lookup

Answer to Q-21

Q5-22: Changing Lookup

Change the lookup procedure to use a dictionary rather than a list index. This does not require any loop. Make sure an error is not raised if the keyword is not in the index; in that case, return None.

Answer to Q-22

Changing Lookup

Congratulations! You have completed unit 5. You now have a search engine that can respond to queries quickly, no matter how large the index gets. You did so by replacing the list data structure with a hash table, which can respond to a query in a time that does not increase even if the index increases.

The problem of measuring cost, analyzing algorithms and designing algorithms that work well when the input size scales, is one of the most important and interesting problems in computer science.

Now, all you have left to do for your search engine is to figure out a way to find the best page for a given query instead of just finding all of the pages that have that keyword. This is what you will learn in unit 6.

Answer Key

A5-1: Measuring Speed

The correct answers are a, b, c.

a. We want to predict how long it will take for a program to execute before running it. If we have to run the program to figure out how long it takes, then we've already done what we wanted so there is no point predicting. Furthermore, we've only found out how long it takes to run for that particular input. It won't be very useful to tell us how long it will take to run it for a different input.

b. We want to know how the time will change as computers get faster. By understanding how time scales with input size, that is how the time increases as the input size increases, we get a better idea how costs will change with time. Computers keep getting cheaper and faster. This was observed by Gordon Moore in 1965 and is known as Moore's Law. It's not a physical law but a trend that has been followed throughout the history of computing. Computing power doubles approximately every 18 months, so what you can get now will be about half the power you will be able to get in a year and a half for the same amount of money. That's a pretty nice property but knowing the monetary cost is today doesn't tell us much about the cost in a year's time. Understanding the cost in a more fundamental way is needed.

c. We want to understand fundamental properties of our algorithms, not things specific to a particular input or machine. Again, this is correct. Understanding fundamental properties will give us much more information that a few measurements from particular inputs and computers.

d. We want abstract answers, so they can never be wrong. Abstract answers *can* be just as wrong as concrete answers, but having abstract answers can help us understand in a more fundamental way than just concrete answers.

A5-2: Predicting Run Time

n	time (in seconds)
10^4	**0.0005**
10^5	**0.005**
10^6	**0.05**
10^7	**?**
10^8	**?**
10^9	**?**

Looking at the values in the table, you can see that when n is increased by a factor of 10 (multiplied by 10), then so is the running time, like this:

n: 10^4 * 10 = 10^5, running time: 0.0005*10 = 0.005 s,
n: 10^5 * 10 = 10^6, running time: 0.005*10 = 0.05 s.

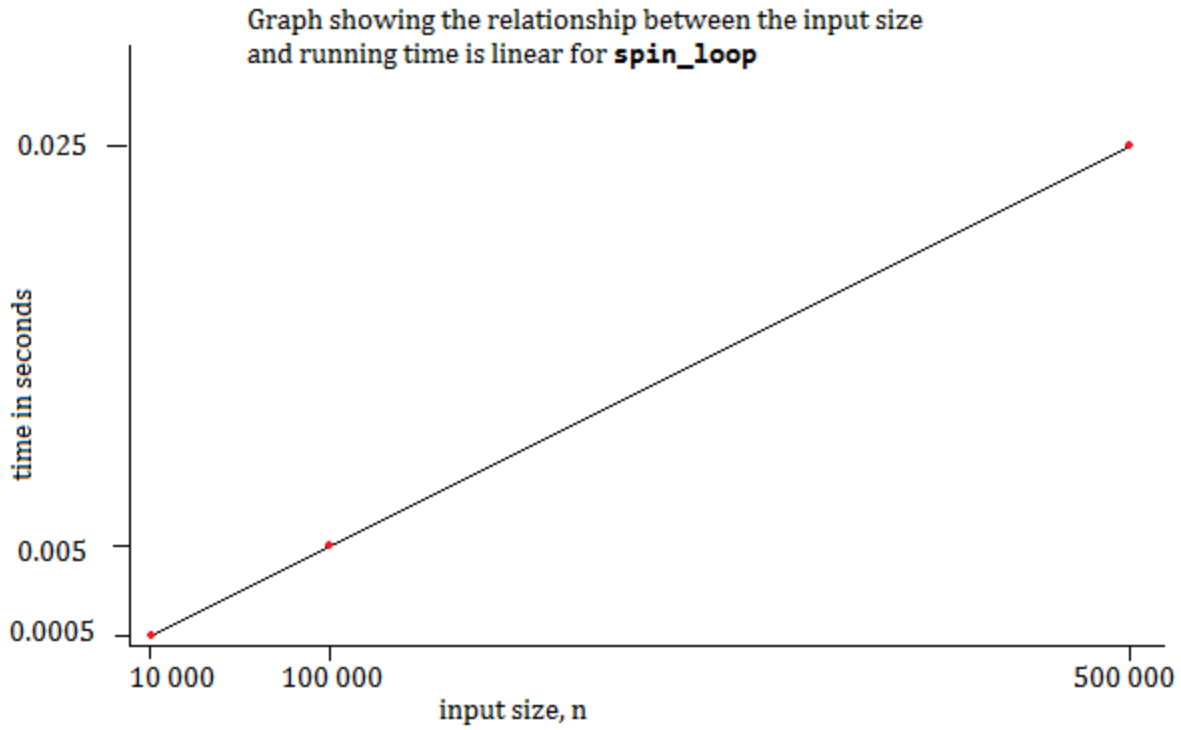
So you can predict that going from one million to ten million, which is an increase by a factor of 10, that the running time will also increase by a factor of 10.

n: 10^6 * 10 = 10^7, predicted running time: 0.05*10 = 0.5 s

Continuing in this way,

n: 10^7 * 10 = 10^8, predicted running time: 0.5*10 = 5 s
n: 10^8 * 10 = 10^9, predicted running time: 5*10 = 50 s, which is the value you were trying to predict.

When the running time increases by the same factor as the size of the input, it is called **linear.** If we plot running time against the size of the input on a graph, we get a straight line. Unfortunately, the graph would have to be very large to plot all the points up to a billion, so the graph below only shows the points up to half a million. (For half a million, we can estimate by take the running time 0.005s for 100 000 and multiply it by 5, since 100 000 * 5 = 500 000. That means the predicted running time is 0.005*5 = 0.025 s)



A5-3: Index Size vs. Time

The answer is c) 10 000 000 keywords. This required some guess work as you haven't seen any examples of this size, but from the earlier executions for 10 000 and 100 000, you can see that the increase in both the number of keywords and the time is by a factor of 10. It's linear just like the **`spin_loop`** was.

keywords	time
10 000	0.0009 s
100 000	0.009 s
1 000 000	0.09 s
10 000 000	0.9 s ~ 1 s
100 000 000	9 s

x 10

A5-4: Lookup Time

The correct answer is a) 0.0 s. It won't be exactly no time, but very close. This is partly because of the way in which **`index`** is constructed and partly because of how **`lookup`** works. The index is constructed so that the first entry in the index is always '**`aaaaaaaa`**', and the lookup always starts at the beginning of the index and goes through in order, so it finds '**`aaaaaaaa`**' immediately.

The point of this quiz was to show that it's not only the size of the input that can matter in determining the run time, but also the actual input. Because the word was at the beginning of the **`index`**, the **`lookup`** was very fast.

A5-5: Worst Case
The last two answers, b, and c are correct. Both these will need to go through every element in the index, which is **`len(index)`** times.

In the case **`lookup(index, word that is not in index)`**, the program goes through every entry and then returns **`None`**. For **`lookup(index, last word added)`**, it goes through every entry, finds a match to the last one and then returns the urls associated with that entry.

Note that this assumes that the length of time to do the comparisons between each entry and the keyword does not depend on what the entry and keyword strings are. Fortunately, for strings in Python, this is the case. String comparisons, that is **`==`** between strings, can be done very quickly. This is because strings are immutable which means that a comparison does not not to look at every letter individually.

All the other operations in the loop have constant time, that is, they don't depend on the input size. This is important because if any of them take a longer time, dependent on the input size, the running time of the program will change and may get much much slower as the size of the input increases.

In summary, the important thing to understand is that the number of iterations of the loop depends on the input size, and that everything in the loop has constant time.

For now the goal is to develop intuition. In a later course, you will be able to learn about how to analyse algorithms in more detail.

A5-6: Fast Enough
The best answer to this question is that it does depend on b, the number of keywords and on d, the number of lookups.

b. The number of times iterated through the loop only depends on the number of keywords because it's the keywords that are being searched. If there are very few keywords, then the lookup is fast enough, but for a search engine there will be very many keywords. Then it isn't fast enough.

c. It does not depend on the number of urls. More pages will likely mean more keywords, but the number of urls in the list does not affect the time taken for the lookup.

d. If there are only a few lookups, then it's probably not worth the time to improve it but it's more likely you'll want to do many lookups. Certainly, if you want to build a real search engine on the web, then you will want it to handle many millions of lookups per day. In the case of Google, there are billions of lookups per *hour.* If you want to do many lookups, you want lookup to be faster.

There are good reasons to want to make lookup faster. In learning how to make this improvement to lookup, you'll meet some interesting computer science concepts.

A5-7: Hash Table

Many of the properties are desirable. The correct answer is b, c, d.

a. Output a unique number between 0 and k-1. This is not desirable as the range for the hash table would be very large and so the hash table would take up a huge amount of memory to store. There would be the same number of buckets as keywords, which isn't going to work very well.

b. Output a unique number between 0 and b-1. This is desirable as each keyword will map to one of the b buckets.

c. Map approximately k/b keywords to bucket 0. All the buckets should have approximately the same number of entries. For b buckets and k keywords, there will be k/b keywords in each bucket.

d. Map approximately k/b keywords to bucket b-1. For the same reason as c), this is desirable.

e. Map more keywords to bucket 0 than to bucket 1. You might think that it would be better to have more keywords early on in the list, but for a hash table this isn't the case as you go directly to the bucket you are interested in. This is different from the list index where the first entry is the fastest to find. There is no reason to prefer having more entries in any one bucket than any other and in fact, a **uniform distribution** (which means the same number of entries in each bucket) is best.

A5-8: Modulus Quiz

a. 0 b. 0 c. 3

a. Since 12 = 4*3, the result when you divide 12 by 3 is 4 with no remainder, so the result of **`12 % 3`** is 0.

b. You could do this by figuring out the value of **`ord('a')`** but you don't need to. Any number (except zero) is divisible by itself with no remainder, so you don't need to know the value of **`ord('a')`** to know the remainder is 0 when **`ord('a')`** is divided by itself.

c. To do this question, you need to know whether **`ord('z')`** is greater than 3 or not. As long as **`ord('z') is greater than 3 then the remainder when ord('z') + 3`** is divided by **`ord('z')`** is 3 This is because for any number x greater than 3, you can write x + 3 = x * 1 + 3, where 3 is the remainder. As z is later in the alphabet than u and **`ord('u')`** is 117, z is definitely greater than 3 and so **`(ord('z') + 3) % ord('z')`** is 3.

```
print (ord('z') + 3) % ord('z')
3
```

Maybe you noticed the extra parentheses in **`(ord('z') + 3) % ord('z')`**? Do they really matter?

```
print ord('z') + 3 % ord('z')
125
```

It certainly looks like it! What is going on here? The grouping is different in the second version without the parentheses. First it does **`3 % ord('z')`** which is 3, and then it adds that to the value of **`ord('z')`** which is 122 giving a total of 3 + 122 = 125.

```
print ord('z')
122
```

So parentheses really do matter when you're using modulus.

A5-9: Equivalent Expressions

B and c are the two correct answers.

A is not the right answer because if x is seven or greater, then 7%7 has the value zero, and 8%7 has the value one, which is not what you started with.

B is correct because when the modulo is greater that the possible value of x, which, in this case was any integer between zero and ten, then the result is always the same as x.

C is correct because when you map x to its character value and then take the order of that, char and ord inverse, making it equivalent.

D is incorrect. Although you might think that this would work in the opposite direction it does not. This is because the input to ord must be a one letter string. If the input is a single character then ord produces an error.

You can see this in the Python interpreter, try:

```
print ord(3)
Traceback (most recent call last):
 File "/code/knowvm/input/test.py", line 1, in <module>
     print ord(3)
TypeError: ord() expected string of length 1, but int found
```

There is a function that allows you to turn functions into strings. The **str** function takes a number and gives you a string corresponding to that number:

```
print str(3)
3
```

You can use **ord** on the string function:

```
print ord(str(3))
51
```

This result is because it is the same as saying:

```
print ord('3')
51
```

A5-10: Bad Hash

All of these choices are true, except for a, because the function doesn't really take that long to compute.

When you write code you should think about what works for all possible inputs. The boundary case can be difficult to think about. For a string, this is often the empty string. So, if you pass in a string with no characters in it (which is a perfectly valid string), then when you try to index element zero you would get an error. Try this in the interpreter.

Answers b and c are correct, and you can understand why by looking at this code:

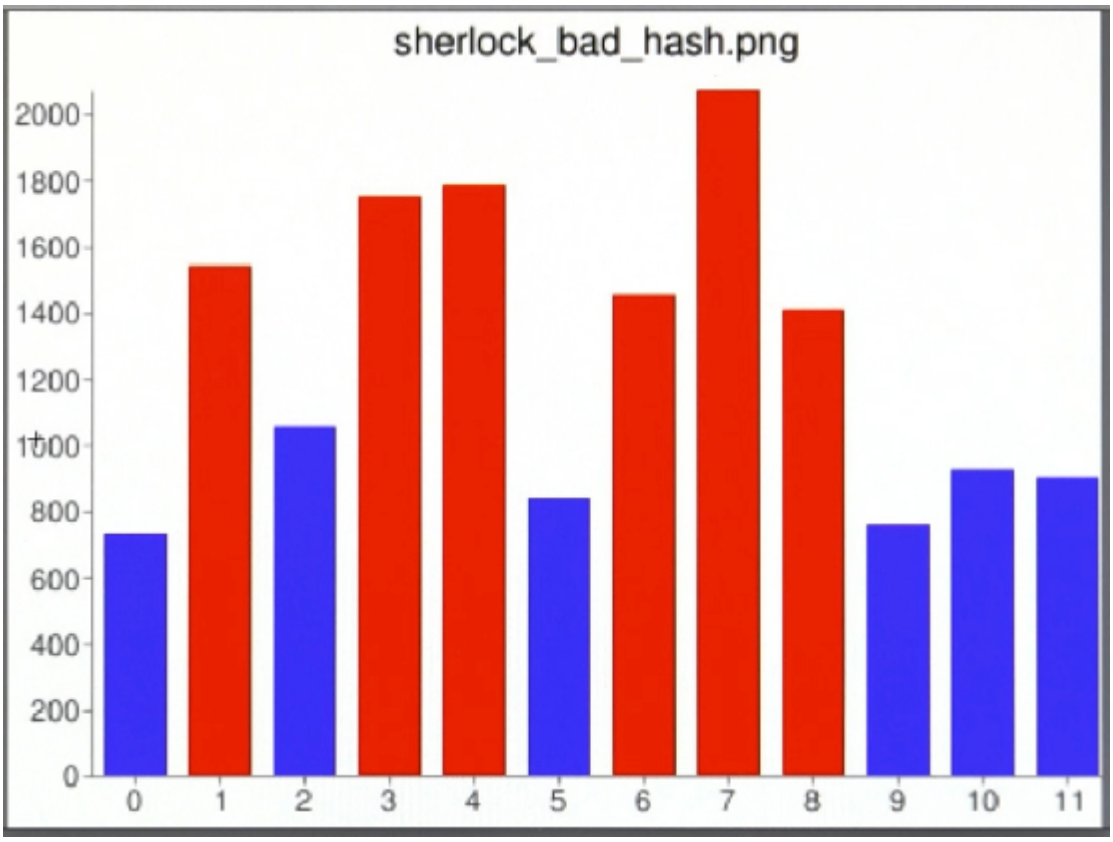
```
def test_hash_function(func, keys, size): #pass in function, keywords,
and size, which is number of buckets
 results = [0] * size # a list of the number of times each bucket is
used
 keys_used = [] # list of keywords already used
 for w in keys: # loop through the keys
     if w not in keys_used: # check if a key was used already
         hv = func(w, size) # if key not used, call hash function where
that key would hash to
         results[hv] += 1 # increase the value of the element at results
by 1
         keys_used.append(w) # add the key to the list of keywords so it
is not used again
 return results # a list where the values in that list are numbers,
giving the number of times a key hashes to that bucket
```

Here is an example of how this procedure works using the **bad_hash_string** function to test:

```
words = get_page('http://www.gutenberg.org/cache/epub/1661/
pg1661.txt').split() # get the words on the page and split them into
words, like in the crawler, store in variable 'words'
print len(words)
107533

counts = test_hash_function(bad_hash_string, words, 12)
[730, 1541, 1055, 1752, 1784, 839, 1452, 2074, 1409, 754, 924, 899]
```

The last function returned 12 elements because you had called for 12 in the function. However, the difference between the smallest and the largest element is quite large. If your hash function was good your results would be about the same size. Here is a graph of what these values look like. You can see there are 12 bars representing each bucket; the ones that are red are too full and the ones that are blue are not full enough. Ideally, you would want this to be a flat graph, distributing all the words evenly between the buckets.



A5-11: Better Hash Functions

```
def hash_string(keyword, buckets):
 h=0 # initialize a variable to keep track of where you are in the
circle
 for c in keyword: # go through the characters in the keyword
     h = (h + ord(c)) % buckets # for each character, add to the hash,
this is the best place to put the modulo
 return h # return the hash value

print hash_string('a', 12)
1

def hash_string(keyword, buckets):
 h=0
 for c in keyword:
     h = (h + ord(c)) % buckets
 return h

print hash_string('b', 12)
2
```

Does this code work on an empty string, or is it a bad hash?

```
def hash_string(keyword, buckets):
 h=0
 for c in keyword:
     h = (h + ord(c)) % buckets
 return h

print hash_string('', 12)
0
```

It makes sense that the result of an empty string is zero because you start with **h=0** and if there is nothing in the string you don't go through the loop at all, so **h** is still zero when you return.

What is the hash value of the string 'udacity'?

```
def hash_string(keyword, buckets):
 h=0
 for c in keyword:
     h = (h + ord(c)) % buckets
 return h

print hash_string('udacity', 12)
11
```

What happens when you increase the number of buckets?

```
def hash_string(keyword, buckets):
 h=0
 for c in keyword:
     h = (h + ord(c)) % buckets
 return h

print hash_string('udacity', 1000)
755
```

A5-12: Keywords and Buckets

There are two correct answers, c and e. A hash table is superior to a linear index because, as stated in c, you can double the number of keywords and double the number of buckets and the lookup time stays the same, whereas if you double the number of keywords in a linear index, the lookup time doubles.

However, a is incorrect because if you double the number of keywords without doubling the number of buckets, the lookup time will increase. Inversely, with answer b, if you double the number of buckets and keep the number of keywords the same, the lookup time will get faster. The same situation will occur when you halve the number of keywords and keep the same number of buckets -- the lookup time gets faster.

A5-13: Implementing Hash Tables

The answer is c. The main component you are looking for in a data structure that will allow you

to implement the hash table is a way of representing a bucket. What you want is a list, where each element of the list is a bucket, which is a list itself. Each element in the bucket/list is a key and a value. In your case the key is a word and the value is the list of urls.

Answer c is the data structure that corresponds most closely to what you are looking for. It is a list where each inner list corresponds to a bucket:

Also, inside the list, a word and a list of urls is one entry, which corresponds to what the entries were in your previous index:

Now, that you are looking to make a hash table, this data structure allows you to collect the entries in buckets. Each element in the outer list corresponds to one bucket.

A5-14: Empty Hash Table

Here is one way to define **`make_hashtable`**:

```
def make_hashtable(nbuckets):
 i = 0
 table = []
 while i < nbuckets:
     table.append([]) # everytime you go through the loop add one empty
bucket to the hashtable
     i = i + 1 # increase i to keep from looping forever
 return table

print make_hashtable(3) # for now use a small number of buckets
[[], [], []]
```

The code returns a list with three empty lists as its elements. While this code looks okay, it also looks like a lot more code than you need. A better way to write this would be to use a **`for`** loop, which looks like:

```
for e in <collection>:
```

In this case ***`collection`*** is usually a list or a string, some set of objects that you are looping through. In this case you want to loop through the numbers from zero to **`nbuckets`** minus 1. You can do this in Python using **`range`**.

`Range` takes two numbers as inputs, the start and the stop numbers and outputs a list of all the numbers from start, increasing by 1, until you get to stop minus 1. It does not include the value passed in as the second parameter in the list.

This is useful because often times when you loop through elements, you do not want to include the last element. For example, what if you evaluated:

```
range(0, 10) → [0, 1, 2, …, 9]
```

Now that you know about range, use the **`for`** loop instead of the **`while`** loop. This will not only

make the code shorter, but it will also save you from the danger of forgetting to increment the variable, a common mistake that results in the loop running forever.

```
table = []
for unused in range(0, nbuckets): # unused indicates a placeholder
variable that is not used
 table.append([])
return table
```

A5-15: The Hard Way

a. Because it is too easy and we like doing things the hard way:

This is actually complete opposite to the truth. You should keep your code as simple as possible (but not simpler). One exception to this is e.g. implementation of hash map, which is already available in Python. The reason behind this exception is that you are learning how it is done and how e.g. has maps work under the hood. This kind of knowledge will significantly help you e.g. to choose the right data structure for the given job.

b. Because each element in the output refers to the same empty list:

This is the correct answer. All elements of the created list refer to the same empty list. Therefore, when this empty list gets changed, it does not matter which of those references (i.e. elements of the resulting list) will be used to access the value - it will always be the same.

This behavior is not very intuitive and one needs to get accustomed to it. However, it is essential to understand the theory behind this quiz, because very ugly and hard-to-spot bugs might be introduced into your code, otherwise.

This theory is explained in Unit2, video #11 named Aliasing and following quizzes. See also the picture below.

c. Because * for lists means something different than it does for strings:

This is simply incorrect. Meaning of ***** for lists is exactly the same as for strings.

A5-16: Finding Buckets

```
#Define a procedure, hashtable_get_bucket,
#that takes two inputs - a hashtable, and
#a keyword, and outputs the bucket where the
#keyword could occur.

#hash_string(keyword,nbuckets) => index of bucket

def hashtable_get_bucket(htable,keyword):
    return htable[hash_string(keyword,len(htable))]

def hash_string(keyword,buckets):
    out = 0
```

```
    for s in keyword:
        out = (out + ord(s)) % buckets
    return out

def make_hashtable(nbuckets):
    table = []
    for unused in range(0,nbuckets):
        table.append([])
    return table
```

Function **hashtable_get_bucket** returns the bucket containing the given keyword, from the hash table, passed in as the first argument.

If you remember structure of a hash table, you will find out that it is composed of **n** buckets, one of which needs to be returned by the **hashtable_get_bucket** function. Index of the bucket, which would eventually contain the given keyword (in case the keyword will be present in the hash table), is computed and returned by already defined function **hash_string**.

The function **hash_string** will in turn take the keyword and number of buckets as its arguments. First argument (the keyword) is straightforward, since it was passed directly to **hashtable_get_bucket** function by its caller. The second argument (number of buckets) can be computed using **len** function on the hashmap (recall how hashmap is composed of **n** buckets).

A5-17: Adding Keywords

```
#Define a procedure,

#hashtable_add(htable,key,value)

#that adds the key to the hashtable
#(in the correct bucket), with the
#correct value.

def hashtable_add(htable,key,value):
    # find the correct bucket and append key-value pair to it
    hashtable_get_bucket(htable,key).append([key,value])

def hashtable_get_bucket(htable,keyword):
    return htable[hash_string(keyword,len(htable))]

def hash_string(keyword,buckets):
    out = 0
    for s in keyword:
    out = (out + ord(s)) % buckets
    return out

def make_hashtable(nbuckets):
    table = []
    for unused in range(0,nbuckets):
```

```
        table.append([])
        return table
```

As it was pointed out at the beginning of this section, in order to add a value, you need to know which bucket to add it into. Function **`hashtable_get_bucket`**, defined in previous quiz, does exactly that.

The rest of the job, which needs to be done is to **`append`** new key-value pair into the bucket returned by **`hashtable_get_bucket`** function.

A5-18: Lookup

```
#Define a procedure,

#hashtable_lookup(htable,key)

#that takes two inputs, a hashtable
#and a key (string),
#and outputs the value associated
#with that key.

def hashtable_lookup(htable,key):
    # find correct bucket
    bucket = hashtable_get_bucket(htable,key)

    # go sequentially through all entries in the bucket
    for entry in bucket:
        # compare their keys to the key being searched
        if entry[0] == key:
            # if they match, return the value associated with the key
            return entry[1]

    # in case control reached this point, all of entries in the bucket
    # were checked and none of them matched given search criteria
    return None

def hashtable_add(htable,key,value):
    bucket = hashtable_get_bucket(htable,key)
    bucket.append([key,value])

def hashtable_get_bucket(htable,keyword):
    return htable[hash_string(keyword,len(htable))]

def hash_string(keyword,buckets):
    out = 0
    for s in keyword:
        out = (out + ord(s)) % buckets
    return out
```

```
def make_hashtable(nbuckets):
    table = []
    for unused in range(0,nbuckets):
        table.append([])
    return table
```

As mentioned before, the first thing you need to do is to find the correct bucket in the given hashmap. Function **hashtable_get_bucket**, defined in earlier quiz, does exactly that.

The next thing to do is to find the correct entry in the bucket (or prove that it is not present). Since entries in the bucket do not adhere to any special rule (e.g. they are not sorted according to any key, etc.), the only way how to do that is to go sequentially through all of entries in the bucket and compare their keys to the searched one.

The last step is to return correct value, associated with the found entry in the bucket (or special value **None** in case matching entry is not present).

A5-19: Update

```
#Define a procedure,

#hashtable_update(htable,key,value)

#that updates the value associated
#with key. If key is already in the
#table, change the value to the new
#value. Otherwise, add a new entry
#for the key and value.

#Hint: Use hashtable_lookup as a
#starting point.

def hashtable_update(htable,key,value):
    # next 3 lines (i.e. searching for key-value pair)
    # are exactly the same as in hashtable_lookup
    # function
    bucket = hashtable_get_bucket(htable,key)
    for entry in bucket:
        if entry[0] == key:
            entry[1] = value
            return

    bucket.append([key, value])
    # or reuse already implemented function,
    # which has the same effect as the line
    # above:
    #hashtable_add(htable,key,value)

def hashtable_lookup(htable,key):
```

```
    bucket = hashtable_get_bucket(htable,key)
    for entry in bucket:
        if entry[0] == key:
            return entry[1]
    return None

def hashtable_add(htable,key,value):
    bucket = hashtable_get_bucket(htable,key)
    bucket.append([key,value])

def hashtable_get_bucket(htable,keyword):
    return htable[hash_string(keyword,len(htable))]

def hash_string(keyword,buckets):
    out = 0
    for s in keyword:
        out = (out + ord(s)) % buckets
    return out

def make_hashtable(nbuckets):
    table = []
    for unused in range(0,nbuckets):
        table.append([])
        return table
```

In case you have been following (and understood) previous quizzes on hash table operations, this one should feel very familiar. Implementation of **`hashtable_update`** function is very similar to **`hashtable_lookup`** function. In fact, the only difference is that while the latter function is just looking up (reading) the value, former one is updating (writing) it.

When you look closer at the hash map implementation above, you will find out that the part of these two functions, responsible for looking-up key-value pair is exactly the same. Such ***code duplication*** might be considered a bad practice (depending on its scale) and such code should be ***refactored***, because any change (be it because of functionality enhancement, bug-fix or anything else) in one "instance" of the duplicated code needs to be done in all other "instances". However, such "propagation" of the change is tedious at very least, and usually (in real-world code), it is also very error-prone process.

A5-20: Population

Here's one way to solve the quiz. Dave's home town is Charlottesville so that is added to the dictionary too.

```
population = {}
population['Shanghai'] = 17.8 #the population of Shanghai is 17.8
million
population['Istanbul'] = 13.3
population['Karachi'] = 13.0
population['Mumbai'] = 12.5
```

```
population['Charlottesville'] =  0.043 # only 43000 in Charlottesville

print population['Shanghai']
17.8

print population['Charlottesville']
0.043
```

A5-21: Modifying the Search Engine

The answers are b, d and e.

In order to change b, **crawl_web**, to use a dictionary, start with the code:

```
def crawl_web(seed):
    tocrawl = [seed]
    crawled = []
    index = []
    while tocrawl:
        page = tocrawl.pop()
        if page not in crawled:
            content = get_page(page)
            add_page_to_index(index, page, content)
            union(tocrawl, get_all_links(content))
            crawled.append(page)
    return index
```

In this code you initialized **index** with an empty list and just used it to pass into **add_page_to_index** . To change **index** to use a dictionary, just change the square brackets to be curly brackets:

```
index = {}
```

Now, instead of starting with an empty list, you are starting with an empty dictionary. This is the only change you need to make to **index**.

The change required to change d, **crawl_web** is a little more complicated.

```
def crawl_web(seed):
    tocrawl = [seed]
    crawled = []
    index = {}
    while tocrawl:
        page = tocrawl.pop()
        if page not in crawled:
            content = get_page(page)
            add_page_to_index(index, page, content)
            union(tocrawl, get_all_links(content))
            crawled.append(page)
```

```
    return index
```

In this code, what happens to each page is that you crawl **add_to_index**, passing in the index, which is now a dictionary. Here is the code for **add_page_to_index**:

```
def add_page_to_index(index, url, content):
    words = content.split()
    for word in words:
        add_to_index(index, word, url)
```

This code takes the **index**, goes through the **words**, adds each word to the index. This can be done whether **index** is a list or a dictionary. You do not need to change **add_page_to_index**, but rather **add_to_index**. Here is the code for **add_to_index**:

```
def add_to_index(index, keyword, url):
    for entry in index:
        if entry[0] == keyword:
            entry[1].append(url)
            return
    # not found, add new keyword to index
    index.append([keyword, [url]])
```

Before, you had **add_to_index**, which took in an **index**, a **keyword** and a **url** – it will still take the same parameters. However, when **index** was a list, your code was written such that you went through all of the entries in the **index**, checked for each one to see if it matched the **keyword** you were looking for. If it did, then you add the **url**, but if you got to the end without finding it then you **append** a new entry, which is the **keyword** and the list of urls, containing just the first **url**.

Here is how you can change this code to work with the hash table index. Remember that with a hash table you do not need to loop through anything. With the dictionary, the built in **in** operation serves this purpose. So, instead of looping, now you can check right away if a **keyword** is in the **index**:

```
def add_to_index(index, keyword, url):
   for entry in index:
       if entry[0] == keyword:
           entry[1].append(url)
           return
   # not found, add new keyword to index
   index.append([keyword, [url]])
```

Before, you had **add_to_index**, which took in an **index**, a **keyword** and a **url** – it will still take the same parameters. However, when **index** was a list, your code was written such that you went through all of the entries in the **index**, checked for each one to see if it matched the **keyword** you were looking for. If it did, then you add the **url**, but if you got to the end without finding it then you **append** a new entry, which is the **keyword** and the list of urls, containing just

the first **url**.

Here is how you can change this code to work with the hash table index. Remember that with a hash table you do not need to loop through anything. With the dictionary, the built in **in** operation serves this purpose. So, instead of looping, now you can check right away if a **keyword** is in the **index**:

```
def add_to_index(index, keyword, url):
   if keyword in index:
       index[keyword].append(url) # this will look up in the dictionary
the entry that corresponds to index, which is going to be the list of
urls
    else:
  # not found, add new keyword to index
    index[keyword] = [url]
```

This code is a lot simpler and is going to run a lot faster because you don't have to loop through. Because of the hash table you can right away look up whether or not the keyword is in the index. If it is, you can find out what the value is by using the dictionary lookup and then append the new url to the list of urls associated with that keyword. If it is not found, you can create a new entry, **index[keyword] = [url]**, using the dictionary syntax that contains just that url.

A5-22: Changing Lookup

Here is one way to define lookup:

```
def lookup(index, keyword):
  if keyword in index: # instead of loop, check to see if keyword is in
the index
        return index[keyword] # if it is, use dictionary lookup
  else:
        return None # if the keyword is not in the index
```

More on Range

Explanation of range(<*start*>, <*end*>, <*step*>)

The syntax used is **range(<start>, <end>, <step>)** where **<start>** is the initial value, **<end>** is one more than the final value and **<step>** is the size of the step the value will increase (or decrease) by from the initial value to the final value. For example, **range(8, 0, -1)** starts at 8, which it decreases by 1 until it gets to 1 (since 0+1 =1 and it ends before it reaches 0.) This gives the list *[8, 7, 6, 5, 4, 3, 2, 1]*. Here are some more examples:

```
print range(5, 0, -1) #=> [5, 4, 3, 2, 1]    decreasing by 1
print range(5, 0, -2) #=> [5, 3, 1]          decreasing by 2
print range(5, 10, 1) #=> [5, 6, 7, 8, 9]    increasing by 1
```

Note that you can omit the **<*step*>** size, in which case the default value of 1 is used.

print range(5, 10) #=> *[5, 6, 7, 8, 9]* default increasing by 1

Similarly, you can omit the **<start>** value as well as the **<step>** (but not on its own) and in this case the default value of 0 is used for start.

print range(10) #=> *[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]* default start 0, default increase 1

Here are a few examples for you to try in the interpreter to get a better feeling for how range works.

1. What do the following print? See if you can work it out before trying it.

a. **print range(2, 0, -1)**
b. **print range(0, 20, 2)**
c. **print range(3, 7, 1)**
d. **print range(0, 10, 1)**
e. **print range(0, 10)**
f. **print range(3, 5)**
g. **print range(3, 4)**
h. **print range(3, 3)**
i. **print range(5, 3)**
j. **print range(2, 0, 1)**
k. **print range(0, 10, -1)**

2. How could you write the following with fewer inputs?

a. **range(2,7,1)**
b. **range(0, 10, 1)**

Answers to 2:
a. **range(2, 7)**
b. **range(10)**