Computational Approaches for Life Scientists

Faculty of Biology, Technion
Spring 2015

Lecturer: Amir Rubinstein
Co-designer and advisor: Benny Chor

Biological Sequences (1) – Hashing

Outline

- The *Common Substring Problem*
  - Naïve solutions
  - Hash based solution

- Hashing
  - hash functions and hash tables
  - Python's sets
  - Implementing our own "home-made" hash tables

- The *Most Frequent k-mer Problem*
  - Python's dictionaries

The Common Substring Problem

- Suppose we have two genomes, and we want to check if they both contain an identical region of a specified length *k*.
- A large such *k* is likely to reflect evolutionary proximity.

Mycobacterium tuberculosis

TTGACCGATGACCCCGGTTCAGGCTTCACCACAGTGTGGAACGCGGTCGTCTCCGAACTTAACGGCGACCC
TAAGGTTGACGACGGACCCAGCAGTGATGCTAATCTCAGCGCTCCGCTGACCCCTCAGCAAAGGGCTTGGC
TCAATCTCGTCCAGCCATTGACCATCGTCGAGGGGTTTGCTCTGTTATCCGTGCCGAGCAGCTTTGTC ...

Salmonella enterica

AGAGATTACGTCTGGTTGCAAGAGATCATAACAGGGGAAATTGATTGAAAATAAATATATCGCCAGCAGCACAT
GAACAAGTTTCGGAATGTGATCAATTTAAAAATTTATTGACTTAGGCGGGCAGATACTTTAACCAATATAGGAAT
ACAAGACAGACAAATAAAAATGACAGAGTACACAACATCCATGAACCGCATCAGCACCACC ...

```
>>> len(Mycobacterium_tuberculosis)
4411532
>>> len(Salmonella_enterica)
4809037
```

The Common Substring Problem

- Formalizing the problem:

  Input: 2 sequences s_1 and s_2

  an integer k

  Output: a common substring of s_1 and s_2 of length k (if exists).

- This is a recommended first step in solving any computational problem.

A Naïve Solution

```
def common_substring_naive(s1,s2,k):
    """ find common substring of s1 and s2 of length k """
    for i in range(len(s1)-k+1):
        for j in range(len(s2)-k+1):
            if s1[i:i+k] == s2[j:j+k]:
                return s1[i:i+k] #return first match
    return None
```

```
>>> common_substring_naive(Mycobacterium_tuberculosis,
                           Salmonella_enterica, 10)

'TTGACCGATG'
```

- How many character comparisons in the worst case, as a function of $|s_1|=n_1$, $|s_2|=n_2$, and k?

This is not exponential, yet may not be efficient enough for long sequences and larger k.

```
>>> common_substring_naive(Mycobacterium_tuberculosis,
                           Salmonella_enterica, 100)

... #takes a LONG time!
```

A Naïve Solution – Time Estimation

```
>>> common_substring_naive(Mycobacterium_tuberculosis,
                           Salmonella_enterica, 10)

'TTGACCGATG'


>>> common_substring_naive(Mycobacterium_tuberculosis,
                           Salmonella_enterica, 100)

... #takes a LONG time!
```

Exercise 1

Estimate the total running time for the above execution for k=100, by adding diagnostic printouts to the function.

Exercise 2

One could expect the previous execution with k=100 to run 10 times slower than with k=10.

Can you explain why this is not the case, and in fact k=10 terminates much faster in this case?

Another Solution

- This solution uses Python's 'in' operator:

```
def common_substring_naive2(s1,s2,k):
    """ find common substring of s1 and s2 of length k """
    for i in range(len(s1)-k+1):
        if s1[i:i+k] in s2:
            return s1[i:i+k] #return first match
    return None
```

- Did we improve? By how much? Are we satisfied?

- The 'in' operator implementation in Python involves rather sophisticated algorithms (based on a mix between Boyer-Moore and Horspool algorithms, which are beyond our scope).

- This solution is therefore naïve only from the programmer's perspective…

Yet Another (3rd) Solution?

```
def common_substring_hash(s1, s2, k):
    """ find a common length k substring of s1 and s2
    using Python built-in sets """
    table = set()
    for i in range(len(s1)-k+1):
        table.add(s1[i:i+k])

    for i in range(len(s2)-k+1):
        if s2[i:i+k] in table:
            return s2[i:i+k]

    return None
```

Note the use of Python's sets

- This solution uses memory and stores k-mers.
- The name of the function (hash??) will be clear very soon.

```
>>> common_substring_hash(Mycobacterium_tuberculosis,
                          Salmonella_enterica, 100)

>>> #None returned, almost immediately!
```

- This dramatic speedup must be investigated!

But before, let's conduct a more in-depth inspection of actual running times.

Actual Running Time Measurements

- We will run and compare our solutions on random strings.

```
import random
def gen_str(size, alphabet):
    ''' Generate a random string of length size over alphabet '''
    s = ""
    for i in range(size):
        s += random.choice(alphabet)
    return s
```

```
import time
def test():
    str_len = int(input("Type length of the random genomes: "))
    k = int(input("Type k: "))

    s1 = gen_str(str_len, "ATCG") #random string
    s2 = gen_str(str_len, "ATCG") #another one

    for f in [ common_substring_naive,  \
               common_substring_naive2, \
               common_substring_hash   ]:

       t0 = time.clock()
       for i in range(10): #repeat 10 times, for statistical significance
            res = f(s1, s2, k)
       t1 = time.clock()
       if res == None:
            res = "(not found)" #for a clear printout in case of None
    print((t1-t0)/10, "sec", f.__name__, res)
```

```
>>> test()
Type length of the random genomes: 1000
Type k: 20
0.31592526238903307   sec common_substring_naive (not found)
0.002703917061858618  sec common_substring_naive2 (not found)
0.0003604806674994521 sec common_substring_hash (not found)

>>> test()
Type length of the random genomes: 2000
Type k: 20
1.3124293173238057    sec common_substring_naive (not found)
0.01057980749109504   sec common_substring_naive2 (not found)
0.0007444535898248006 sec common_substring_hash (not found)

>>> test()
Type length of the random genomes: 4000
Type k: 20
5.351912174155499     sec common_substring_naive (not found)
0.041724375053100005  sec common_substring_naive2 (not found)
0.0015739525633208019 sec common_substring_hash (not found)
```

- Note: all executions printed "not found" (why is this relevant?).
- What can we learn about the growth rate of the solutions' time complexity?

Hash Functions and Hash Tables

- Python's set (and also dict) is implemented "behind the scenes" using hash tables. This is the source for the major speedup we are witnessing.

- Hash tables and hash functions are fundamental notions in CS, and we will learn about them now.

- By the end of this lecture you will:

  - ✓ have a good understanding of how hash tables enable efficient solutions for problems, such as the *common substring problem*

  - ✓ know how to use sets and dictionaries efficiently for various problems

  - ✓ be able to use your own "home-made" hash tables, which you can tailor for specific tasks.

Hash Tables

- Suppose elements belong to a large set, called "**universe**", denoted ***U***.

  For example, all possible genomic *k*-mers for some *k*.

- Suppose we need to store *n* elements from ***U***, and ***n*** << |***U***|.

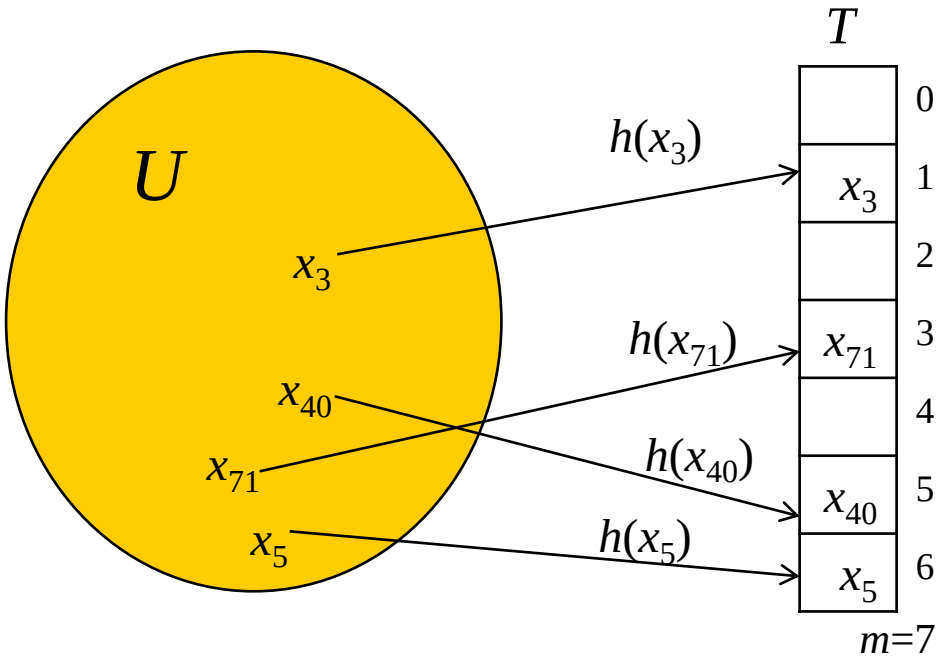
  For example, all *k*-mers actually contained in some genomic string.

- We will keep the elements in a table ***T*** called **hash table**, whose size is $m \approx n$.
- $|T| = m \approx n \quad (<< |\boldsymbol{U}|)$

- We will use a **hash function**

  $h: U \rightarrow \{0, 1, \ldots, m\text{-}1\}$

  $x \in U$ will be saved at $T[\, h(x)\,]$



Hash Tables – Simple Example for Integers

- U = {all possible Israeli ID numbers }
- n = number of students who attended class today
- $|T| = m = 10$
- $h: U \rightarrow \{0, 1, \ldots, m\text{-}1\}$

  $h(\text{id}) = \text{id} \% m$

Collisions

- Collision: $h(x) = h(y)$ for $x \neq y$.

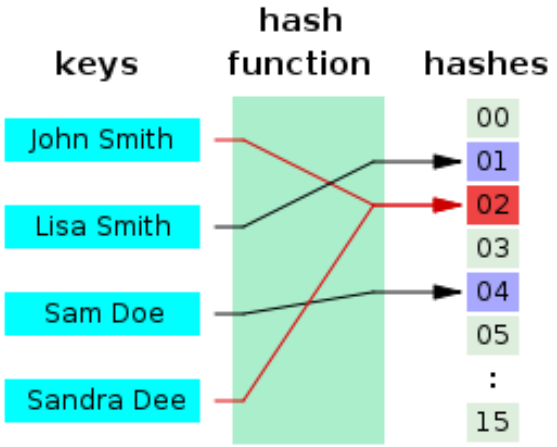


Image taken from Wikipedia

- How can we lower the probability for collisions?

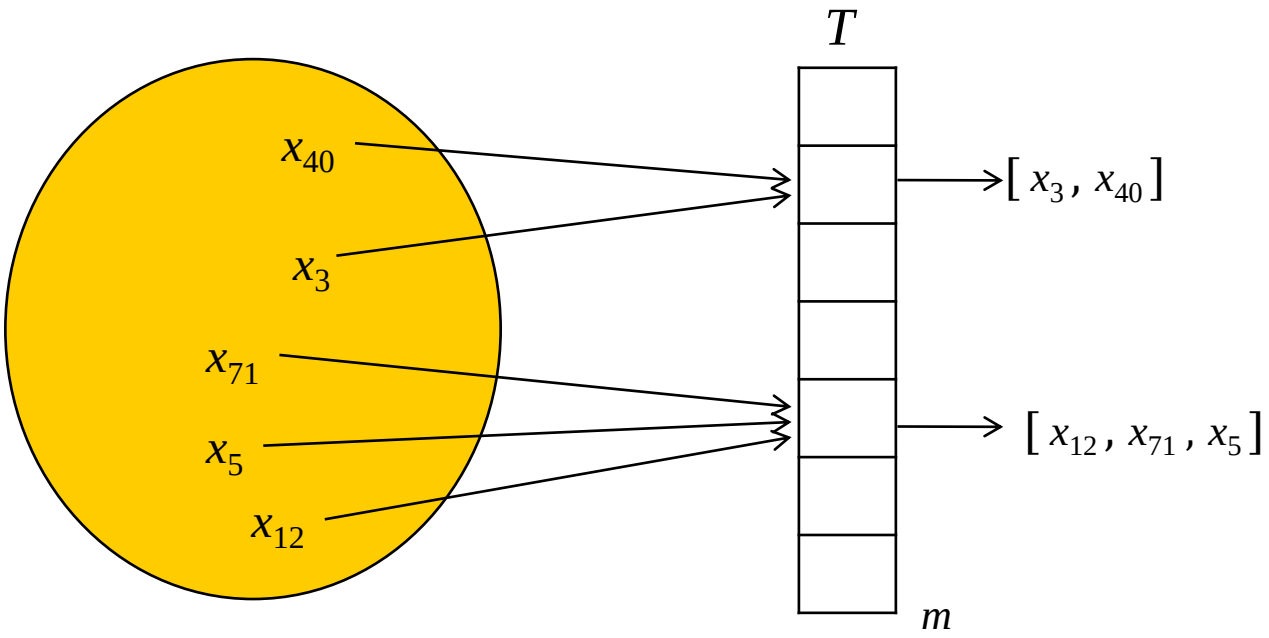
- Can we totally avoid collisions, if $m \approx n << |\boldsymbol{U}|$?

Pigeonhole principle:

if n+1 pigeons enter n holes, at least 1 hole will contain at least 2 pigeons.

Possible Solutions for Collisions

- There are 2 types of solutions:
  1. Chaining: keep a list ("chain") at every entry of the hash table:



Insert(*key*) – append *key* to the list $T[h(key)]$

Delete(*key*) – remove *key* from the list $T[h(key)]$

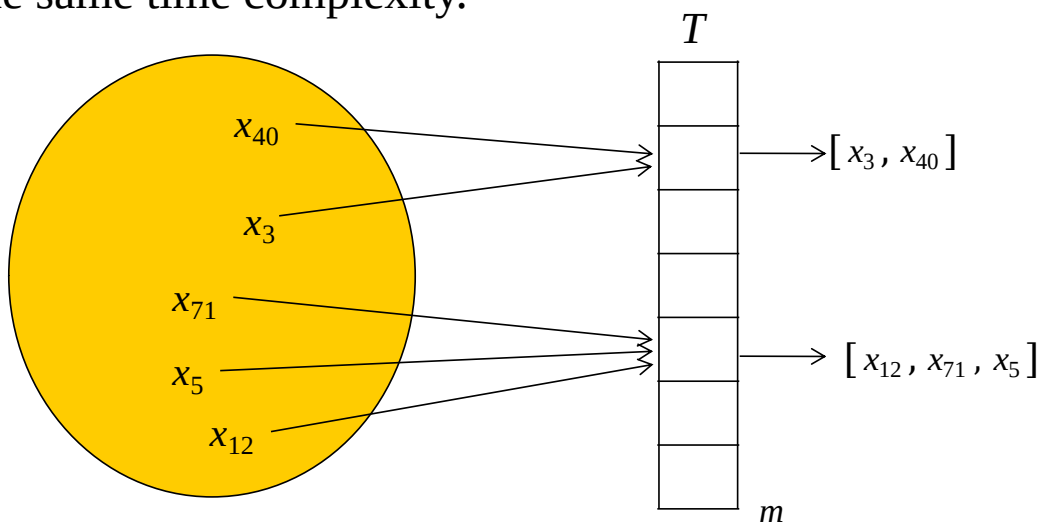
Search(*key*) – check if *key* in $T[h(key)]$

  2. Open addressing: a colliding element will be redirected to another empty cell, by some predefined rules.

Python's set (and dict) uses this approach, which we will briefly discuss in class.

Complexity

- Chaining is easier to analyze and understand than open addressing, but essentially both support the same time complexity.



- The average number of elements per list is termed the load factor, denoted α.

  $\alpha = n/m$.

- How many elements on average do we examine, when searching an element in the table?

  And when $m = n$? $\frac{1}{2}n$? $\frac{1}{4}n$?

- What about the **worst-case time complexity**?

Good Hash Functions for Strings

```
def badhash1(st):
    return ord(st[0]) + ord(st[1])
```

```
def badhash2(st):
    return sum([ord(c) for c in st])
```

- Python's ord converts a character to its Unicode value (same as ASCII value for standard characters)
- Why are these bad hash functions for strings?

- What are the requirements from a good hash function?
  - distributes well elements from U in the hash table T
  - easy to compute in short time

- Python has a built-in hash function. It will be very good for our needs.

- But just to get an impression, here is a possible implementation:

```
def hash4strings(s):
    sum = 0
    for i in range(len(s)):
        sum = (128*sum + ord(s[i])) % (2**120+451)
    return sum**2 % (2**120+451)
```

($2^{120}+451$ is a prime number)

Implementing Our Own "Home-made" Hash Tables

- Having our own hash table implementation, we can tailor it to specific needs.

  For example: allow repeating elements in the table, use another hash function, etc.

- For simplicity, we will use chaining to solve collisions.

```
def create_hash_table(m):
    """ initial hash table, m empty entries """
    return [[] for i in range(m)] #list comprehension
```

```
def find(htable, key):
    """ return index if key is in htable, else None """
    m = len(htable)
    index = hash(key) % m
    if  key in htable[index]:
        return index
    return None
```

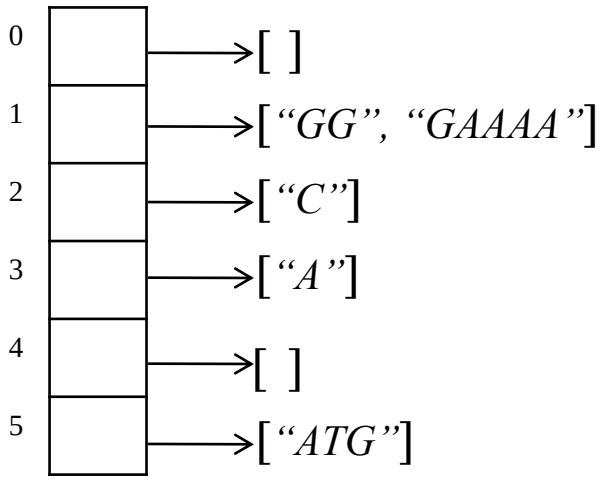
```
def insert(htable, key):
    """ insert key into htable """
    if find(htable, key) == None:
        m = len(htable)
        index = hash(key) % m
        htable[index].append(key) #add at the end of list
```

If you'd like to avoid repetitions.

Implementing Our Own Hash Tables

- Example:

```
>>> ht = create_hash_table(6)
>>> insert(ht, "A")
>>> insert(ht, "ATG")
>>> insert(ht, "C")
>>> insert(ht, "GG")
>>> insert(ht, "GAAAA")
>>> ht
[[], ['GG', 'GAAAA'], ['C'], ['A'], [], ['ATG']]
```



Results may differ from one IDLE instance to another, since Python's hash may produce different results!

Python vs. "Home-made" Hash Tables

```
def common_substring_hash(s1, s2, k):
    """ find common substring of s1 and s2 of
    length k, using Python sets """

    table = set()

    for i in range(len(s1)-k+1):
        table.add(s1[i:i+k])

    for i in range(len(s2)-k+1):
        if s2[i:i+k] in table:
            return s2[i:i+k]

    return None
```

Hmm…
a table of what size?
Python maintains the size dynamically

```
def common_substring_hash2(s1, s2, k):
    """ find common substring of s1 and s2
    of length k. uses hash tables """

    m = len(s1)-k+1
    table = create_hash_table(m)

    for i in range(len(s1)-k+1):
        insert(table, s1[i:i+k])

    for i in range(len(s2)-k+1):
        if find(table, s2[i:i+k]) != None:
            return s2[i:i+k]

    return None
```

So α=1

- Not surprisingly, Python's built-in hash tables perform better than our version…

```
Type length of the random genomes: 1000
Type k: 10
0.0012188 common_substring_hash (not found)
0.0053986 common_substring_hash2 (not found)
```

A Word About Memory Resources

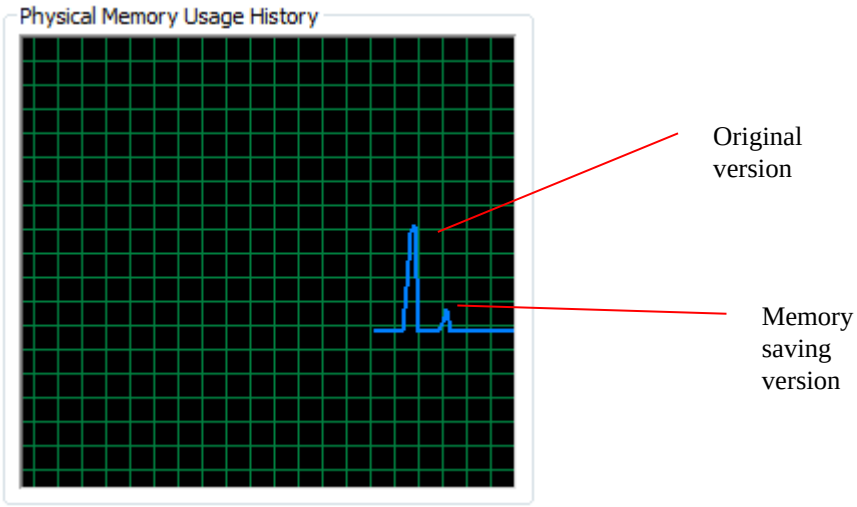
- How many characters are stored in total, for all *k*-mers of s_1 ?

- When s_2 is shorter, we could easily save memory and further improve our solution.

```
def common_substring_hash(s1, s2, k):
    """ find a common length k substring of s1 and s2
    using Python built-in sets """
    if len(s1) > len(s2):
        s1,s2 = s2,s1 #this works in Python!

    table = set()

    for i in range(len(s1)-k+1):
        table.add(s1[i:i+k])

    for i in range(len(s2)-k+1):
        if s2[i:i+k] in table:
            return s2[i:i+k]

    return None
```

Observing Memory Load

```
>>> common_substring_hash(Mycobacterium_tuberculosis, Salmonella_enterica[0:10**6], 150)
```



- In extreme cases, this could make the difference between a successful execution and a memory flow failure!

- This improvement has positive effect of on the time as well.

  In the above case, the speedup was about 20%!

Python's dict

- So far we used Python sets.
- We mentioned that Python's dict is also implemented "behind the scenes" as a hash table.
- We will now show an example for using Python dict in this context, for solving another, very common problem in biology:

The most frequent *k*-mer problem.

The Most Frequent k-mer Problem

Input: a sequence *s*

an integer *k*

Output: a *k*-mer contained in *s* the maximal number of times.

Possible use: Find most common triplet in a genome.

We'll use this genome:

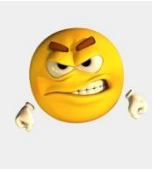
```
>>> len(Mycobacterium_leprae)
3268071
```

- Issues that require clarifications:
  - What to do when there is more than one maximum? (Provide one arbitrarily)
  - Do we allow overlaps? (yes, for now)

The Most Frequent k-mer: Naïve Solutions

- Naïve 1
  1. For every possible *k*-mer:
     - 1.1 Go over all possible *k*-mers and count matches (using string slicing).
     - 1.2 Update maximum if needed
  2. Return maximum

  Requires ~38 days on my PC for the *Mycobacterium Leprae* genome, *k*=3.



- Naïve 2

  The same, but use Python string's count function instead of line 1.1

  Turns out ~100 times faster on my PC. That's ~9 hours.

  Also, does not allow overlaps:

```
>>> "AAA".count("AA")
1
```

- I hope by now you are convinced:

  naïve approaches may be (but are not always) too inefficient.

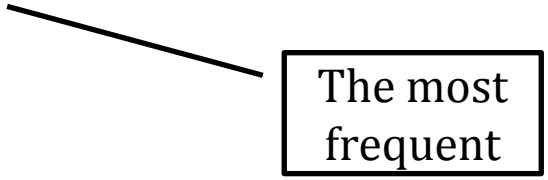
The Most Frequent k-mer Problem: Solution

- We will keep, for each *k*-mer, a counter with the number of times it appears in *s*.
- So we will be storing *k*-mers mapped to counters in Python's dict.

For example:

s = "CCCTTGCTT" , *k* = 3

{'GCT': 1, 'TTG': 1, 'CTT': 2, 'TGC': 1, 'CCT': 1, 'CCC': 1}



The Most Frequent k-mer: Solution

```
def frequent_repeating_substring(st, k):
    counters = dict()
    most_freq = st[:k]
    max_freq = 1

    for i in range(len(st)-k+1):
        if st[i:i+k] not in counters:
            counters [st[i:i+k]] = 1 #first time
        else:
            counters[st[i:i+k]] += 1 #found one more
            #update maximum
            if max_freq < counters[st[i:i+k]]:
                max_freq = counters[st[i:i+k]]
                most_freq = st[i:i+k]

    return most_freq, max_freq #return a tuple of 2 values
```

Piece of cake – just a few seconds!

```
>>> frequent_repeating_substring(Mycobacterium_leprae, 3)
('CGG', 89360)
>>> Mycobacterium_leprae.count('CGG') #sanity check.
89360
```

Reflection

Key notions from this lecture:

- Hash functions as a mapping from a large set into a small set (of integers).

- Hash tables are used for storing elements with efficient lookup.

  Key issues:

  - Collisions are inherent
  - A good hash function spreads the elements uniformly (minimizes collisions)
  - The load factor needs to be some small constant (i.e. the table is large enough)

- Naïve solutions vs. more sophisticated ones (sometimes naïve is good enough)

Reflection (2)

Data structures

- Hash tables organize data in the computer's memory, for efficient lookup.
- This is an example for a data structure.

- A list in Python is another (simple) data structure.
- Some regard even simpler types, such as integers, as (very simple) data structures.
- Other famous examples of data structures include: Linked list, Stack (מחסנית), Queue (תור), Graph, Tree (AVL, Red-black, Trie), Heap.

- Some common data structures are built-in in some languages. Others have to be implemented by the user. The operations we try to optimize should guide our choice for an appropriate data structure.

Reflection (3)

The biological problems we solved today:

- Common substring of 2 strings
- Most frequent *k*-mer in a string

Other related problems:

- Common substring of >2 strings
- Repeating *k*-mer in a string (with / without overlaps)
- Longest common substring / repeating k-mer
- …

Reflection (4)

The Birthday Paradox

An interesting phenomena, known as the "birthday paradox, is closely related to the collision probability in hash tables.

Suppose you are in a (large) room, containing 23 people. What is the probability that two of them (including yourself) have the same birthday (day and month. It is impolite to ask about the year, and in some places it is even illegal, as in interviews for job application)?

It turns out that the probability for such collision is 0.5073, i.e. strictly greater than 0.5. This stands against "common intuition", which assigns a much lower probability to this event. If there are 29 other people in the room, the birthday collision probability goes to 0.7063.

How is this cool "paradox" related to hash tables? The "objects" are the people in the room. The hash function of a person is his/her birthday. There are 365 possible days, which is the size of our imagined "hash table". So it turns out, that when hashing 23 objects into a table with 365 entries, the probability of at least one collision is 0.5073.

Note that we assume that the people in the room got there independently of their birth day. If you handpicked 30 people, born in different days in April, there will obviously be no collisions.

Exercise – Longest Common Substring

A. Write a function longest_common_substring(s1,s2),

that returns a longest common substring of s1 and s2.

Use common_substring_hash that we saw in class, on increasing values of *k*, i.e. *k*=1, 2, 3,… .

B. What is the longest common substring of these two sequences?

s1 = "GATTAGCCGTAGATTGA"

s2 = "AGGAAGGATGCCGTGAAA"

C. How much time does it take to find the longest common substring of the genome of *Mycobacterium tuberculosis* and *Mycobacterium leprae*?

These genomes are provided to you in the files holding these names (remember to remove "\n" characters).

Note: the genomes are very large, avoid printing them.

D. Use a binary search approach to speed up the process from the previous section.

First, increase *k* 2-fold each time: *k*=1,2,4,8,… until there is no common substring.

Then use binary search on the last interval.