The Dictionary Problem
Hash Functions and Hash Tables

Instructors: Benny Chor, Amir Rubinstein
Teaching Assistants: Yael Baran, Michal Kleinbort

Founding Teaching Assistant (and Python Guru): Rani Hod

School of Computer Science
Tel-Aviv University
Spring Semester, 2014
http://tau-cs1001-py.wikidot.com
Lecture 16: Plan

- The dictionary problem (find, insert, delete).
- Python hash and `<class 'dict'>`.
- Hash functions and hash tables.
- Resolving collisions: Chaining and open addressing.

- Using hashing to solve large string problems.
Hash

Definition (from the Merriam–Webster dictionary):

hash
transitive verb
1 a: to chop (as meat and potatoes) into small pieces
   b: confuse, muddle
2: to talk about: review – often used with over or out

Synonyms: dice, chop, mince
Antonyms: arrange, array, dispose, draw up, marshal (also marshall), order, organize, range, regulate, straighten (up), tidy

In computer science, hashing has multiple meaning, often unrelated. For example, universal hashing, perfect hashing, cryptographic hashing, and geometric hashing, have very different meanings. Common to all of them is a mapping from a large space into a smaller one.

Today, we will study hashing in the context of the dictionary problem.
Hash Functions, Hash Tables, and Search

(figure from http://searchengineland.com/search-market-share-google-up-bing-flat-yahoo-hits-new-low-124519)

And, while at that

(figure from http://www.designbaskets.com/services/seo-sem/) (May 2012 data.)
Search (reminder from lectures 5 and 15)

Search has always been a central computational task. The emergence and the popularization of the world wide web has literally created a universe of data, and with it the need to pinpoint information in this universe.

Various search engines have emerged, to cope with this big data challenge. They constantly collect data on the web, organize it, and store it in sophisticated data structures that support efficient (very fast) access, resilience to failures, frequent updates, including deletions, etc. etc.

In lecture 5, we have dealt with much simpler data structure that support search:

- unordered list
- ordered list
Sequential vs. Binary Search

For unordered lists of length $n$, in the worst case, a search operation compares the key to all list items, namely $n$ comparisons.

On the other hand, if the $n$ elements list is sorted, search can be performed much faster, in time $O(\log n)$.

One disadvantage of sorted lists is that they are static. Once a list is sorted, if we wish to insert a new item, or to delete an old one, we essentially have to reorganize the whole list – requiring $O(n)$ operations.

Linked lists also exhibit $O(n)$ worst time performance for some insert, delete, and even search operations.
Dynamic Data Structure: Dictionary

A dictionary is a data structure supporting efficient \texttt{insert}, \texttt{delete}, and \texttt{search} operations.

We will introduce \texttt{hash functions}, and use them to build \texttt{hash tables}. These hash tables will be used here to implement the abstract data type \texttt{dictionary}.

The abstract data type \texttt{dictionary} should \texttt{not} be confused with Python's \texttt{<class 'dict'>}.
Dynamic Data Structure: Dictionary

A dictionary is a data structure supporting efficient insert, delete, and search operations.

We will introduce hash functions, and use them to build hash tables. These hash tables will be used here to implement the abstract data type dictionary.

In our setting, there is a dynamic (changing with time) collection of up to $n$ items. Each item is an object that is identified by a key. For example, items may be instances of our Student class, the keys are students’ names, and the returned values are the students’ ID numbers and grades in the course.

We assume that keys are unique (different items have different keys).
Other Dynamic Data Structure

There are data structures, known as balanced search trees, which support these three operations in worst case time $O(\log n)$. They are fairly involved, and studied extensively in the data structures course.

Figure from MIT algorithms course, 2008. Shows item insertion in an AVL tree.
Dynamic Data Structure: Dictionary

**Question:** Is it possible to implement these three operations, insert, delete, and search, in time \(O(1)\) (a constant, regardless of \(n\))? 

As we will shortly see, this goal can be achieved on average using the so called hash functions and a data structure known as a hash table.

![Hash Table Diagram](figure from Wikipedia)

We note that Python’s dictionary (storing key:value pairs) is indeed implemented using a hash table.
Dictionary Setting

- There is a very large universe of elements with unique keys, $\mathcal{U}$. Say students with their names.
- Within this universe, we should process a set of keys, $\mathcal{K}$, containing up to $n$ keys.
- The keys in the set $\mathcal{K}$ are initially unknown, and may change.
- We wish to map the set $\mathcal{K}$ to a table, $T = \{0, \ldots, m - 1\}$ of size $m$, where $m \approx n$.

Figures from MIT algorithms course, 2008.
Implementing Insert, Delete, Search

The universe of all possible keys, $\mathcal{U}$, is much much larger than the set of actual keys, $\mathcal{K}$, whose size is up to $n$. Mapping is by a (fixed) hash function, $h : \mathcal{U} \mapsto T$ that does not depend on $\mathcal{K}$.

- Given an item with key $k \in \mathcal{U}$.
- Compute $h(k)$ and check if in $T$ (this is search).
- If not, can insert item to cell $h(k)$ in $T$.
- If it is, can delete item from cell $h(k)$ in $T$.

If $h(k)$ can be computed in constant time and insertion/deletion can be implemented in constant worst case time, we will achieve our goal.

Since $|\mathcal{U}| \gg n$ and $h$ does not depend on $\mathcal{K}$, this last goal is clearly impossible.

If we are really unlucky, $h$ will map all $n$ keys in $\mathcal{K}$ to the same value. Going over all these items will take $O(n)$ steps, instead of the desired $O(1)$ steps.
Collisions of Hashed Values

We usually assume that the set of keys is generated independently of \( h \), so that the values \( h(k) \) are randomly distributed in the hash table. We will analyze hashing under this assumption.

We say that two keys, \( k_1, k_2 \in \mathcal{K} \) collide (under the function \( h \)) if \( h(k_1) = h(k_2) \).

Let \( |\mathcal{K}| = n \) and \( |\mathcal{T}| = m \), and assume that the values \( h(k) \) for \( k \in \mathcal{K} \) are distributed in \( \mathcal{T} \) at random. What is the probability that a collision exists? What is the size of the largest colliding set (a set \( S \subset \mathcal{K} \) whose elements are all mapped to the same target by \( h \)).

The answer to this question depends on the ratio \( \alpha = n/m \). This ratio is the average number of keys per entry in the table, and is called the load factor.

If \( \alpha > 1 \), then clearly there is at least one collision (pigeon hall principle). If \( \alpha \leq 1 \), and we could tailor \( h \) to \( \mathcal{K} \), then we could avoid collisions. However, such tinkering is not possible in our context.
Python’s hash Function

Python comes with its own hash function, from everything immutable to integers (both negative and positive).

```python
>>> hash(1)
1
>>> hash(0)
0
>>> hash(10000000)
10000000
>>> hash("a")
-468864544
>>> hash(-468864544)
-468864544
>>> hash("b")
-340864157
```

Note that Python’s hash function is not “truly’ random”. Yet what we care about is how it typically handles collisions, and it does seem to handle them well.

We intend to employ Python’s hash function for our needs. But we will have to make one important modifications to it.
Python’s *hash* Function, cont.

Python comes with its own hash function, from *everything immutable* to integers (both negative and positive).

```python
>>> hash("Benny")
5551611717038549197
>>> hash("Amir")
-6654385622067491745  # negative! I knew it in advance :-) 
>>> hash((3,4))
3713083796997400956
>>> hash([3,4])
Traceback (most recent call last):
  File "<pyshell#16>" , line 1, in <module>
    hash([3,4])
TypeError: unhashable type: 'list'
```
What concerns us mostly right now is that the range of Python’s hash function is too large.

To take care of this, we simply reduce its outcome modulo $p$, the size of the hash table. It is recommended to use a prime modulus.

```python
def hash_mod(key, p, func=hash):
    return func(key) % p
```

Note that our default parametr is using Python’s built-in `hash`. But we could (and would) employ other functions, (hopefully) good or bad.
Approaches for Dealing with Collisions: The First Approach

1) Chaining:

In our example of hashing students’ names, we are going to use chaining. We will implement and analyze chaining on this small list and also on much larger examples.
In our example we will hash students’ names, and are going to use chaining for resolving collisions. We will implement and analyze chaining on this small list and then on much larger examples.

We process the 6 items (students’ records) to be inserted into the hash table one by one. For each item, we apply the hash function, hash_mod, to its key (the student’s name). The result is an integer, \( \ell \), which is an index of an entry in the hash table \( (0 \leq \ell \leq m - 1) \).

We access the \( \ell \)-th element in the hash table, which is a list. We search this list sequentially. If an equal item is not found in this list, we append “our student” at the end of the list.
We’ll construct a hash table with $m = 23$ entries. We’ll insert $n = 14$ students’ record in it and check how insertions are distributed, and in particular what is the maximum number of collisions.

Our hash table will be a list with $m = 23$ entries. Each entry will contain a list with a variable length. Initially, each entry of the hash table is an empty list.

We employ a hash function that maps strings (possible names of students) to the range $\{0, 1, \ldots, 22\}$ (indices in the hash table). Given a student, we apply the hash function to its name, which is the key in our case.
Given a student, we apply the hash function to its name, which is the key in our case. The hash function in the code below is our `hash_mod`. If the result is \( \ell \), we will map (the record of) this student to entry number \( \ell \) in the hash table.

Please welcome our 14 new students (new to this class, that is):

```python
>>> names=['Reuben','Simeon','Levi','Judah','Dan','Naphtali','Gad','Asher','Issachar','Zebulun','Benjamin','Joseph','Ephraim','Manasse']
```

```python
>>> [(name, hash_mod(name,23)) for name in names]
```
Constructing the Hash Table: A Very Small Example

```python
>>> [(name, hash_mod(name, 23)) for name in names]
```

```python
>>> sorted(_, key=lambda x:x[1]) # easier to view collisions
```

In the example above, with \( n = 14, m = 23 \), we see that there are three collisions: Judah and Benjamin are both mapped to the same entry in the hash table, \( \ell = 2 \), Asher, Naphtali and Zebulun are mapped to the same entry in the hash table, \( \ell = 11 \), and Reuben and Manasse are both mapped to entry \( \ell = 14 \).

In this case, `hash_mod(name, 23)` performed rather poorly.

Question is, how shall we deal with such collisions?
A Slightly Larger Example \((n = 14, m = 31)\)

Let us take a somewhat bigger table \((m = 31\) slots\) for the same number of keys \((n = 14)\). Are there fewer collisions?

```python
>>> [(name, hash_mod(name, 31)) for name in names]
[('Reuben', 18), ('Simeon', 30), ('Levi', 24), ('Judah', 23),
 ('Dan', 16), ('Naphtali', 9), ('Gad', 26), ('Asher', 23),
 ('Issachar', 20), ('Zebulun', 5), ('Benjamin', 18), ('Joseph', 25),
 ('Ephraim', 1), ('Manasse', 4)]
```

```python
>>> sorted(_, key=lambda x:x[1])
[('Ephraim', 1), ('Manasse', 4), ('Zebulun', 5), ('Naphtali', 9),
 ('Dan', 16), ('Reuben', 18), ('Benjamin', 18), ('Issachar', 20),
 ('Judah', 23), ('Asher', 23), ('Levi', 24), ('Joseph', 25),
 ('Gad', 26), ('Simeon', 30)]
```

We see that there are just two collisions: Reuben and Benjamin are both mapped to the same entry in the hash table, \(\ell = 18\), Judah and Naphtali are mapped to the same entry in the hash table, \(\ell = 23\). No size 3 collision this time! In this case, `hash_mod(name, 31)` performed better than `hash_mod(name, 23)`.

Question still is, how shall we deal with such collisions?
Collisions’ Sizes: Throwing Balls into Bins

We throw \( n \) balls (items) at random (uniformly and independently) into \( m \) bins (hash table entries). The distribution of balls in the bins (maximum load, number of empty bins, etc.) is a well studied topic in probability theory.

The figure is taken from a manuscript titled “Balls and Bins – A Tutorial”, by Berthold Vöcking (Universität Dortmund).
A Related Issue: The Birthday Paradox

(figure taken from http://thenullhypodermic.blogspot.co.il/2012_03_01_archive.html)
The Birthday Paradox and Maximum Collision Size

A well known (and not too hard to prove) result is that if we throw \( n \) balls at random into \( m \) distinct slots, and \( n \approx \sqrt{\pi m/2} \), then with probability about 0.5, two balls will end up in the same slot. This gives rise to the so called “birthday paradox” – given about 24 people with random birth dates (month and day of month), with probability exceeding 1/2, two will have the same birth date (\( m = 365 \) here, and \( \sqrt{\pi \cdot 365/2} = 23.94 \)).

Thus if our set of keys is of size \( n \approx \sqrt{\pi m/2} \), two keys are likely to create a collision.
It is also known that if \( n = m \), the expected size of the largest colliding set is \( \ln n / \ln \ln n \).
Collisions of Hashed Values

We say that two keys, $k_1, k_2 \in \mathcal{K}$ collide (under the function $h$) if $h(k_1) = h(k_2)$.

Let $|\mathcal{K}| = n$ and $|\mathcal{T}| = m$, and assume that the values $h(k)$ for $k \in \mathcal{K}$ are distributed in $\mathcal{T}$ at random. What is the probability that a collision exists? What is the size of the largest colliding set (a set $S \subset \mathcal{K}$ whose elements are all mapped to the same target by $h$).

The answer to this question depends on the ratio $\alpha = n/m$. This ratio is the average number of keys per entry in the table, and is called the load factor.

If $\alpha > 1$, then clearly there is at least one collision (pigeon hall principle). If $\alpha \leq 1$, and we could tailor $h$ to $\mathcal{K}$, then we could avoid collisions. However, such tinkering is not possible in our context.
Collision Size

Let $|\mathcal{K}| = n$ and $|\mathcal{T}| = m$. It is known that

- If $n < \sqrt{m}$, the expected maximal capacity (in a single slot) is 1, i.e. no collisions at all.
- If $n = m^{1-\varepsilon}$, $0 < \varepsilon < 1/2$, the expected maximal capacity (in a single slot) is $O(1/\varepsilon)$.
- If $n = m$, the expected maximal capacity (in a single slot) is $\ln n / \ln \ln n$.
- If $n > m$, the expected maximal capacity (in a single slot) is $n/m + \ln n / \ln \ln n$. 
In our example of hashing students’ names, we are going to use chaining for resolving collisions. We will implement and analyze chaining on this small list and then on much larger examples.

We process just 6 items (students’ records), and insert them into the hash table one by one. For each item, we apply the hash function, hash_mod, to its key (the student’s name). The result is an integer, \( \ell \), which is an index of an entry in the hash table \( 0 \leq \ell \leq m - 1 \).

We access the \( \ell \)-th element in the hash table, which is a list. We search this list sequentially. If an equal item is not found in this list, we append “our student” at the end of the list.
The Student Class

The **key** will be the **name**, while the value will be the pair **israeli_id, grade**.

class Student:
    def __init__(self):
        self.name = generate_name()
        self.israeli_id = random.randint(2*10**7, 6*10**7)
        self.grade = random.randint(19, 99)

    def __repr__(self):
        return str.format("<{} ,{} ,{} >", self.name, self.israeli_id, self.grade)

    def __lt__(self, other):
        return self.name < other.name

def students(n):
    return [Student() for i in range(n)]

def next_prime(start):
    for i in range(start, 2*start):
        if is_prime(i):
            return i
def find(candidate_key, table, func=hash, mod=0):
    if mod == 0:
        mod = len(table)
    i = hash_mod(candidate_key, mod, func)
    list_of_items = table[i]
    k = contained(candidate_key, list_of_items)
    # print(i, k)  # diagnostic print
    if k != None:  # key exists in table
        return list_of_items[k].israeli_id, list_of_items[k].grade
    else:
        return None
def insert(new_item, table, func=hash, mod=0):
    """ insert an item into a hash table. If key already exists, the value is updated. Default mod for hash_mod is table’s size. Always returns None""
    if mod==0:
        mod=len(table)
    i=hash_mod(new_item.name,mod,func)
    list_of_items=table[i]
    k=contained(new_item.name,list_of_items)
    if k != None:  # key exists in table
        print(list_of_items[k])
        list_of_items[k]=new_item
        print(list_of_items[k])
    else:
        list.append(list_of_items,new_item)
    return None
Underlying Dictionary Operations
(hash_mod and contained)

def hash_mod(key,p,func=hash):
    return func(key) % p

def contained(candidate_key,list_of_items):
    """ checks if item is a member in list_of_items 
    returns location in list (if found), or None""
    for k in range(len(list_of_items)):
        if candidate_key==list_of_items[k].name:
            return k # return index of candidate in list
    return None
Initializing the Hash Table

def hash_table(m):
    return [[] for i in range(m)]
# initial hash table, m different empty entries

>> ht=hash_table(11)
>>> ht
[[], [], [], [], [], [], [], [], [], [], []]  
>>> ht[0] == ht[1]
True
>>> ht[0] is ht[1]
False

Since our table is a list of lists, and lists are mutable, we should be careful even when initializing the list.
The following code does produce the desired outcome, and is an alternative to the code presented in previous slide.

```python
def fixed_hash_table(m):
    empty=[]
    return [list(empty) for i in range(m)]
# initial hash table, m empty entries

>>> ht=fixed_hash_table(11)
>>> ht[0] is ht[1]
False

>>> list.append(ht[0],7)
>>> ht
[[7], [], [], [], [], [], [], [], [], []]
```
Initializing the Hash Table: a Bogus Code

Consider the following alternative initialization

```python
def bogus_hash_table(m):
    empty=[]
    return [empty for i in range(m)]
# initial hash table, m empty entries
```

```bash
>>> ht=bogus_hash_table(11)
>>> ht
[[], [], [], [], [], [], [], [], [], [], []]
>>> ht[0] == ht[1]
True
>>> ht[0] is ht[1]
True
```

The entries produced by `bogus_hash_table(n)` are identical. Therefore, mutating one mutate all of them:

```bash
>>> list.append(ht[0],7)
>>> ht
[[7], [7], [7], [7], [7], [7], [7], [7], [7], [7], [7]]
```
A Small Example with $n = 6$ and $m = 11$

```python
>>> ht=hash_table(11)
>>> st=students(6)
>>> insert(st[0],ht,mod=11)  # inserting an item
>>> ht
[[],[],[],[],[<Lebcja Lnifvcdo,22342180,50>],[],[],[],[],[],[]]
>>> insert(st[0],ht,mod=11)  # re-inserting same item
>>> ht
[[],[],[],[],[<Lebcja Lnifvcdo,22342180,50>],[],[],[],[],[],[]]
>>> for elem in st:  # inserting n=6 items
    insert(elem,ht,mod=11)
>>> for i in range(11):  # printing 11 hash table entries
    print(ht[i])

[<Vqbuc Oifvlb,23741272,34>] [<Oznsq Dhqybcjs,20832587,30>] [] []
[<Lebcja Lnifvcdo,22342180,50>, <Nfzml Juoksm,32983836,52>]
[<Rlds Jevmrt,22459615,55>] [] [] [<Xbradg Cjtfl,31028001,41>]
[] []
```
A Small Example with $n = 6$ and $m = 11$, cont.

We now try to find some objects.

```python
>>> st[0]
<Lebcja Lnifvcd0,22342180,50>
>>> find(st[0],ht)
    # result is None: we search for keys, not complete items

>>> find(st[0].name,ht)
(22342180, 50)  # result is a 2-tuple: israeli_id and grade

>>> find("Walt Disney",ht)
    # result is None: we searched for a non existing key
```
Measuring the Maximal Load

We are interested in the maximal number of collisions across all entries in the hash table. We denote this value by \texttt{max_load(table)}.

```python
def max_load(table):
    #""" max load and no. times attained """
    max_load = max(len(elem) for elem in table)
    return max_load, len([i for i in range(len(table)) if len(table[i]) == max_load])
```

This function returns the maximal load, as well as the number of entries attaining this maximal value.

Continuing the previous, small example:

```python
>>> max_load(ht)
(2, 1) # one collision of two items
```
Preparing **Larger** Hash Table and Students’ Lists

```python
def hash_table(m):
    return [[] for i in range(m)]
# initial hash table, m empty entries

def students(n):
    return { Student() for i in range(n)}

The sizes of our hash tables will be prime numbers. The following two functions will be useful: `is_prime` checks if a number is prime. `next_prime` produces the first prime greater or equal to its argument.

def is_prime(m):
    """probabilistic test for m’s compositeness ""
    for i in range(0,100):
        a = random.randint(1,m-1) # a random integer in [1..m-1]
        if pow(a,m-1,m) != 1:
            return False
    else:
        return True

def next_prime(start):
    for i in range(start,2*start):
        if is_prime(i):
            return i
```

39 / 56
Measuring Collisions’ Sizes

We will now run some experiments, primarily measuring maximal load for tables of size $m = 10^6$, and different number of inserted items, $n = m^{1/2}, m^{2/3}, m^{3/4}, m^{4/5}, m^{5/6}, m^{6/7}, m^{7/8}, m^{8/9}$.

For every such size, $n$, we repeat this experiment $t = 10$ times, in order to have some statistical meaning to the outcomes.

Note that the keys, namely the students’ names, are chosen anew, at random, in every experiment.

The maximal load is a random variable whose value we expect to vary from one run to the next. Yet, we expect the fluctuations to be fairly moderate: These random variables tend to concentrate fairly narrowly around their mean value.
Measuring Collisions' Sizes: Python Code

This code prepares an empty hash table of size \( m \). It then generates \( n \) random items \( (n = m^{1/2}, m^{2/3}, m^{3/4}, m^{4/5}, m^{5/6}, m^{6/7}, m^{7/8}, m^{8/9}) \), inserts them to the table, and measures the maximal loads.

```python
def sublinear_simulation(t, m=10**6, f=hash):
    """ performs hashing \( t \) times for \( m \) slots and
    \( n=m^{(1-\epsilon)} \), \( \epsilon = 1/2, 1/3, \ldots , 1/9 \)
    measures maximal load and multiplicity of occurrence """
    modulus = next_prime(m)
    n = [int(m**(1 - 1/(2 + i))) for i in range(8)]
    loads = []
    for i in range(8):
        for times in range(t):
            students_list = students(int(m**(1 - 1/(2 + i))))
            table = hash_table(modulus)
            for s in students_list:
                insert(s, table, func=f)
            loads[i].append(max_load(table))
    return loads

>>> loads = sublinear_simulation(10, m=10**6)
```

Repeat the experiment \( t=10 \) times for \( m = 10^6 \) and each value of \( n = m^{1-\epsilon} \), where \( \epsilon = 1/2, 1/3, \ldots , 1/8, 1/9 \).
Actual Collision Sizes - No. Keys Sublinear in Table Size

First element in each list equals $n$. Second one is a pair (maximal load, how many times) in the first try; then such pair for the second try, etc.

```python
>>> for i in range(8):  # from $n=m^{(1 - 1/2)}$ to $n=m^{(1 - 1/9)}$
    print (loads[i])

[1000, (1, 1000), (2, 3), (2, 1), (1, 1000), (1, 1000), (1, 1000),
 (1, 1000), (2, 3), (1, 1000), (2, 1)]  # $n=m^{(1/2)}$
[10000, (3, 1), (2, 48), (2, 53), (3, 1), (3, 1), (2, 68), (2, 44),
 (3, 1), (2, 59), (2, 57)]  # $n=m^{(2/3)}$
[31622, (3, 6), (3, 4), (3, 5), (4, 1), (3, 2), (3, 5), (3, 4),
 (3, 5), (3, 8), (3, 5)]  # $n=m^{(3/4)}$
[63095, (4, 1), (3, 44), (3, 36), (4, 1), (5, 1), (3, 53), (4, 1),
 (4, 1), (3, 46), (4, 1)]  # $n=m^{(4/5)}$
[100000, (4, 4), (4, 3), (4, 6), (4, 4), (5, 1), (4, 4), (4, 4),
 (4, 4),(4, 5), (4, 7)]  # $n=m^{(5/6)}$
[138949, (4, 15), (4, 19), (4, 9), (4, 11), (4, 14), (5, 1),
 (4, 16), (5, 1), (4, 12), (5, 1)]  # $n=m^{(6/7)}$
[177827, (6, 1), (4, 34), (5, 2), (5, 1), (5, 1), (4, 30), (5, 3),
 (4, 31), (5, 1), (5, 2)]  # $n=m^{(7/8)}$
[215443, (5, 1), (5, 3), (5, 1), (5, 5), (6, 1), (5, 4), (5, 3),
 (5, 6), (5, 2), (5, 2)]  # $n=m^{(8/9)}$
```
Collision Sizes, Again - No. Keys Sublinear in Table Size

```python
>>> loads = sublinear_simulation(10, m=10**6)

>>> for i in range(8):  # from \(n=m^{1-1/2}\) to \(n=m^{1-1/9}\)
    print(loads[i])  # first element in each list equals \(n\)

[1000, (1, 1000), (2, 2), (1, 1000), (1, 1000), (2, 1), (1, 1000), (2, 1), (2, 2)]
[10000, (2, 52), (2, 51), (2, 56), (2, 62), (2, 46), (2, 43), (2, 52), (2, 46), (3, 1), (2, 40)]
[31622, (3, 3), (3, 11), (3, 5), (3, 1), (3, 4), (3, 5), (3, 4), (3, 4), (3, 8)]
[63095, (3, 33), (4, 1), (4, 2), (3, 33), (5, 1), (3, 44), (4, 1), (3, 53), (3, 45), (5, 1)]
[100000, (4, 4), (4, 6), (5, 1), (4, 3), (4, 4), (4, 3), (4, 6), (4, 5), (5, 1), (4, 3)]
[138949, (4, 18), (4, 14), (5, 1), (5, 1), (4, 15), (5, 1), (4, 14), (4, 18), (4, 16), (4, 15)]
[177827, (4, 32), (4, 29), (4, 41), (5, 2), (4, 39), (5, 1), (5, 2), (5, 2), (5, 3), (5, 2)]
[215443, (5, 4), (5, 2), (5, 2), (5, 5), (5, 2), (5, 1), (5, 2), (5, 1), (5, 3), (5, 4)]
```
We will now run some experiments, measuring maximal load for tables of size \( m = 10^6 \), and number of keys \( n = m \). We repeat this experiment \( t = 10 \) times, in order to have some statistical meaning to the outcome. Recall that we expect the maximal load to be \( \ln n / \ln \ln n \).

```python
def linear_simulation(t, m=10**6, f=hash):
    """ performs hashing t times for m slots and n=m .
    measures maximal load and multiplicity of occurrence ""
    modulus = next_prime(m)
    loads = [m]
    for i in range(t):
        students_list = students(m)
        # new students’ list of size m
        table = hash_table(modulus)  # restart hash table
        for s in students_list:
            insert(s, table, func=f)
        loads.append(max_load(table))  # add load to loads
    return loads
```
Actual Collision Sizes, $n = m$

```
>>> linear_simulation(10) # n=m=1000000
[1000000, (9, 1), (9, 1), (8, 12), (9, 3), (8, 9), (9, 1), (10, 1),
 (9, 1), (9, 1), (9, 1)]
```

How is the outcome compared to the theoretical bound, $\frac{\ln n}{\ln \ln n}$? Let the interpreter do the calculation

```
>>> import math
>>> def ln(n): return math.log(n, math.e)
>>> ln(10**6)/ln(ln(10**6))
5.261464353591485
```

Execution time here is quite high. For example, these 10 executions took well over 7 minutes. Most time is taken by the randomized generation of names, IDs, and grades.
Measuring Collision Sizes, $n = 5 \cdot m$: Code

We will now run some experiments, measuring maximal load for tables of size $m = 10^6$, and number of keys $n = 5 \cdot m$. We repeat this experiment $t = 10$ times, in order to have some statistical meaning to the outcome. Recall that we expect the maximal load to be $(m/n) + \ln n / \ln \ln n = 5 + \ln n / \ln \ln n$.

```python
def super_linear_simulation(t, m=10**6, f=hash):
    """performs hashing t times for m slots and n=5*m .
    measures maximal load and multiplicity of occurance ""
    modulus = next_prime(m)
    loads = [m]
    for i in range(t):
        students_list = students(5*m)  # new students’ list of size m
        table = hash_table(modulus)    # restart hash table
        for s in students_list:        # restart students list:
            insert(s, table, func=f)
        loads.append(max_load(table))  # add load to loads
    return loads
```

Actual Collision Sizes, $n = 5 \cdot m$

```python
>>> super_linear_simulation(10, m=10**5)
[100000, (16, 1), (17, 2), (17, 4), (17, 1), (17, 2),
  (18, 2), (17, 1), (17, 3), (17, 1), (17, 2)]
```

How is the outcome compared to the theoretical bound, $5 + \frac{\ln n}{\ln \ln n}$? Again, let the interpreter do the calculation

```python
>>> import math
>>> 5+math.log(10**5)/math.log(math.log(10**5))
9.711710714547694
```

Execution time here is quite high. For example, these 10 executions took well over 12 minutes.
Find, Insert, Delete, with Hash Tables

We have implemented the operations insert and find. We deliberately did not implement deletion. But by now, this should be an easy exercise for you.
Approaches for Dealing with Collisions: The Second Approach

2) **Open Addressing:**

In open addressing, each slot in the hash table contains at most one item. This obviously implies that $n$ cannot be larger than $m$. Furthermore, an item will typically not stay statically in the slot where it “tried” to enter, or where it was placed initially. Instead, it may be moved a few times around.
Open Addressing, cont.

Open addressing is important in hardware applications where devices have many slots, but each can only store one item. Fast switches and high capacity routers are examples of this.

We will not discuss “strict” open addressing in this course.

However, in the next lecture we will discuss a related approach, using more than one hash function (two, three, or four different functions). This is known as cuckoo hashing (Pagh and Rodler, 2001).
And Now For Something Completely Different: Using Hashing To Solve Large String Problems

For Rent

In Levanda street, 150/200/350 meters, open space, airconditioned, parking, for a Church

Or any other purpose

054-4983338 / 054-4443104

(photo taken in the corner of Ha’Aliya and Levinsky St., Tel-Aviv, April 2014.)
Using Hashing To Solve Large String Problems

We will discuss (on the board) the naïve solution to the homework problem 4b (Fall semester, 2011/12), which is \textit{quadratic} in the length of the input strings, thus totally infeasible if both strings are millions characters long. We will then outline how \textit{hashing} can be used to yield a much more efficient algorithm.

Our interest in this problem (as well as yours :-) stems from the fact that in HW5 this semester you will have a similar problem (albeit dealing with bacterial genomes, instead of proteomes).

\[
\]
Large String Problems: naïve approach

Let us look at the task 4b above. We are given two strings $S$ and $T$, both of length $n$. We want to find (contiguous) substrings of maximal length that are exactly shared by $S$ and $T$.

We do not know in advance what this maximal length will be. Let us say we are now looking for substrings of length $\ell$.

The naïve method is to compare all substrings of length $\ell$ from $S$ with all substrings of length $\ell$ from $T$. A single comparison involves exactly $\ell$ character comparisons. The number of such substrings in each of $S$ and $T$ is exactly $n - \ell$. So overall, the naïve approach takes $(n - \ell)^2 \cdot \ell \approx n^2 \cdot \ell$ operations (since $\ell \ll n$).

If $n \approx 10^6$ and $\ell \approx 10^2$, such task will take about $10^{14}$ operations and is completely infeasible.
Using Hashing To Solve Large String Problems: Insertion

Let us consider a hashing based approach. We could create an empty Python dictionary called `cholera_dict`. We then hash all substrings of length \( \ell \) of \( S \), using Python’s \texttt{hash}. The key is the substring (with \( \ell \) letters), and the value is the location of this substring in \( S \). \textit{E.g.}, for \( \ell = 40 \), ’TTTCATCAGGTCGTTTATGGTAATTTTTTTTCATGTTTAGT’ equals the first \( \ell \) characters of the cholera genome. So we would set \texttt{cholera_dict[’TTTCATCAGGTCGTTTATGGTAATTTTTTTTCATGTTTAGT’]=0}

How much time does this take? Hashing a substring costs \( O(\ell) \) operations. The number of substrings is \( n - \ell = O(n) \). So overall, inserting all substrings to the hash table we created would take \( O(n\ell) \) expected number of operations.
We now go over all length $\ell$ substrings of $T$, one by one, and try to find each in the hash table. For example, we could make the query "TTTCACTCAGGTCGTATGTAACGCTTGGATTTCAGTCCC" in cholera_dict. If found, we perform a comparison (check equality) with the corresponding substring of $S$ (why bother?).

The number of operations needed to look up a single $\ell$ long substring is $O(\ell)$. If found, the comparison also takes $\ell$ operations. All by all, to process one substring of $T$ takes $O(\ell)$ operations. There are $n - \ell = O(n)$ substrings. So this finding phase takes $O(n\ell)$ operations.
Using Hashing To Solve Large String Problems: Time Analysis

Both “insert” and “find” take $O(n\ell)$ operations on the average (with a small constant in the big O notation).

If $n \approx 10^6$ and $\ell \approx 10^2$, such task will take a small constant times $10^8$ operations, which is completely feasible.

We did not say anything on how $\ell$ of maximal length substrings can be found. We can use a process similar to binary search. This will be explained in the text of HW5.