# WORKSHEET, WEEK 11: HASHTABLES

A hash function $\mathcal{U} \to \{0, 1, \ldots, m-1\}$ maps the universe $\mathcal{U}$ of keys into $m$ non-negative integers (the size of the hashtable). It is used for map ADT (insert, delete and exact search). They are used for efficient storing maps and sets.

## 11.1 Concepts and Facts

Hash function should follow the following guidelines:

- Keys are discrete objects having a discrete representation (large integers, strings, lists and other data structures). Hash function could be defined on float variables as well, but it is **not** a real-valued function: it depends on the way the real values are encoded.
- The hash of an object must not change during its lifetime (either the hashable object is immutable or the hash function depends only on immutable attributes).
- `a == b` implies `hash(a) == hash(b)`. (The reverse might fail during a hash collision).
- A hash function should be fast. It is often assumed to be constant-time $O(1)$; sometimes it is linear-time $O(|k|)$ for longer input keys $k$.

**Simple Uniform Hashing Assumption:** Under this assumption a key maps to any slot $\{0, \ldots, m-1\}$ with the same probability, independent from the hashes of all the other keys.

This can be approximated by randomized seed to the hashing algorithm; hash function should map similar, but non-identical objects far from each other. (People sharing the same lastname or birth year should have considerably different hashes.)

---

**Note:** Hashing for datastructures (hashtables) does not use *cryptographic hashing*. Cryptographic hashes should create a unique *fingerprint* for the object being hashed. Examples include SHA-256 and other algorithms in SHA-2 family (MD5 no longer considered safe).

---

In practice most hash functions are computed as a composition $h(obj) = h_2(h_1(obj))$:

- $N = h_1(obj)$ is a *prehash function* which takes objects of various types and returns large integers.
- $h_2(N)$ is usually a modular division to get a nonnegative integer within the desired range.

In Python the prehash function is named simply `hash(...)`; it returns signed 64-bit integers. It can take different argument types. For small integer arguments it returns the integer itself; larger integer objects are hashed to fit into 64-bits. For example, you can run the Python environment on Linux like this (tested on Ubuntu command-line):

```
export PYTHONHASHSEED=0
python
```

On Windows Powershell (such as Anaconda terminal) run Python environment like this:

```
$Env:PYTHONHASHSEED=0
```

On Windows regular terminal run Python environment like this:

```
set PYTHONHASHSEED=0
python
```

```
>>> hash(10**18)
1000000000000000000
>>> hash(10**19)
776627963145224196
>>> hash('\x61\x62\x63')
5573379127532958270
>>> hash('abc')  # 'abc' is same string as '\x61\x62\x63'
5573379127532958270
>>> hash(3.14)
322818021289917443
>>> hash((1,2))
-3550055125485641917
```

**Hash Collisions:** Finding hash collisions can sometimes be used to slow down Python-related data structures such as dictionaries. Here is advice how to find hash collisions efficiently: https://bit.ly/3nf4bdk. Instead of looping until two hash values will collide, the approach uses some reverse enginering and also Meet in the Middle which reduces the number of checks to be made. In an unsophisticated Python implementation of a Web application one could cause massive hash collisions to stage denial of service attacks. Since Python 3.2 the hash computations are randomized with a seed (`PYTHONHASHSEED` environment variable); such attacks are now much harder. See https://bit.ly/3CeAVYi on how these collisions affect other programming languages.

### 11.1.1 String Matching Problem

**Definition:** We have a text $T$ and a pattern $P$. We need to find all the offsets where the pattern $P$ occurs in the text.

**Naive String Matching:** Here is a straightforward solution that finds all offsets:

NAIVESTRINGMATCHING$(T, P)$:
    $n = len(T)$
    $m = len(P)$
    **for** $s = 0$ **to** $n - m$:
        **if** $P[0 : m - 1]$ == $T[s : (s + m - 1)]$:
            **output** Pattern found at offset, $s$

We can find worst-case behavior. Consider the following pattern and text:

$$\begin{cases} P = \texttt{aaab} \\ T = \texttt{aaaaaaaaaaaaaa} \end{cases}$$

You will end up comparing all four times for any offset $s = 0, \ldots, 10$. The total number of comparisons is $4 \cdot 11 = 44$. In general the number of comparisons is $O(m \cdot (n - m + 1)) = O(m \cdot n)$.

**Rabin-Karp Algorithm** Hashing algorithms can be used for faster string searching:

RABINKARPSTRINGMATCHING($T, P, a, q$)
    $rP = $ HASH([])
    $rT = $ HASH([])
    **for** $i$ **in** RANGE($0, len(P)$):
        $rP$.APPEND($P[i]$)
        $rT$.APPEND($T[i]$)
    **for** $i$ **in** RANGE($len(P), len(T)$):
        **if** $rP$.HASH() == $rT$.HASH():
            (*Here we need to double-check as collisions are possible*)
            **if** $P == T[i - len(P) + 1 : i + 1]$
                **output** Pattern found at offset, $i - len(P) + 1$
        $rT$.SKIP($T[i - len(P)]$)
        $rT$.APPEND($T[i]$)

Can we ensure that false matches (hash collisions) do not happen more frequently than with the probability $1/len(P)$?

**Definition:** Multi-String Matching Problem. We have a text $T$ of length $n$ as before. But now we have not just one pattern to search, but a set of $k$ patterns $\mathcal{P} = \{P_0, P_1, \ldots, P_{k-1}\}$; each pattern has the same length $m$.

**Rabin-Karp Multistring Algorithm** Hashing algorithms can be used for faster string searching:

RABINKARPMULTISTRING($T, \mathcal{P}, m$):
    $hashes$ := $Set$.EMPTY()
    **foreach** $P_i \in \mathcal{P}$:
        $rP_i = $ ROLLINGHASH([])
        **for** $j$ **in** RANGE($0, m$):
            $rP_i$.APPEND($P[j]$)
        $hashes$.INSERT($rP_i$.HASH())
    $rT = $ ROLLINGHASH([])
    **for** $j$ **in** RANGE($0, m$):
        $rT$.APPEND($T[i]$)
    **for** $j$ **in** RANGE($1, n - m + 1$)
        **if** $rT$.HASH() $\in hashes$ **and** $T[j : j + m - 1] \in \mathcal{P}$
            **output** Pattern found at offset, $j$
        $rT$.SKIP($T[j]$)
        $rT$.APPEND($T[j + m]$)

This algorithm would take $O(n + km)$ running time. Naive string matching could take $O(nmk)$ running time, if we probe all the $k$ patterns one by one.

### 11.1.2 Rolling Hash

**Definition:** Rolling hash is an ADT: It is a data structure that accumulates some input fragment as a sort of list/queue and supports the following operations:

- $RH$ := ROLLINGHASH([]) – initialize a rolling hash to an empty list of symbols.
- $RH$.HASH() – return the hash value from the current list.
- $RH$.APPEND($val$) – adds symbol `val` to the end of the list (like `enqueue(val)` for a queue)
- $RH$.SKIP($val$) – removes the front element from the list (like `dequeue(val)` for a queue). Parameter `val` is often implicit as it is stored at the front of the list stored in the rolling hash.

In the case of strings, the list is a list of characters. It can be a list of anything, but elements on that list are represented as integers in some encoding. For example if we interpret characters as ASCII codes, then character `'A'` is stored as 65 and `'B'` is stored as 66.

We want to treat a list of items as a multidigit number $u \in \mathcal{U}$ in base $a$ (the list in the rolling hash is interpreted as a big number). For example, we can choose $a = 256$, the alphabet size for ASCII code.

**Polynomial Rolling hash:** This is one of the most popular implementations for rolling hashes; it uses modular arithmetic. Pick some prime number $q$ such that the number base $a$ is not divisible by $q$. Define the three ADT functions as follows:

- $hash() = (u \bmod q)$
- $append(val) = ((u \cdot a) + val) \bmod q = ((u \bmod q) \cdot a + val) \bmod q$
- $skip(val) = (u - val \cdot (a^{|u|-1} \, mod \, q)) \bmod q = \big((u \bmod q) - val \cdot (a^{|u|-1} \bmod q)\big) \bmod q$

**Example:** Pick $a = 100$ and $q = 23$. Let `RH` be a rolling hash storing `[61, 8, 19, 91, 37]`. We can compute hash value:

$$hash([61, 8, 19, 91, 37]) = (6108199137 \bmod 23) = 12.$$

In general

$$hash([d_3, d_2, d_1, d_1]) = \big(d_3 \cdot a^3 + d_2 \cdot a^2 + d_1 \cdot a^1 + d_0 \cdot a^0\big) \bmod q$$

To make it easier to compute, consider computation with a Hamming code:

$$hash([d_3, d_2, d_1, d_1]) = (((d_3 \cdot a + d_2) \cdot a + d_1) \cdot a + d_0) \bmod q$$

Making this faster:

- Cache the result $(u \bmod p)$ (memorize it in the rolling hash data structure).
- Avoid exponentiation in skip: cache $a^{|u|-1} \bmod p$.
- To append: multiply the cashed $(u \bmod p)$ by base $a$.
- To skip: divide $(u \bmod p)$ by base (division is expensive, can use multicative inverse).

**Rolling Hash with a Cyclic Polynomial (Buzhash):** First introduce an arbitrary hash function `h(c)` mapping single characters to integers from the interval $\big[0, 2^L\big)$. Assume that there are not too many characters and the function can be defined by a lookup table. Define the cyclical shift (bit rotation) to the left. For example, $s(\texttt{1011}) = \texttt{0111}$. The rolling hash function for a list of characters $[c_1, \ldots, c_k]$ is defined by this equality:

$$H = s^{k-1}(h(c_1)) \oplus s^{k-2}(h(c_2)) \oplus \ldots \oplus s(h(c_{k-1})) \oplus h(c_k).$$

It looks like a polynomial, but the powers are replaced by rotating binary shifts. The result of this hash function is also a number in $\big[0, 2^L\big)$.

## 11.2 Problems

**Problem 1 (Hash Table with Chaining):** The hash function used in this exercise is computed as Pythons `hash()`, and then the remainder `hash(x)` mod 11 is found.

**(A)** Draw a hashtable with exactly 11 slots (enumerated as `T[0]`, `T[1]`, and so on, up to `T[10]`). Insert the following items into this hashtable (keys are country names, but values are float numbers showing their population in millions):

```
('Austria',8.9), ('Azerbaijan',10.1), ('Belgium',11.6), ('Bulgaria',6.9),
('Estonia',1.3), ('Italy',59.6), ('Latvia',1.9), ('Lithuania',2.8)
```

If there are any collisions between the hash values, add the additional hash values to a linked list using *chaining*. Each item in the linked list contains a key-value pair and also a link to the next item.

**(B)** What is the expected lookup time, if we randomly search any of the 8 countries to look up its population. Finding an item in a hash table takes 1 time unit; following a link in a linked list also takes 1 time unit.

**(C)** What is the expected lookup time, if the 11-slot hashtable is randomly filled with 8 keys (each key has equal probability to be in any of the slots). You can find this lookup time rounded up to one tenth (one decimal digit precision) – analytic methods as well as a computer simulation is fine.

**Answer:**

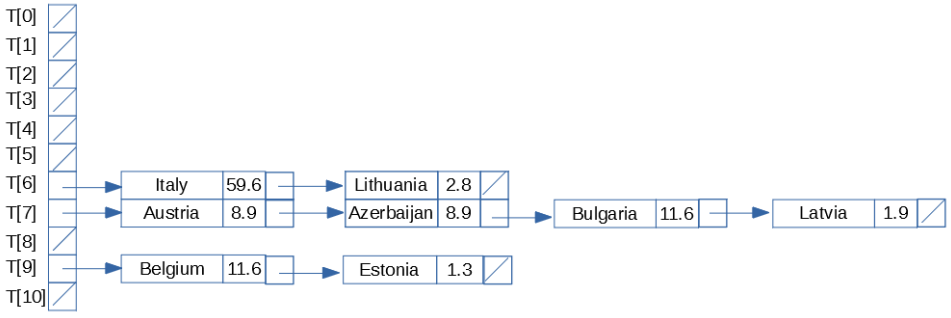
**(A)** This Python code snippet computes hash values for strings `'a'`, `'ab'`, `'abc'`. First set up the Python environment to compute (repeatable) hash function values:

```
$Env:PYTHONHASHSEED=0
python
```

In the interactive Python environment compute the values of the hash function:

```
list(map(lambda x: hash(x) % 11, ['Austria', 'Azerbaijan', 'Belgium',
'Bulgaria', 'Estonia', 'Italy', 'Latvia', 'Lithuania']))

[7, 7, 9, 7, 9, 6, 7, 6]
```




**(B)** The lookup time can take values $T = 1$, $T = 2$, $T = 3$, and $T = 4$, but with different probabilities. The expected lookup time for a random value is given by the expression:

$$E(T) = 1 \cdot \frac{3}{8} + 2 \cdot \frac{3}{8} + 3 \cdot \frac{1}{8} + 4 \cdot \frac{1}{8} = \frac{16}{8} = 2.$$

**(C)** The *load factor* in this case is $8/11$, so the average linked list has length $8/11$. The total space of this data structure is $O(m + n)$ (first store a table with $m$ entries, then store $n$ items belonging to our map). The total time for a lookup can be computed as $1 + 8/11$ (one plus the load factor of the hashtable).

**Problem 2 (Computing Rolling Hash – Polynomial Method):** Assume that we have an alphabet of 100 symbols. They are denoted by pairs of digits: $\{00, 01, \ldots, 99\}$. Select the sliding window size $m = 5$ and the prime number for modulo $q = 23$.

Let the input be `[3, 14, 15, 92, 65, 35, 89, 79, 31]`.

- Initialize an empty rolling hash `rH` and add the first five numbers using `append()` function defined as follows:

$$append(val) = ((u \cdot a) + val) \bmod q = ((u \bmod q) \cdot a + val) \bmod q.$$

- Show how the rolling hash can roll from the list `[3, 14, 15, 92, 65]` to `[14, 15, 92, 65, 35]` (first skip value `3`, then append value `35`.

**Problem 3 (Computing Rolling Hash – Cyclic Polynomial):** Consider 16 Latin letters with randomly assigned 4-bit codes (i.e. $L = 4$):

| Letter | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| h(x) | 1100 | 0100 | 0010 | 0110 | 0111 | 0000 | 1001 | 1111 | 0101 | 1101 | 1110 | 1011 | 0011 | 1010 | 1000 | 0001 |

Also assume that the sliding window has size $k = 5$. Find the hash value for $ABIDE$ and then rotate it over to $BIDEN$. Please recall that the rolling hash uses the following formula:

$$H = s^{k-1}(h(c_1)) \oplus s^{k-2}(h(c_2)) \oplus \ldots \oplus s(h(c_{k-1})) \oplus h(c_k),$$

**Problem 4 (Rolling Hash ADT with Cyclic Polynomial):** Write formulas for the Rolling Hash functions (when using Cyclic Polynomial or Buzhash):

- Formula to initialize `RH` to an empty list.
- Formula to append a new character $c_i$ to the existing list (without skipping anything).
- Formula to skip the head of the existing list $c_j$ (without appending anything).

**Problem 5 (Average Complexity of Naive String Matching)** Suppose that pattern $P$ and text $T$ are randomly chosen strings of length $m$ and $n$, respectively, from an alaphabet with $a$ letters ($a \geq 2$). What is the expected character to character comparisons in the naive algorithm, if any single comparison has a chance $1/a$ to succeed?

**Problem 6:** Assume that we need to create an set containing short phrases written in some human language. Use the following hash function: Compute the sum of character values (modulo the size of our hash table). Will the performance of such hash function implementation be as good as Pythons *hash()* function (also modulo the size of the hash table)? If there are risks using the character value sum, explain these risks.