General Purpose Hash Function Algorithms

www.partow.net

.: Home :. .: Links :. .: Search :. .: Contact :.

Main Menu
About
Programming
Projects
Miscellaneous

Topics
String Toolkit Library
C++ Mathematical Expression Library
Callbacks In C++
General Purpose Hash Function Algorithms
C++ Bitmap Library
C++ TCP Proxy Server
C++ Bloom Filter Library
C++ DSV Filter Library
C++ Vector Expression Template Library
C++ Lexer Library
C++ Date And Time Parsing Utilities
C++ Logging Toolkit
C++ Summation Toolkit
Wykobi Computational Geometry Library
Schifra Reed Solomon Error Correcting Code Library
C++ Makefile Template
Win32 Registry Activity Monitor
String Tokenizer
Primitive Polynomials

Introduction

Hash functions are by definition and implementation generally regarded as Pseudo Random Number Generators (PRNG). From this generalization it can be assumed that the performance of hash functions and comparisons between other hash functions can be determined by modeling the functions as PRNGs.

Analysis techniques such a Poisson distribution can be used to analyze the collision rates of different hash functions for different groups of data. In general there is a theoretical hash function known as the **Perfect Hash Function** for any specific group of data. The perfect hash function by definition states that no collisions will occur meaning no repeating hash values will arise from different elements of the group.

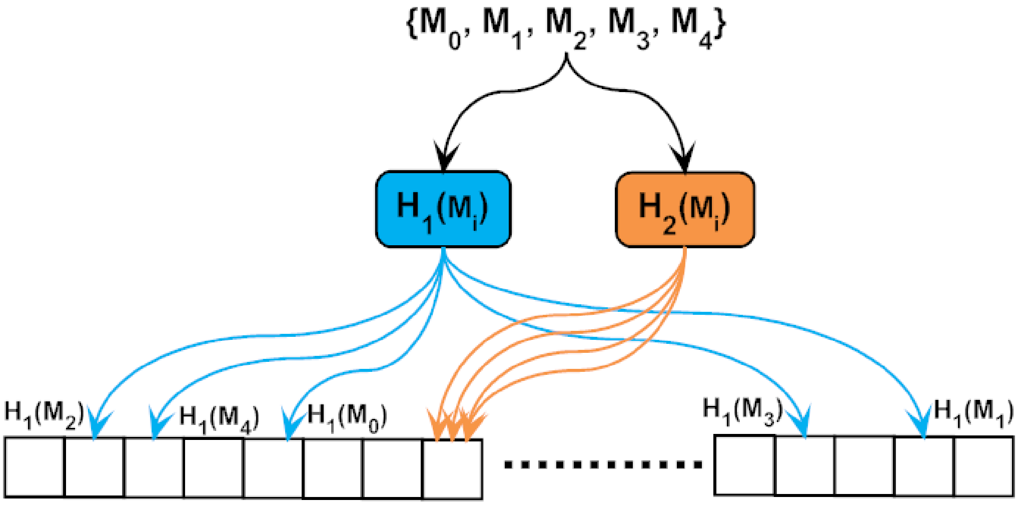
In reality it is very difficult to find a perfect hash function for an arbitrary set of data, and furthermore the practical applications of perfect hashing and its variant minimal perfect hashing are quite limited. So instead one may choose to pursue the concept of an **Ideal Hash Function**, which is commonly defined as a function that produces the *least* amount of collisions for a particular set of data.

One of the fundamental problems with hashing data or specifically mapping values from one domain to another, is that there are soo many permutations of types of data, some highly random, others containing high degrees of patterning or structure that it is difficult to generalize a hash function for all data types or even for specific data types. All one can do is via trial and error and utilization of formal hash function construction techniques derive the hash function that best suites their needs. Some factors to take into account when choosing hash functions are:

Data Distribution

This is the measure of how well the hash function distributes the hash values of elements within a set of data. Analysis in this measure requires knowing the number of collisions that occur with the data set meaning non-unique hash values, If chaining is used for collision resolution the average length of the chains (which would in theory be the average of each bucket's collision count) analyzed, also the amount of grouping of the hash values within ranges should be analyzed.

In the following diagram there are five unique messages to be hashed (M_0,...,M_4) and inserted into a hash-table. Each message is hashed using two different hash functions $\mathbf{H_1}$ and $\mathbf{H_2}$. The diagram depicts how H_1 generally distributes the hash values for the messages evenly over the given bucket-space *(table)*, where as H_2 returns the same value for each message causing numerous collisions.



Hash Function Efficiency

This is the measure of how efficiently the hash function produces hash values for elements within a set of data. When algorithms which contain hash functions are analyzed it is generally assumed that hash functions have a complexity of O(1), that is why look-ups for data in a hash-table are said to be *"on average of O(1) time complexity"*, where as look-ups of data in other associative containers such as maps (typically implemented as Red-Black trees) are said to be of O(logn) time complexity.

A hash function should in theory be a very quick, stable and deterministic operation. A hash function may not always lend itself to being of O(1) complexity, however in general the linear traversal through a string or byte array of data that is to be hashed is so quick and the fact that hash functions are generally used on primary keys which by definition are supposed to be much smaller associative identifiers of larger blocks of data implies that the whole operation should be quick, deterministic and to a certain degree stable.

The hash functions in this essay are known as simple hash functions or **General Purpose Hash Functions**. They are typically used for data hashing (string hashing). They are used to create keys which are used in associative containers such as hash-tables. These hash functions are not cryptographically safe, they can easily be reversed and many different combinations of data can be easily found to produce identical hash values for any combination of data.

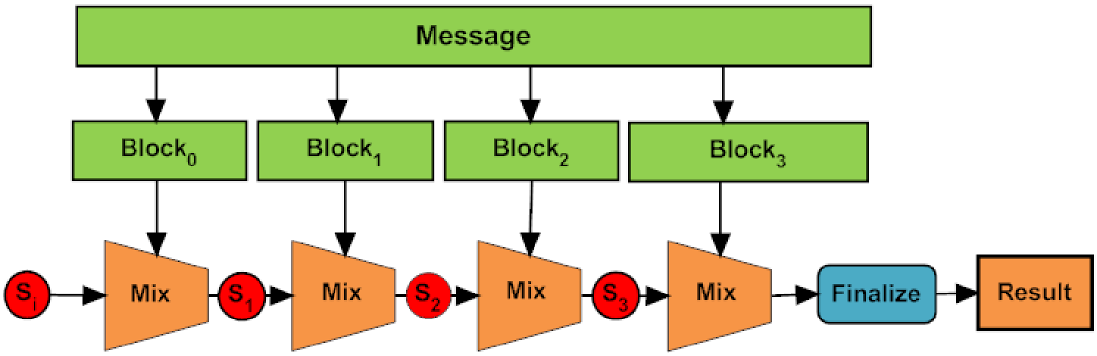
Hashing Methodologies

When designing and implementing hash functions the common building blocks typically used are bitwise operations, mathematical operations and look-up tables. These operations are applied either to individual bytes or blocks of bytes (words etc), furthermore they are fast, deterministic and readily available on most CPU architecture, making them ideal for implementing hash functions. The following is a list of the commonly used operations:

- **Bitwise Operations:** Not (!), Or (|), And (&), Xor (^), Shift-Left/Right (<<, >>), Rotate-Left/Right (<<<, >>>)
- **Mathematical Operations:** Addition (+), Multiplication (*)
- **Lookup Tables:** List of prime numbers, List of magic numbers, S-Box, P-Box

A common method for constructing collision resistant cryptographic hash functions is known as the **Merkle–Damgard construction**. A simplified version of this method can be used to easily generate well performing general purpose hash functions. In this method a piece of data (message) is transformed into a hash value as follows:

1. Initialise an internal state
2. Consume the message N-bits at a time (aka block size typically 32-bits, 64-bits, 128-bits etc..)
3. Perform a mixing operation with the current block and the internal state
4. Update the internal state with the the result of **[3]**
5. If there are any remaining bytes in the message proceed to **[2]**
6. Finalize the internal state and return the hash value



Note: The term **Internal State** (IS), depicted in the diagram as red circles, refers to a piece of memory, that is to be at least as large as the number of bits expected in the hash value. The IS at the start of the hash function will be initialised with a predetermined value - this is typically known as the **Initialisation Vector** (IV). The IS is primarily used during the mixing process and can generally be thought of as the final state of the hash function after the previous round. Its use in the mixing process is to create a data dependency between the result of the mix with the current message block and the results of the mixes from all the preceding message blocks. This process is commonly referred to as: Chaining

Note: The term **Finalize** means to take the internal state and prepare the final hash value that will be returned from the function. In the event the IS contains more bits than is required for the hash value, the IS itself will undergo a mixing process that will result in a value that is size compatible with the hash value type. Some implementations will also attempt to integrate side information such as the total number of bits hashed, or mixes with magic numbers based on the final value of the IS.

The "Mix" operation denoted above takes as input the internal state and the current message block, performs some computation and returns the value of the internal state. The following is a pseudo code example of a possible implementation of "Mix", that operates on blocks and internal state of size *32-bits* at a time:

```
uint32_t mix(uint32_t message_block, uint32_t internal_state)
{
   return (internal_state * message_block) ^
          ((internal_state << 3) + (message_block >> 2));
}
```

The "Mix" operation denoted above, will compute the multiplication between the current message block and the internal state, it will then proceed compute the sum of the internal state shifted by three bits to the left and the message block shifted by two bits to the right, then it will bitwise xor the first computation with the second and return that value as the result of the mix.

Putting it all together requires looping over all of the message *32-bits* at-a-time, performing the mix operation upon each block, updating the internal state and taking care of any remainder bits after the main loop has completed. The resulting hash function will generally look like the following:

```
uint32_t hash(const char* message, size_t message_length)
{
   uint32_t internal_state = 0xA5A5A5A5; // IV: A magic number
   uint32_t message_block = 0;

   // Loop over the message 32-bits at-a-time
   while (message_length >= 4)
   {
      memcpy(message_block, message, sizeof(uint32_t));

      internal_state = mix(message_block, internal_state);

      message_length -= sizeof(uint32_t);
      message        += sizeof(uint32_t);
   }

   // Are there any remaining bytes?
   if (message_length)
   {
      memcpy(message_block, message, message_length);
      internal_state = mix(message_block, internal_state);
   }

   return internal_state;
}
```

Lookup Tables (LUTs)

Various hash function designs utilize look-up tables within their mixing process. The reasons for using LUTs are varied but primarily seem to center around the ability to increase the variance of **"static values"** used within the fundamental mixing operation.

In the mix process defined above there are two static values 2 and 3, being used - if one where to utilize a LUT of values, then perhaps as an example a different value can be used on each round, rather than just the values 2 and 3 for each and every round.

When LUTs are employed in a hash function they are typically used in a combination of one or more of the following ways:

- Lookup per round index
- Lookup per internal state
- Lookup per current message

Given a LUT named P, the various ways P can be used to generate the next state of the hash via the mixing process based on the above list of possible operations are as follows:

$$Option\ 1 : H_{i+1} = (P[round_i \bmod |P|] * H_i) \oplus (P[round_i \bmod |P|] * Block_i)$$

$$Option\ 2 : H_{i+1} = P[H_i \bmod |P|] * Block_i$$

$$Option\ 3 : H_{i+1} = P[Block_i \bmod |P|] * H_i$$

$$Option\ 4 : H_{i+1} = (P[Block_i \bmod |P|] * H_i) \oplus (P[H_i \bmod |P|] * Block_i)$$

Where: $\mathbf{H_i}$, $\mathbf{H_{i+1}}$ are the current and next internal states respectively, $\mathbf{round_i}$ is the index of the current mixing round, $\mathbf{Block_i}$ is the block of the message in the current mixing round and **|P|** is the size of the lookup table.

As far as the composition of the LUTs are concerned, the types of values (numbers) used are typically implementation defined. Sometimes they may be a list of prime numbers, other times they may be a set of values that possess specific bit patterns *(eg: 0xA5A5A5A5 etc)*, or they may simply just be a list of randomly selected numbers *(**Nothing up my sleeve numbers**)* .

In the design of cryptographic hash functions and ciphers, the construction of S-Boxes and P-Boxes have become a well studied area. If one is looking to implement a general purpose hash function based on utilizing a list of values during the mixing process, a review of the literature associated with S-Box and P-Box design techniques would be highly recommendable.

The following are some general recommendations to consider when implementing a hash function:

- **Recommendation 1:** The "mix" operation does not need to be the same on each round. There could be multiple mix operations which are selected based on criteria such as the index of the current round, the value of the internal state etc.

- **Recommendation 2:** When hashing large messages, one could break the message up into chunks and compute the hash of each chunk in **parallel** then aggregate the hashes using a mix operation, this will dramatically increase the throughput of the hash function when running upon architectures that support multiple cores. This method of hashing is known as a **Merkle-Tree** or a hash-tree.

- **Recommendation 3:** The operations present in the **Mix** should be carefully chosen, as more often than not the mixing process may result in lowering the entropy of the internal state to the point where the internal state does not change. As an example in the mix operation denoted above if the internal state reaches the value **ZERO** it will typically end-up returning zero.

- **Recommendation 4:** The mix operation should handle repeated values in the message block, without causing a *"flush effect* on the internal state. A typical scenario might be a message comprised of bytes with the value of zero.
  Question: how many consecutive zero bytes will it take to get the internal state of the hash function to become zero? The answer for a well designed hash function should be: **A-LOT**.

- **Recommendation 5:** When hashing a key derived-from or otherwise aliasing a pointer to memory that has been obtain from an allocator that returns aligned addresses, one will observe that the first 2 or 3 Least Significant Bits (LSBs) of the key will **always** be zero, or in other words will have **zero entropy**. This is due to the fact that the pointers will be storing addresses that are multiples of either 4 or 8 depending on the machine's addressing granularity. The following are a couple of techniques to resolve the issue of low-entropy LSBs:
  - Fill in the LSBs with bits from their higher order neighbours: `pointer |= (pointer >> 3) & 0x03`
  - Fill in the LSBs with bits from a LUT: `pointer |= lut[round_i % lut_size] & 0x03`

Hash Functions and Prime Numbers

Generally speaking there is a rich tapestry of fact and fiction when it comes to the use of prime numbers in hash functions. Prime numbers are typically used in two areas regarding hash function implementation, they are:

- Hash value quantisation
- Mixing and entropy boosting

Hash Value Quantisation Using Prime Numbers

This area is by far the most reasonable and contains mathematically solid explanations for the use of prime-numbers under certain situations. The problems that arise here are typically related to the **Pigeonhole Principle.**

When using a hash function as part of a hash-table, one will want to quantize or in other words reduce the hash value to be within the range of the number of buckets in the hash-table. It is assumed that a good hash functions will map the message m within the given range in a uniform manner. The quantisation of the hash value is efficiently and simply obtained by computing the hash value modulus the table size N, as follows:

$$H \equiv h(m) \bmod N$$

The result of this operation will be a value H in the range R defined as [0, N).

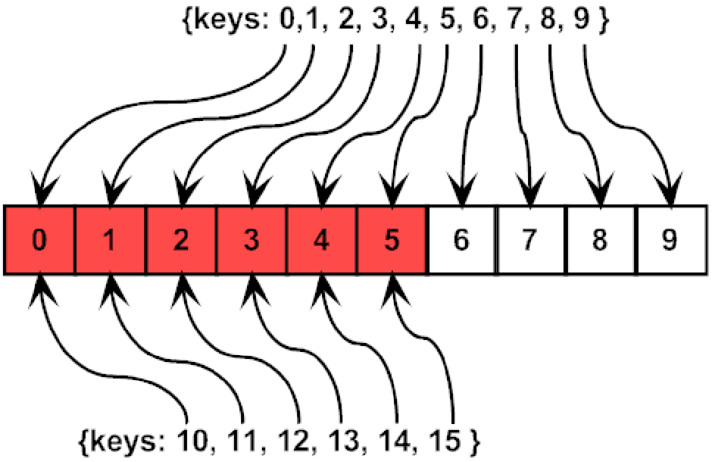
Problem 1 - Non-Uniform Mappings

Because the size of the table N will typically be smaller than the maximum value of the hash function H_{max} *(hence the requirement for quantization)* and H_{max} will most likely not be a multiple of N, nor will N likely be a factor or a multiple of a factor of H_{max}, the first **(H_{max} mod N)** values in the range will have a higher probability of having values hashed to them when compared to the remaining values in the range. So rather than having a value be mapped to any of those buckets with equal probability of 1/N, they will instead be mapped with a probability of 2/N, which is twice that of the other buckets. This will result in a higher load factor (collision rate) for the first (H_{max} mod N) buckets in the table.

As an example assume we have a good hash function h(x) that uniformly maps keys in the range [0,15] as H and a hash-table with only 10 buckets, we may simply quantize the hash values as follows:

$$H \equiv h(m) \bmod 10$$

The problem, assuming uniformly distributed keys in the range [0,15], is that the buckets [0,5] will have a higher probability of having values mapped to them when compared to the other buckets in the range [6,9].


{keys: 0,1, 2, 3, 4, 5, 6, 7, 8, 9 }
0
1
2
3
4
5
6
7
8
9
{keys: 10, 11, 12, 13, 14, 15 }

Simply put this particular problem can't be resolved by using prime numbers as the quantisation value. That being said there are a couple of things that can be done to resolve or mitigate this problem, they are as follows:

- Set the hash table size to be a factor of H_{max} (eg: A good basis would be 2^n)
- Increase the hash table size (where possible have the size tend closer to H_{max})

Note: This problem also appears when generating random numbers within a specified range, where the underlying random number generator generates values in a range that is larger than and is not a multiple of the desired range.

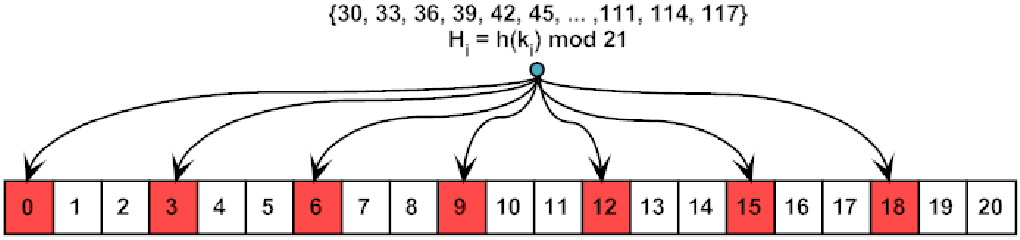
Problem 2 - Non-Uniformly Distributed Keys And Factors of N (Table Size)

N is the number of buckets in a hash table and as such is commonly used as the quantisation value. There is an issue which can arise if the keys being hashed are not uniformly distributed. Specifically when the keys result in values before quantisation that are factors or multiples of factors of N. In this situation buckets that aren't factors of N or multiples of factors of N will remain empty, causing the load factor of the other buckets to increase *disproportionately*.

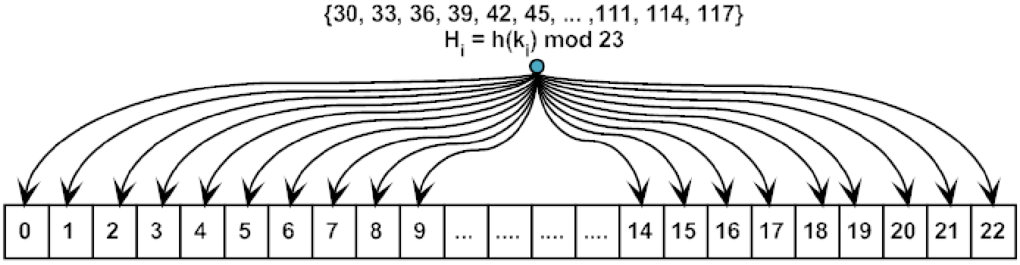
This situation seems to be the only valid reason to use a prime number as a quantisation value. Given that a prime number has only itself and one as factors, using a prime number to resolve this problem means even if the keys are not uniformly distributed and instead possess some of kind structure (specifically multiples of a value etc), the likelihood that those values will arbitrarily hash to either the value one or N (the prime number) will be vanishingly small.

Example: Lets assume we have a set of 30 messages **M_0, M_1, ..., M_{29}**, where we proceed to hash each message using a well behaved hash function **h(x)**. This gives us the values **H_0, H_1, ..., H_{29}**, which so happened to be in the form: **30, 33, 36, 39, 42, 45, ... ,111, 114, 117** - essentially all are multiples of three.

Furthermore we have a hash table with **21** (3x7) buckets or in other words a hash-space in the range [0,20]. We decide to quantise the values by computing their modulus by 21 like so: **H_i = h(M_i) mod 21**. By doing this we discover that the only buckets used to map the values denoted above are: **0, 3, 6, 9, 12, 15** and **18** - That is only 7 out of a total of 21 buckets or roughly 33% utilisation of the bucket-space and as a result incurred 21 collisions over the 30 values hashed.



Now if we had instead chosen a prime number, for example **23**, we would have utilized each and every one of the buckets in the table and would have only had 7 collisions - which would also be the theoretical minimum number of possible collisions given the table size *(bucket-space)*.



One further thing to consider would be that instead of a prime number we could have chosen a composite number that does not have three as one of its factors. But we would then also have to guarantee that *none* of the multiples of three, which we intend to quantise, are also not one of its factors - in short the problem becomes very hairy very quickly for little to no gain - Q.E.D

That being said, this particular solution has a slight overhead of its own - that is once a maximum table load factor has been reached the table will need to be re-sized. This is typically achieved by simply doubling the size of the current table, but one must also remember to snap to the next largest prime number as the new size, rather than simply doubling the current size.

In conclusion it would be far more productive and effective to mix the keys more thoroughly and rigorously than it would be to faff around with the ever changing quantisation parameter. Using something as simple as a power of two (2^n) will generally suffice, given a satisfactory mixing process and relatively well distributed keys **[Mitzenmacher et al. 2011]**.

Mixing And Entropy Boosting Using Prime Numbers

This area is where most of the *mythology* surrounding the use of prime numbers in hash functions reside. In implementing the **mixing** operation, one tries to define a process where by all the bits of the message block, equally affect all the bits of the internal state.

As an example using a well designed hash function, given two messages that differ by only one bit, the expectation is that the hash values will be starkly different *(aka having a large hamming distance)*. This property is derived from the Bit Independence Criterion (BIC) - cryptographic hash functions typically posses this trait, whereas general purpose hash functions don't necessarily need to have this trait (as it's largely an unrequired overhead), but instead attempt to attain something similar or close to BIC.

The use of prime numbers in the mixing function, is due to an assumption that fundamental mathematical operations (such as addition and multiplication) with prime numbers *"generally"* result in numbers who's bit biases are close to random.

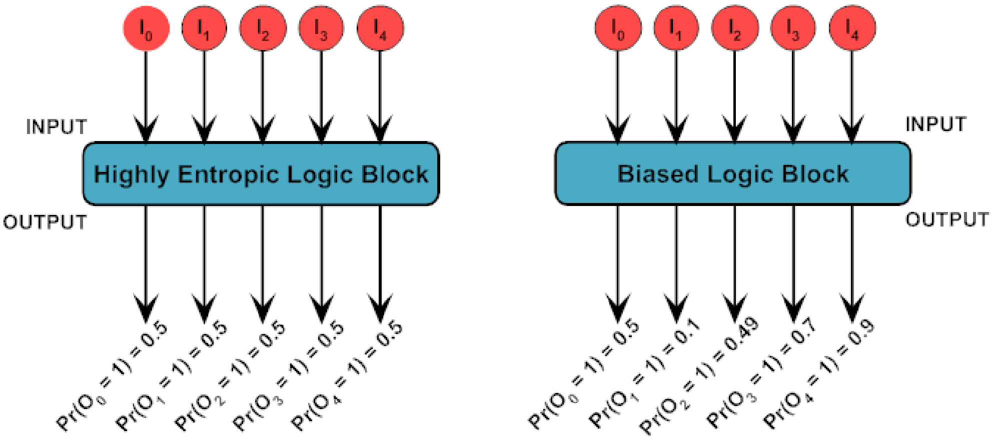
In simple terms the *"theory"* states that when you multiply *(or add)* a set of random numbers (aka keys) by a prime number the resulting numbers when as a group statistically analyzed at their bit levels should show no *(or very little)* bias towards being one state or another **ie: Pr(B_i = 1) ~= 0.5**

There is **no concrete proof** or other forms of evidence to support that this is the case or that it only (or more often) happens with prime numbers as opposed to composite numbers. It just seems to be an ongoing self-proclaimed intuition that some professionals in the field seem obliged to follow and preach.

Bit Biases

Bit sequence generators, be they purely random or in some way deterministic, will generate bits with a particular probability of either being one state or another - this probability is known as the **Bit Bias**. In the case of purely random generators the bit bias of any generated bit being high or low is always 50% (Pr = 0.5).

However in the case of pseudo random number generators (PRNG), the algorithm generating the bits will define the bit bias of the bits generated in the minimal output block of the generator.



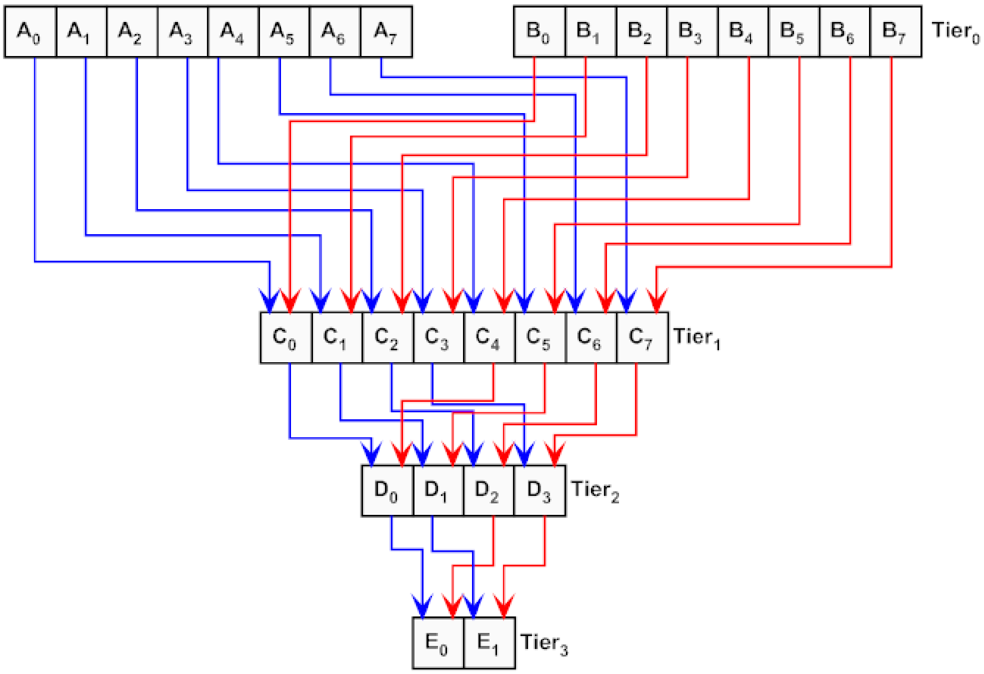
Example: Let's assume a PRNG that produces 8-bit blocks as its output. For some reason the MSB is always set to high, the bit bias then for the MSB will be a probability of 100% being set high. From this one concludes that even though there are 256 possible values that can be produced with this PRNG, values less than 128 will never be generated. Assuming for simplicity that the other bits being generated are purely random, then there is an equal chance that any value between 128 and 255 will be generated, however at the same time, there is 0% chance that a value less than 128 will be produced.

All PRNGs, be they the likes of hash functions, ciphers, m-sequences or anything else that produces a bit sequence will all exhibit some form of bit bias. Most PRNGs will attempt to converge their bit biases to an equality, stream ciphers are one example, whereas others will work best with a known yet unstable bit bias.

Mixing or scrambling of a bit sequence is one way of producing a common equality in the bit bias of a stream. Though one must be careful to ensure that by mixing they do not further diverge the bit biases. A form of mixing used in cryptography is known as avalanching, this is where a block of bits are mixed together sometimes using a substitution or permutation box (S-Box, P-Box), with another block to produce an output that will be used to mix with yet another block.

As displayed in the figure below the avalanching process begins with one or more pieces of binary data. Bits in the data are taken and operated upon (*usually some form of input sensitive bit reducing bitwise logic*) producing an ith-tier piece data. The process is then repeated on the ith-tier data to produce an i+1'th tier data where the number of bits in the current tier will be less than or equal to the number of bits in the previous tier.

The culmination of this repeated process will result in one bit whose value is said to be dependent upon all the bits from the original piece(s) of data. It should be noted that the figure below is a mere generalisation of the avalanching process and need not necessarily be the only form of the process.



In data communications that use block code based error correcting codes, it has been seen that to overcome burst errors, that is when there is a large amount of noise for a very short period of time in the carrier channel, if one were to bit-scramble whole code blocks with each other, then have the scrambled form transmitted and then descrambled at the other end that burst errors would then most likely be distributed almost evenly over the entire sequence of blocks transmitted allowing for a much higher chance of fully detecting and correcting all the incurred errors. This type of deterministic scrambling and descrambling without the need for a common key is known as interleaving and deinterleaving.

Various Forms Of Hashing

Hashing as a tool to associate one set or bulk of data with an identifier has many different forms of application in the real-world. Below are some of the more common uses of hash functions.

- ### String Hashing

  Used in the area of data storage access. Mainly within indexing of data and as a structural back end to associative containers*(ie: hash tables)*

- ### Cryptographic Hashing

  Used for data/user verification and authentication. A strong cryptographic hash function has the property of being very difficult to reverse the result of the hash and hence reproduce the original piece of data. Cryptographic hash functions are used to hash user's passwords and have the hash of the passwords stored on a system rather than having the password itself stored. Cryptographic hash functions are also seen as irreversible compression functions, being able to represent large quantities of data with a signal ID, they are useful in seeing whether or not the data has been tampered with, and can also be used as data one signs in order to prove authenticity of a document via other cryptographic means.

- ### Geometric Hashing

  This form of hashing is used in the field of computer vision for the detection of classified objects in arbitrary scenes.

  The process involves initially selecting a region or object of interest. From there using affine invariant feature detection algorithms such as the Harris corner detector (HCD), Scale-Invariant Feature Transform (SIFT) or Speeded-Up Robust Features (SURF), a set of affine features are extracted which are deemed to represent said object or region. This set is sometimes called a macro-feature or a constellation of features. Depending on the nature of the features detected and the type of object or region being classified it may still be possible to match two constellations of features even though there may be minor disparities (such as missing or outlier features) between the two sets. The constellations are then said to be the classified set of features.

  A hash value is computed from the constellation of features. This is typically done by initially defining a space where the hash values are intended to reside - the hash value in this case is a multidimensional value normalized for the defined space. Coupled with the process for computing the hash value another process that determines the distance between two hash values is needed - A distance measure is required rather than a deterministic equality operator due to the issue of possible disparities of the constellations that went into calculating the hash value. Also owing to the non-linear nature of such spaces the simple Euclidean distance metric is essentially ineffective, as a result the process of automatically determining a distance metric for a particular space has become an active field of research in academia.

  Typical examples of geometric hashing include the classification of various kinds of automobiles, for the purpose of re-detection in arbitrary scenes. The level of detection can be varied from just detecting a vehicle, to a particular model of vehicle, to a specific vehicle.

- ### Bloom Filters

  A **Bloom filter** allows for the *"state of existence"* of a large set of possible values to be represented with a much smaller piece of memory than the sum size of the values. In computer science this is known as a membership query and is a core concept in associative containers.

The Bloom filter achieves the storage efficiency through the use of multiple distinct hash functions and by also allowing the result of a membership query for the existence of a particular value to have a certain probability of error. The guarantee a Bloom filter provides is that for any membership query there will never be any false negatives, however there may be false positives. The false positive probability can be controlled by varying the size of the table used for the Bloom filter and also by varying the number of hash functions.

Subsequent research done in the area of hash functions and their use in bloom filters by Mitzenmacher et al. suggest that for most practical uses of such constructs, the entropy in the data being hashed contributes to the entropy of the hash functions, this further leads onto theoretical results that conclude an optimal bloom filter (one which provides the lowest false positive probability for a given table size or vice versa) providing a user defined false positive probability can be constructed with at most two distinct hash functions also known as *pairwise independent hash functions*, greatly increasing the efficiency of membership queries.

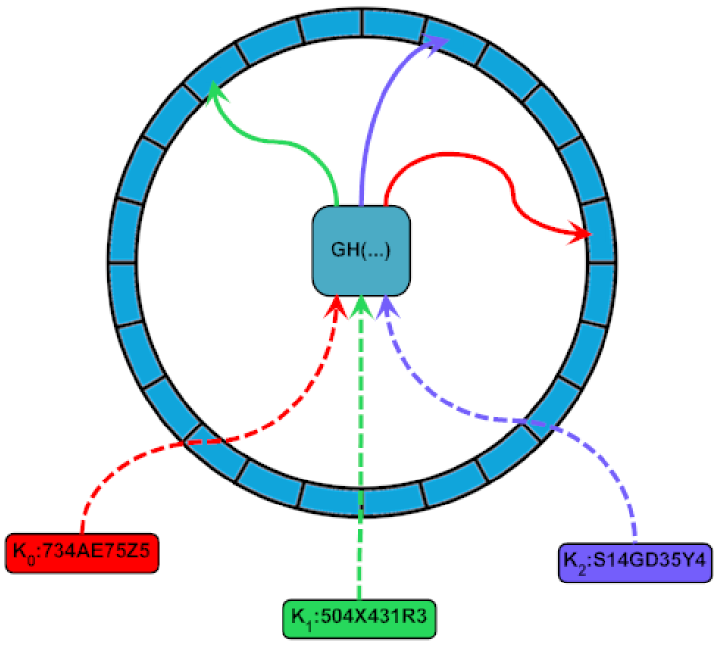
Bloom filters are commonly found in applications such as spell-checkers, string matching algorithms, network packet analysis tools and network/internet caches.

- ### Consistent Hashing

In **consistent hashing**, the hash space is broken into address ranges of roughly equal length. Each range is associated with one or more nodes, which hold values for keys that hash into that address range. By using a **Global Hash Function**, one can query the address space to determine the node(s) that will be holding the data corresponding to the key and then subsequently query the node directly for the required data.

Typically nodes in adjacent addresses will replicate the closer half of keys from their neighbours on both sides, and in some situations nodes may also replicate a randomly selected node in the ring (random graph approach). This replication is done to provide resilience against node failures and allow for load balancing and overall latency reductions when data that hashes to a particular node is highly sought after or queried by many clients. This data structure is the principle component of **Distributed Hash Tables (DHT)**.

In the diagram below, the blue segmented ring represents the address space for the range of values the hash function (GH) can produce. Each segment, which denotes an address range, will have one more nodes associated with it, and when a client with a key wants to look up the corresponding data, they first invoke the GH which will give them the address range, they then look up the node(s) handling that address range using a directory service (obtain node's IP address or routing ID etc), then proceed to directly query them using the key.



- ### Proof Of Work

Cryptographic hash functions that are pre-image resistant, can be used in specific ways to verify that a certain amount or type of work has been carried out, this is commonly known as **Proof Of Work**, and makes up the fundamental basis of such things as **crypto-currencies**, **challenge response hand-shakes** etc.

The way it typically works, is a known value K is given (some number of bits), then either an expected hash value or a hash value with a specific trait (eg: certain number of leading zero bits) is proposed.

The applicant *(aka entity performing the work)* will then go off and perform a computation to determine another piece data D, such that **H(K | D)** (the hash of the concatenation of K and D) will either be equal to the expected hash value or will posses the properties required to prove the work has been done. Furthermore because the verification of the hash value can be performed cheaply and efficiently and the work in computing the value D itself must be done in full *(as currently there are no known shortcuts - theoretic or otherwise)*, makes the use of such hash functions for **POW** style protocols an optimal choice.

Renaissance Technologies - Rentec

Available Hash Functions

The General Hash Functions Library has the following mix of additive and rotative general purpose string hashing algorithms. The following algorithms vary in usefulness and functionality and are mainly intended as an example for learning how hash functions operate and what they basically look like in code form.

- ### 00 - RS Hash Function

A simple hash function from Robert Sedgwicks Algorithms in C book. I've added some simple optimizations to the algorithm in order to speed up its hashing process.

```
unsigned int RSHash(const char* str, unsigned int length)
{
   unsigned int b    = 378551;
   unsigned int a    = 63689;
   unsigned int hash = 0;
   unsigned int i    = 0;

   for (i = 0; i < length; ++str, ++i)
   {
      hash = hash * a + (*str);
      a    = a * b;
   }

   return hash;
}
```

- ### 01 - JS Hash Function

A bitwise hash function written by Justin Sobel

```
unsigned int JSHash(const char* str, unsigned int length)
{
   unsigned int hash = 1315423911;
   unsigned int i    = 0;

   for (i = 0; i < length; ++str, ++i)
   {
      hash ^= ((hash << 5) + (*str) + (hash >> 2));
   }

   return hash;
}
```

- ### 02 - PJW Hash Function

This hash algorithm is based on work by Peter J. Weinberger of Renaissance Technologies. The book Compilers (Principles, Techniques and Tools) by Aho, Sethi and Ulman, recommends the use of hash functions that employ the hashing methodology found in this particular algorithm.

```
unsigned int PJWHash(const char* str, unsigned int length)
{
   const unsigned int BitsInUnsignedInt = (unsigned int)(sizeof(unsigned int) * 8);
   const unsigned int ThreeQuarters     = (unsigned int)((BitsInUnsignedInt  * 3) / 4);
   const unsigned int OneEighth         = (unsigned int)(BitsInUnsignedInt / 8);
   const unsigned int HighBits          =
                      (unsigned int)(0xFFFFFFFF) << (BitsInUnsignedInt - OneEighth);
   unsigned int hash = 0;
   unsigned int test = 0;
   unsigned int i    = 0;

   for (i = 0; i < length; ++str, ++i)
   {
      hash = (hash << OneEighth) + (*str);

      if ((test = hash & HighBits) != 0)
      {
         hash = (( hash ^ (test >> ThreeQuarters)) & (~HighBits));
      }
   }

   return hash;
}
```

- ### 03 - ELF Hash Function

Similar to the PJW Hash function, but tweaked for 32-bit processors. It is a widley used hash function on UNIX based systems.

```
unsigned int ELFHash(const char* str, unsigned int length)
{
   unsigned int hash = 0;
   unsigned int x    = 0;
   unsigned int i    = 0;

   for (i = 0; i < length; ++str, ++i)
   {
      hash = (hash << 4) + (*str);

      if ((x = hash & 0xF0000000L) != 0)
      {
         hash ^= (x >> 24);
      }

      hash &= ~x;
   }

   return hash;
}
```

04 - BKDR Hash Function

This hash function comes from Brian Kernighan and Dennis Ritchie's book "The C Programming Language". It is a simple hash function using a strange set of possible seeds which all constitute a pattern of 31....31...31 etc, it seems to be very similar to the DJB hash function.

```
unsigned int BKDRHash(const char* str, unsigned int length)
{
   unsigned int seed = 131; /* 31 131 1313 13131 131313 etc.. */
   unsigned int hash = 0;
   unsigned int i    = 0;

   for (i = 0; i < length; ++str, ++i)
   {
      hash = (hash * seed) + (*str);
   }

   return hash;
}
```

05 - SDBM Hash Function

This is the algorithm of choice which is used in the open source SDBM project. The hash function seems to have a good over-all distribution for many different data sets. It seems to work well in situations where there is a high variance in the MSBs of the elements in a data set.

```
unsigned int SDBMHash(const char* str, unsigned int length)
{
   unsigned int hash = 0;
   unsigned int i    = 0;

   for (i = 0; i < length; ++str, ++i)
   {
      hash = (*str) + (hash << 6) + (hash << 16) - hash;
   }

   return hash;
}
```

06 - DJB Hash Function

An algorithm produced by Professor Daniel J. Bernstein and shown first to the world on the usenet newsgroup comp.lang.c. It is one of the most efficient hash functions ever published.

```
unsigned int DJBHash(const char* str, unsigned int length)
{
   unsigned int hash = 5381;
   unsigned int i    = 0;

   for (i = 0; i < length; ++str, ++i)
   {
      hash = ((hash << 5) + hash) + (*str);
   }

   return hash;
}
```

07 - DEK Hash Function

An algorithm proposed by Donald E. Knuth in *The Art Of Computer Programming Volume 3*, under the topic of sorting and search chapter 6.4.

```
unsigned int DEKHash(const char* str, unsigned int length)
{
   unsigned int hash = len;
   unsigned int i    = 0;

   for (i = 0; i < length; ++str, ++i)
   {
      hash = ((hash << 5) ^ (hash >> 27)) ^ (*str);
   }

   return hash;
}
```

08 - AP Hash Function

An algorithm produced by me Arash Partow. I took ideas from all of the above hash functions making a hybrid rotative and additive hash function algorithm. There isn't any real mathematical analysis explaining why one should use this hash function instead of the others described above other than the fact that I tired to resemble the design as close as possible to a simple LFSR. An empirical result which demonstrated the distributive abilities of the hash algorithm was obtained using a hash-table with 100003 buckets, hashing **The Project Gutenberg Etext of Webster's Unabridged Dictionary**, the longest encountered chain length was 7, the average chain length was 2, the number of empty buckets was 4579.

```
unsigned int APHash(const char* str, unsigned int length)
{
   unsigned int hash = 0xAAAAAAAA;
   unsigned int i    = 0;
   for (i = 0; i < length; ++str, ++i)
   {
      hash ^= ((i & 1) == 0) ? (  (hash <<  7) ^ (*str) * (hash >> 3)) :
                               (~((hash << 11) + ((*str) ^ (hash >> 5))));
   }

   return hash;
}
```

Note: For uses where high throughput is a requirement for computing hashes using the algorithms described above, one should consider unrolling the internal loops and adjusting the hash value memory foot-print to be appropriate for the targeted architecture(s).

General Hash Function License

Free use of the General Hash Functions Algorithm Library available on this site is permitted under the guidelines and in accordance with the **MIT License.**

Compatibility

The General Hash Functions Algorithm Library C and C++ implementation is compatible with the following C & C++ compilers:

- GNU Compiler Collection (3.3.1-x+)
- Intel® C++ Compiler (8.x+)
- Clang/LLVM (1.x+)
- Microsoft Visual C++ Compiler (8.x+)

The General Hash Functions Algorithm Library Object Pascal and Pascal implementations are compatible with the following Object Pascal and Pascal compilers:

- Borland Delphi (1,2,3,4,5,6,7,8,2005,2006)
- Free Pascal Compiler (1.9.x)
- Borland Kylix (1,2,3)
- Borland Turbo Pascal (5,6,7)

The General Hash Functions Algorithm Library Java implementation is compatible with the following Java compilers:

- Sun Microsystems Javac (J2SE1.4+)
- GNU Java Compiler (GJC)
- IBM Java Compiler

Download

- General Hash Function Source Code (C)
- General Hash Function Source Code (C++)
- General Hash Function Source Code (Pascal & Object Pascal)
- General Hash Function Source Code (Java)
- General Hash Function Source Code (Ruby)
- General Hash Function Source Code (Python)
- General Hash Function Source Code (All Languages)
- Open Bloom Filter Source Code (C++)
- Bloom Filter Source Code (Object Pascal)
- Selecting a Hashing Algorithm (Bruce J. McKenzie, R. Harries, Timothy C. Bell)
- Cryptographic Hash Functions : A Survey (S. Bakhtiari, R. Safavi-Naini, J. Pieprzyk)

Renaissance Technologies - Rentec