FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Name Surname

Thesis title

Name of the department

Supervisor of the bachelor thesis: Supername Supersurname
Study programme: study programme

Prague YEAR

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

Dedication.

Title: Thesis title

Author: Name Surname

Department: Name of the department

Supervisor: Supername Supersurname, department

Abstract: Use the most precise, shortest sentences that state what problem the thesis addresses, how it is approached, pinpoint the exact result achieved, and describe the applications and significance of the results. Highlight anything novel that was discovered or improved by the thesis. Maximum length is 200 words, but try to fit into 120. Abstracts are often used for deciding if a reviewer will be suitable for the thesis; a well-written abstract thus increases the probability of getting a reviewer who will like the thesis.

Keywords: keyword, key phrase

Název práce: Název práce česky

Autor: Name Surname

Katedra: Název katedry česky

Vedoucí bakalářské práce: Supername Supersurname, department

Abstrakt: Abstrakt práce přeložte také do češtiny.

Klíčová slova: klíčová slova, klíčové fráze

Contents

Introduction

Advances in technology in the last decade led to huge accumulation of data. This have created need for algorithms able to work with large data in small memory. In our work we focus on the problem of finding the symmetric difference of two large sets, than differs only in a small number of elements.

Algorithms

The state of art to solution to such problem is to use a Invertible Bloom Lookup Table (IBLT). This data structure is very close to a hashtable. It differs mainly in the way how it deals with collisions. It just adds the values of the colliding keys and keeps the count of keys in each bucket. This means when some item is added to the IBLT and then removed, the IBLT will be same as IBLT where these two operations were not performed. This property is very useful in our case, because we can add all elements from the first set to the IBLT and then remove all elements from the second set. An only items in the difference had influence on the IBLT. We than may try to recover the difference from the IBLT, returning all values in buckets where only one key was hashed. There comes another crucial difference. IBLT is that we do not use one hash function but k. So, each key is stored in k buckets. Improving a probability a no two keys are hashed into same buckets. We may use such structure for finding the symmetric difference of two sets in linear space to the size of the difference. The authors NAME build up on the IBLT and proposed a new data structure improving the space consumed by a constant. Their improvement lies in replacing summation of values in the buckets with a XOR operation and not keeping the count of keys in the buckets. This may lead to recovery of keys not in the difference, but the authors successfully showed such event does not happen with high probability.

Both structures success depend heavily on the probability of existence of k-cores in random k-uniform hypergraphs depending on ratio number of edges and vertices. This comes from the fact, we may look on buckets as vertices and hashes of keys as edges. The ratio number of vertices to number of edges for which no 2-core exists in random k-uniform hypergraph with high probability is know for any k and the highest possible is 1:0.8 for $k = 3$. However, improvements may be gained by on non uniform hypergraphs as shown in the work WORK.

We implemented the IBLT and compared with a improved version of such algorithm by the authors NAMES. We tested the algorithms for different familes of hash function. As we expected the common usecase of our implementation is

Finding symmetric difference of two sets may have many applications from keeping instances of distributed databases in sync to comparing closely related genomes.

1 Preliminaries

1.1 Randomized Algoritms

We are going to need some tools to work with mean and variance. The simplest is:

Theorem 1. *(**Markov Inequality**) Given a nonnegative random variable X with expectation*

$$Pr[X > kE[X]] \leq \frac{1}{k}$$

In simplicity, this theorem says that only $\frac{1}{k}$ people can be k-times richer than the mean.

Definition 1. *(**Variance**) Let X be random variable, than*

$$Var[X] := E[(E[X] - X)^2]$$

.

One can, with little work, get a Chebichev Inequality from Markov Inequality.

Theorem 2. *(**Chebyshev inequality**) Given a nonnegative random variable X with expectation*

$$Pr[|X - E[X]| > kVar \leq \frac{Var[X]}{k^2}$$

Both previous theorems work for all random variables. We may get a better bound by requiring more from the random variables. We will require that they be independent and have a Bernoulli distribution.

Theorem 3. *(**Multiplicative Chernoff bound**) Given independant Bernoulli random variable $X_1, X_2 \dots X_n$ such that $X = \sum_{n=1}^{n} X$ and $E[X] = \mu$, then for $0 \leq \beta \leq 1$*

$$Pr[X \leq (1 - \beta)\mu] \leq \exp\left(\frac{-\beta^2\mu}{2}\right)$$

We have two main types of randomized algorithms - Las Vegas and Monte Carlo. Las Vegas algorithms are guaranteed to have correct answers, but they are not guaranteed to end. Monte Carlo does not always return the correct answer, but the probability of failure is low. Often, the value is set not to be greater $\frac{1}{3}$. A correct answer does not have to be exactly the right value; sometimes, we allow some amount of error and answers close enough, we count as success.

The number $\frac{1}{3}$ may seem arbitary. It is. We will show that any algorithm with a probability of success is more than $\frac{1}{2}$. Can be modified to increase its probability of success to any value smaller than 1. This trick is often called a Median trick.

Median Trick

Let's say we have some randomized algorithm A with a probability of success p. Now we run k instances of A. We then pick a median of all results. This approach fails when the median instance is a failure. This happens only if more than $\frac{k}{2}$ instances were failures.

We can now use Multiplicative Chernoff bound to determine how many instances of A we have to run to get some desired success rate of p'. We set X as the number of successful instances. We need $(1-\beta)\mu$ to equal $\frac{n}{2}$.

$$(1-\beta)\mu = (1-\beta)np = \frac{n}{2}$$

.

$$\beta = \frac{2p+1}{2p}$$

Then we get:

$$Pr[X \leq \frac{n}{2}] \leq \exp\left(\frac{-(2p+1)pn}{2}\right) = 1 - p'$$

This means that the probability of failure lowers exponentially with the number of instances run. So, when we want a error rate δ, than we need to run $\mathcal{O}(\log(\frac{1}{\delta}))$ instances.

This is a very useful trick as it allows us to strive for randomized algorithms with low success rates and we are punished just by logarithmic slowdown.

Some algorithms have a better probability of succeeding and the larger the data they work on are. If we can prove that the limit in the infinity of such probability is one, we say that the algorithm succeeds with high probability. An abbreviation "whp" is often used.

1.2 Hypergraphs

A graph is a tuple of two sets. One set contains vertices and one edges. Normally, edges may connect only two vertices. Formally edges are sets containing exactly two vertices, but what if we allowed edges to contain any number of vertices?

1.2.1 Basics

This leads to hypergraphs:

Definition 2. *Let V be a set and $E \subseteq P(V)$, then the tuple (V, E) is called a hypergraph. Elements of V are called vertices, and aspects of E are called edges.*

One should easily see that every graph is also a hypergraph. The other way is clearly not true. Some terms are connected to graphs, which would be useful to state also for hypergraphs. Definitions are going to be very similar to their graph equivalents. We begin with the subgraph:

Definition 3. *Let $G := (V, E)$ be a hypergraph, then $G' := (V', E')$ is a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$, and we may write that as $G' \subseteq G$.*

Then continue with degree:

Definition 4. *Let G be some hypergraph and its G' subgraph, then G' is a k-core if every vertex of G' has a degree of at least k and G' is the largest such subgraph of G'.*

Sometimes, we would like to have only the same size edges. This concept is called k-uniformity.

Definition 5. *Let $G := (V, E)$ be than a hypergraph, than G is k-uniform, if $\forall e \in E : |e| = k$. Then, we define degree in hypergraph*

Definition 6. *Let $G := (V, E)$ be than a hypergraph, and be v some vertice from V than $\Delta(v) = |e \in E; v \in e|$. We call this property degree of vertice.*

1.2.2 Paring in Random Hypergraphs

Imagine being a kindergarten teacher. Your goal is to ensure that children have enough toys to play with. There are a few problems with that. Children are quite picky, and everyone is willing to play with exactly 3 toys. Children do not understand that sharing is caring. Thus, every child needs at least one toy with which they are willing to play. Children pick toys they are willing to play with randomly, but you assign the toys to the children. Lastly, you are limited in budget, and the fewer toys you buy, the better.

We can translate this problem to hypergraphs. Toys are vertices, and children's possible choices of toys are edges. We now name this hypergraph G and let $G = (V, E)$. This hypergraph is actually 3-universal. Now we define a new term:

Definition 7. ***k-core*** *is the largest subgraph H of graph G such that all vertices have a degree of at least k.*

One should see that if G contains a 2-core hypergraph, pairing is not possible. The other way is to get it from Hall's Theorem. It may not be clear what exactly means, "Children pick toys they are willing to play with randomly." By randomly, we mean they uniformly choose from the set of subsets of V with size 3.

We now may call a result from [https://arxiv.org/pdf/1101.2245]. This says that random 3-uniform hypergraphs with n vertices a and m edges do not contain a 2-core with high probability if $\frac{n}{m} \geq 1.222\ldots$. This solution is tight, thus when the ratio is less than $1.222\ldots$, than 2-core is found with high probabilty.

This gives a solution to our problem. This result shows we need at least a 1.22 toy for each kid. This, however, does help us as it is unclear whether we can find such a pairing quickly. There is a simple hungry algorithm:

```
Step G : (V,E) -> (V,E)
    G = (V,E)
    ToBeRemoved <- { e in E;
            (exists v in  e) (degree of v is 1)
            }
    return (V, E - ToBeRemoved)

Run G : (V,E) -> False or True
    G = (V,E)
    (V, New ) <- Step G
```

```
E = New return False
E = {} return True
return Run (V,New)
```

Hungry algorithms have a bad tendency not to work. However, it works for this problem. If some vertices lie in only one edge, we can pair these two together and remove such edges. If there exists none such vertice, then we found $2 - core$.

1.3 Hashing

1.3.1 Problem of Dictionary

The first time one meets with hashing functions is when faced with the need for a data structure with an $O(1)$ time cost for insert and get by key. Suppose the keys are from $[n]$, where the number of inserts $O(n)$, then the solution is simple. We can create an array of length n and index into it.

This solution, however, depends on the size of the universe. If the universe U has a size of M, then we need to use a table of size M. This may and often is not feasible. For example, there are 2^{32} 32-bit integers. The main solution to this problem is to have some "good" hashing function $f : U \to [n]$, where n is some desired table size. What means "good"? When we insert some item a with key k in the table, we insert at $f(k)$ index. When there is already value in the table, we get a hashing conflict. There are many possible solutions to this. The simplest solution is to keep more values in the same field.

1.3.2 Hashing Function Families

By "good," we mean that it minimizes hashing conflicts. There are a few properties that may be useful in determining such "goodness." Before we look at them, we need to address one problem. If we choose a hashing function deterministically, the enemy may choose data hashing with very few values. We may prevent this by picking a hashing function randomly from some hashing function family.

Definition 8. *A family F of hashing functions from $X \to Y$, where $|Y| = n$, is* ***c-universal*** *if*

$$\forall(x, y \in X)\left(P\left(f(x) = f(y)\right) \leq \frac{c}{n}\right),$$

where f is chosen uniformly randomly from F and $x \neq y$.

Universality guarantees, that hashes are more or less evenly distributed. This is a useful property. As it allows us to state, that the expected number of items with the same hash is $E(\frac{c}{n})$.

One problem with universality is that it allows for a hashing function family where all hashing functions hash from 1 to 0. This is prevented by being a hashing function family (c-k)independent. This property is sometimes called strong universality in the literature.

Definition 9. *A family F of hash functions from $X \to Y$, where $|Y| = n$, is* ***(c,k)-independent*** *if*

$$\forall(x_1, \ldots, x_k \in X, y_1, \ldots, y_k \in Y)\left(P\left(f(x_1) = y_1 \wedge \cdots \wedge f(x_k) = y_k\right) \leq \frac{c}{n^k}\right),$$

1.3.3 Examples of Hash Function Families

Linear Congurence

One of the simplest hash functions is linear congruence. This function is defined as:

Definition 10. *Let p be a prime number, n be from* $\mathbf{N}$*, a a number from $[n-1]$, and b a number from $[p]$. Then the hashing function is defined as:*

$$f_{a,b}(x) = (ax + b) \mod n \mod p$$

Let H_p^n be set of all such functions with prime p and size n , than we H_p^n a linear congruence family.

Linear congruence family is also useful in practice. Dictionaries in C# are actually implemented using a linear congruence family. The main advantage of this family is that if we choose the size of the universe we are hashing to, then we get a function from a linear congruence family practically for free. The linear congruence family is $(2, 4)$-independent. [https://mj.ucw.cz/vyuka/dsnotes/06-hash.pdf]

Multitplyshift

Multiply shift family is another often used hash function family. For example murmurhash is based of it. It has one advantage over linear congruence. It being, that it does use only logical shift and

2 Streaming Algorithms

Definition 11.

Theorem 4. *Let f be a function whose derivative exists at every point, then f is a continuous function.*

2.1 Introduction to Streaming algorithms

2.1.1 Motivation

In the early days of computing, memory was a significant hurdle. It was both limited and expensive. However, this has changed dramatically. Today, we can store vast amounts of data inexpensively. As a result, people often opt to add more storage rather than spend time deleting unnecessary files.

This environment has led to a massive accumulation of data, sometimes exceeding what local machines can handle. One example might be some network router that processes large quantities of data. This is a prime example of a problem that requires the use of a streaming algorithm. The router gets some data, then performs some operations with them, and lastly, it flushes them and never sees them ever again. This may be useful for finding heavy hitters and thus deciding which IP addresses to keep in the cache.

Another challenge is limited throughput. Imagine having two distant instances of a massive database; it would be ideal for synchronizing them. However, sending all the data over the network to check if they are the same isn't always feasible due to bandwidth constraints. We would like to have an algorithm minimizing such costs.

At first glance, these two problems do not seem very similar. However, the opposite is true. When we have a good algorithm for processing large data in small memory, we can also use it to solve the second example. May we have two databases, S_1 and S_2. We would like to perform operation o on $S_1 \cup S_2$. We have access to some algorithm A that can do o with memory of M and A does not care about the order in which it consumes data, and it does not expect to see data again. Then we may first do operation o on instance S_1, save the state of the algorithm, send it to instance S_2, and restart the process. We had to use at most $\mathcal{O}(M)$ bits of information, to ship the content of working memory to the other database.

The way an internet router accesses data may be formalized by the concept of Data Stream. We may imagine the Data Stream as an oracle holding internally some sequence of words, which are inputs to our program. We may ask the oracle to provide us with the next item in the sequence. This is the only way to access the input data, and there is no way back. So if we get the k-th item from the sequence, there is no way to see the $(k-i)$-th item again for some natural i. More formally:

Definition 12. *Data Stream is tuple $(A, i, Access)$, where A is a sequence of unknown length N of finite words from universe U over finite language L. i is counter, which may only increase in value. $Access()$ returns value of A_i and increases value of counter by one.*

This may not seem to be limiting as we might create a large array and copy elements from the Data Stream to it and then continue with normal computation as in RAM. This leads us to the second limitation, limiting memory usage.

We would like to be memory-bound. When designing algorithms, we will choose those with polylogarithmic space complexity over the size of the universe and the number of items in the Data Stream.

In computer science, Oh notation is often used to describe algorithms' time or space complexity. While this simplification is convenient and helps us to compare algorithms easily, it may lead to wrong conclusions as algorithms with worse Oh complexity may perform better in the real world than algorithms with better Oh complexity.

One reason may be that our theoretical model of computation RAM (Random Access Machine) hides some constants from us. If our only need is to find asymptotic complexity, then hiding constants is not an issue, but in the real world, these hidden costs may cause significant performance problems. For example, random access to memory is much more expensive, than serial read.

Consequently, algorithms with the same time complexity can differ significantly in performance, sometimes by orders of magnitude. For example, IBLT (Invertible Bloom Lookup table) outperforms the naive algorithm for finding the symmetric difference of two large sets by using memory linear to the size of the symmetric difference. In contrast, the naive algorithm requires linear memory to the size of both sets combined.

As such, we would like to have a model that is stricter about memory and how we can use it. The main intuition should be, that we can see every piece of data just once and never again.

2.1.2 Data Stream model

We may imagine the Data Stream as an oracle holding internally some sequence of words, which are inputs to our program. We may ask the oracle to provide us with the next item in the sequence. This is the only way to access the input data, and there is no way back. So if we get the k-th item from the sequence, there is no way to see the $(k-i)$-th item again for some natural i. More formally:

2.1.3 Use cases for Streaming Algorithms

At first glance, it is unclear how strong Streaming Algorithms are and for what purposes they may be used. In this section, we give some examples of such problems and Streaming Algorithms that solve them:

Counting

As humans, we are often used to count on our fingers. The basic strategy of raising one finger for every increase leads to the highest count of 10. This may be significantly improved by using fingers as bits and counting in binary. We can get at most $2^{10}-1$. Is there any upper bound on the maximum? One should see that for describing 2^{10} states, we need at least 10 bits. If we decided to count only even numbers, we could go higher. This has one little problem, it is not clear how to add one to our count. The solution is to abandon the deterministic algorithm

and go for a probabilistic one. We will use s for the true sum and s' for the sum returned algorithm.

```
Update c : N -> N
    let s = uniform pick from [0,1]
    s > 0.5 return c + 1
    s <= 0.5 return c

Query s : N -> N
    return s * 2
```

We can use *Update* when we want to increase the count by one. It is not hard to see that the expected value is the same as the true value. More specifically, s' is going to have a binomial distribution. Conversely, it is unclear how close the algorithm returns sums s. We will accept all s' in $U(s, \epsilon s)$ as successes. We will use a Chebychev inequality to set bounds on probability. We have to know the variance of random variables to use this inequality. Luckily, it is common knowledge, that binomial distribution has a variance with value $np(p-1)$. Now we just input the value to the Chebychev inequality:

$$Pr[|s' - s| > \epsilon s] = Pr[|X - pn| > \epsilon np] \leq \frac{np(p-1)}{(\epsilon np)^2} = \frac{(1-p)}{p\epsilon^2 n}$$

So, we can see that the probability of failure decreases as δ or n gets larger. The latter is important as it implies that the algorithm succeeds with high probability. When we set parameters δ and n, so the probability of failure is less than $\frac{1}{2}$, we may use a median trick to get a failure rate of δ, by running $\mathcal{O}(\log(\frac{1}{\delta})$ instances of the algorithm.

This algorithm may be usable, but it allows us to double a range; we can count on this being probably not worthwhile. Getting the same result would cost us only one bit of memory. So, running multiple instances of this algorithm would essentially be worthless. Also, we are losing accuracy and determinacy, which are great costs.

Moris Counter

We may modify this algorithm to multiply the value exponentially. We change the meaning of increasing the counter by one from "add two" to "multiply by two." We also change the probability of increasing the counter appropriately.

The algorithm we get is called a Moris Counter. It is regarded as the first true Streaming Algorithm ever discovered.

```
Update c : N -> N
    let s = uniform pick from [0,1]
    s > 2 to power of -c return c + 1
    s <= 2 to power of -c return c

Query s : N -> N
    return 2 to power of s
```

Again, we say s is the real sum and s' sum returned by our algorithm. It does not take a lot of work to prove that: $E[s'] = s$ and that $\frac{n^2}{4}$. We are going to skip it. However, interested readers may find proof here [Small Summaries for Big Data].

Again, we could use Chebichev inequality to get bound on the probability of success.

$$Pr[|s' - s| > \epsilon s] = Pr[|X - pn| > \epsilon np] \leq \frac{n^2}{4}(\epsilon np)^2 = \frac{1}{\epsilon^2}$$

There one should see, that for $\epsilon < \sqrt{\frac{1}{2}}$, we may no longer use Median Tric. This has one simple solution, we may lower the probability of a counter increase. This means the Moris counter gets finer but at the cost of higher memory usage.

2.1.4 Number of distinct elements in MultiSet

We are going to move to a different problem. Imagine having a very large multiset where some elements may be multiple times, which is given to us as a Data Stream. Our goal is to find the number of different elements. In this subsection, we will not show just a sketch of the proof. We will omit some details.

We start by cheating. We assign each element a random number from some uniform distribution so that the same elements have the same number. We clearly do not have space for that, but we will solve this issue later.

Now, we find k different random numbers associated with some element (we expect the numbers to be linearly ordered). We are going to call this value v. Let n be the true number of distinct elements in the set, and R be the size of the universe from which we pick random numbers. Let v be the largest of the smallest numbers, then our prediction $n' := \frac{R(k-1)}{v_0}$

The hash function may be used to assign random numbers, and we can keep the k-smallest hashes in memory. Now, we should check how good our approximation is. We may see that some distinct values may have the same hashes. This is easily overcome, but we are not going to, and we just pretend all values have different hashes. We would also pretend that the hashes are independent of each other. (These are clearly contradictory, but again, we are not going to care).

We have two possible errors, one n' being too large. We first look at n' being too large. By too large, we mean that $n' > (1 + \epsilon)n$. This means at least k values had their hash with value lover than some threshold $T = T(\epsilon, n, R)$. We should see that increasing k decreases the probability of such an event happening.

The second is that the value is too low, so this means all hashes of values fall under threshold $F = F(\epsilon, n, R)$. Whether a hash is under or above some threshold is just a random variable of Bernolli distribution, and all the hashes are "independent" as we know the variance, and we may call a Chebychev inequality.

Here we end our sketch of proof. This algorithm may improved, which leads to HyperLogLog, which is actually used.

2.2 Statement of the Problem

We will return to the example from the introduction. Imagine we have two instances of one database, and we would like to find if they differ, and if they do, then make corrections so they become the same. Also, we would like to exchange as little data between the instances as possible.

We can model this problem by having two large sets A, B such that their symmetric difference is slight (the symmetric difference of A, B contains elements

that are just in one of the sets). The elements of A and B are given to us as a Data Stream, first we get elements from A, then from B.

Definition 13. *k-SetRecoveryProblem of sets A and B is the problem of finding the symmetric difference of A, B of size at most k, where elements of A and B are provided to us using a data stream, first giving elements of A, then sending a dividing symbol, and then sending all elements of B. If not said otherwise, we would use N for the number of elements in both sets and M for the universe size from which items of the Data Stream are taken.*

We would expect that elements of both sets have some unique binary representation. If we talk about some arithmetic operations over these elements, we are actually doing these operations over their binary representations.

One may imagine that sets contain just integers that are then mapped to the actual elements.

We can solve any k-Problem in $\mathcal{O}(n)$ space and time, where n is the number of elements in the Data Stream. We will build a HashSet, adding all elements from A and then removing all elements from B. This solves our problem, but our space complexity is much larger than LogDSpace. Actually, the best-known algorithm has $\mathcal{O}(k)$ complexity, where k is the size of the symmetric difference.

2.3 From Simple solutions to IBLT's

It's not easy to see how to proceed to get an algorithm better than the naive version. In this chapter, we will try to build Invertible Bloom Lookup Tables that solve the problem of the aforementioned space complexity. The first question should be how?

2.3.1 Solving 1-Problem

Let's start by simplifying the problem. We would limit the size of the symmetric difference to size 1.

Lemma 5. 1-*The problem of two sets A, B is solvable by a Streaming Algorithm, and we will call this algorithm.*

The construction is quite simple; we will remember two numbers C and S:

$$C := |A| - |B|$$
$$S := \sum_{i \in A} i - \sum_{i \in B} i$$

It is trivial to see that there exists a Streaming Algorithm returning values C and S for any sets A and B.

If C is 1, then the symmetric difference is $\{S\}$. If C is -1, then the symmetric difference is $\{-S\}$, if $C = 0$, then the symmetric difference is empty.

It should be clear that C may be just -1, 0, or 1.

This algorithm may not be the best in terms of memory efficiency, as with S, C may consume $\mathcal{O}(\log(N) + \log(M))$ memory, but this may be improved to

$\mathcal{O}(\log(M))$. We leave this as an exercise to the reader. Hint: Can C be from a finite field?

We are going to reuse this algorithm in the next section; thus, it would be nice to give it a name.

Definition 14. *SumOneFinder is a Streaming Algorithm solving* 1*-Problem using sum and counter.*

2.3.2 From 1 to k-SetRecoveryProblem Solution

We now can use SumOneFinder to find a solution to 2-SetRecoveryProblem. The sets may differ in up to two elements. For simplicity, we assume that they differ in exactly two elements a and b and $A \subset B$. Firstly, we take some hash function $f : A \cup B \to \{0, 1\}$. We can use it to divide all elements into two buckets. If we were lucky, we got a and b in different buckets, then we need to solve just two 1-SetRecoveryProblems. If we were not lucky, the two elements would have been hashed into the same bucket, and we would have no way to recover them. We only know that there are two elements in such buckets as we keep a count. Sadly, the probability of failure is $\frac{1}{2}$, so we cannot use a median trick; however, if we divide the elements into more parts, the probability of a and b increases.

We can generalize to get a general solution to k-SetRecoveryProblem. For every k we just need to find some l of number parts. Such as the probability of every element holding exactly one element is more than $\frac{1}{2}$. When we have k elements and l parts, the probability of no two elements hashing to the same part is:

We can see that our problem is equivalent to the Birthday problem. It is known that large k l needs to be in $\mathcal{O}(k^2)$. [https://www.physics.harvard.edu/files/sol46.pdf] This means we got a solution to our problem in $\mathcal{O}(log(\frac{1}{\delta})k^2)$ with a probability of success of $\frac{1}{\delta}$. We remind you that the logarithm is gained from using a Median trick. We are going to call this algorithm without using Median tric as KSumOneFinder.

This solution has two problems: it is quadratic space complex compared to the size of the symmetric difference, and it has a nonzero probability of failure.

The main idea is that when we run multiple instances of KSumOneFinder, we may use the results from each to improve the others. Let's say we have two instances of KSumOneFinder G_1 and G_2, and both algorithms fail to recover k elements, but they recover two sets of elements K_1 and K_2. We then may take K_1 and remove all these elements from G_2. If we were lucky, some unrecovered elements in G_2 lay in the same buckets as those in K_2, and then we can recover those. When we recover all elements or none, we may end; if not, then we may use elements in K_2 to recover more elements from G_1.

```
SymDifference A B : (A,B) -> C
    G1 <- Encode A B
    G2 <- Encode A B
    K1 <- Recover G1
    K2 <- Recover G2
    let Recovered = K1 union K2
    while true:
        G1 <- Recover (G1 - Recovered) union Recovered
        G2 <- Recover (G2 - Recovered) union Recovered
        if no element was newly recovered return recovered
```

Before we analyze this algorithm, we first show some intuition. To recover an element we need to pair it with some bucket. Every element lies in m buckets, where m is the number of instances run. As such, we may look at buckets as vertices and elements as edges in some m-uniform hypergraphs. We are not able to recover some elements if some 2-core exists. We made a little change to our algorithm; we let our instances of KSumOneFinder share one single set of buckets.

Here, we may call a theorem from preliminaries about the threshold for the appearance of 2-core in 3-uniform hypergraph. As such, we get a high probability that we can recover all elements from the symmetric difference of the two sets A and B if the number of buckets to the size of the symmetric difference is approximately at least $1.222\ldots$. This is the best we can get for any k-uniform hypergraph. We do not need to limit ourselves to uniform hypergraphs, as better bounds were found for non-uniform ones. Michael Rink's experimental result shows [https://arxiv.org/pdf/1204.2131] that by using edges of size $(3, 21)$, it is possible to get a bound of 1.086. It should be noted, that the best possible result is using this method 1 as every element needs to paired with some bucket.

2.3.3 Invertible Bloom Lookup Table

The algorithm is equivalent to a known data structure called an IBLT (Invertible Bloom Lookup Table). This algorithm is described here[https://arxiv.org/pdf/1101.2245] This structure may also work as a dictionary, supporting recovery key-value pairs. There is no use case for this for us; we will describe only a simplified version. We describe the structure now. Let U be the universe of possible keys. Keys must be a binary representation; we will work just with it and finally be B, the sets of buckets. We start by picking a set of random hash functions $U \to B$ named H and some helper hash function D.

The buckets are very similar to SumOneFinder:

```
type Bucket:
    SumOfKeys
    SumOfHashes
    Count
```

This is practically the same as the fields of SumOneFinder, but we added a new field called SumOfHashes. This field will be important for allowing the removal of keys not present in the Sketch.

Now, we describe three possible operations that are done with the IBLT data structure.

```
Add Key : (Value) -> void
    for every hash function h in H:
        bucket <- (h Key)
        bucket.SumOfKeys +<- Key
        bucket.SumOfHashes +<- D Key
        bucker.Count +<- 1

Remove Key : (Value) -> void
    for every hash function h in H:
        bucket <- (h Key)
        bucket.SumOfKeys -<- Key
        bucket.SumOfHashes -<- D Key
        bucker.Count -<- 1
```

```
IsPure bucket: Bucket -> bool
    if bucket.Count is not  1 or -1 return false
    if bucket.SumOfHashes * bucket.Count is not equal to
        (D bucket.SumOfValues * bucket.Count) return  false
    return true
Recovery: -> Values
    until there is some bucket, that is pure:
        if bucket.Count = 1 Remove bucket.SumOfKeys
        if bucket.Count = -1 Add bucket.SumOfKeys
    if some bucket contains non zero value recovery failed
```

Why do we have the SumOfHashes? The problem is that the count does not need to correspond to the actual number of elements hashed to such a bucket. It may be significantly lower, but without the helper hashing function D, we would not know, and we could recover items that are not present in the symmetric difference. By using the SumOfHashes, we lower the probability of such an event happening by size the size of the universe D is hashing to.

2.4 HPW algorithm

IBLT needs 1.222... buckets with three main hash functions and one helper function. We are not going to improve that. However, the size of the bucket can be improved. If we take an IBLT bucket, we see that it has three fields. In a trivial implementation using, for example, a 32-bit integer for them, we would actually need three times the memory. This may be improved as Count and SumOfHashes may have smaller sizes. We will not dive deep into these methods as they are out of scope. There is actually an algorithm needing just one field per bucket. This algorithm was discovered by Jakob Bæk Tejs Houen, Rasmus Pagh, Stefan Walzer. [https://arxiv.org/pdf/2211.03683] As the authors did not name their algorithm for the purpose of this text, we are going to use the abbreviation HPW.

The main idea is simple: instead of the sum of values and counts, we will use the xor of values hashed to the bucket. The interesting property of xor is that it is computative, and $aoplusa = 0$. This means that if some element a is added to both sets A and B, the corresponding Sketch will stay the same.

```
type Bucket:
    Value

Toggle Value : (Value) -> void
    for every hash function h in H:
        bucket <- (h Key)
        bucket.Value xor<- Key

LooksPure bucket: Bucket -> bool
    value <- bucket.Value
    if key = null return false
    for hash function h in H:
        if (h value) = bucket return true
    return false

Recovery: -> Values
    pure <- { b in B; LooksPure(b)}
    recovered <- {}
```

```
while true:
    pureNext <-{}
    for b in pure if LooksPure:
        if recovered.Contains(b.Value) remove b from recovered
        else remove add else to recovered
        Toggle b.Value
        for h in H:
            add (h b.Value) pureNext
        pure <- pureNext
        pureNext <-  {}
    if pure = {} return success if all bucket.Value are null
```

We will not explain why the algorithm works in depth, but we will try to give some insight into how it might work. We start by imagining a sunshine world, where LookPure returns true only, when a value in the bucket corresponds to some element of symmetric difference. In this world, when our algorithm succeeds it returns all the elements in symmetric difference and not one more. Sadly, not all dreams may be fulfilled, and we must face the problem of LooksPure not returning a correct answer. It may happen that some xor x of some set of values K that hashes to one same bucket looks pure. If this happened during recovery, then the value x would be recovered values despite not being an element of symmetric difference. This does not mean that it stays there; it can be removed and probably will as the value was toggled back to the sketch. We are going to call such keys outside keys. What may also happen is that some set of outside keys cancel themselves out, and then we have no chance of removing them. It is possible to show that these issues are unlikely to cause problems. As such, this HPW returns a high-probability right answer if the number of buckets is large enough.

3 Dotnet and C# Preliminaries

We chose C# as the main programming language. This may seem like a weird choice, as many consider it a slow language. Why not C++, C, or Zig? C# is probably a little slower than compiled languages due to being a jitted and garbage collected. This is, however little price for benefits we get. We do not need to worry about freeing memory, and users do not need to compile the source code for their machines.

One main strength of C# is that the compiler was rewritten with the idea of the compiler-as-a-service (CaaS) in mind. Originally, the compiler was written in C++, but later, Microsoft decided to rewrite it to C#. Now, developers may use the compiler from the code itself. This means compiling code during runtime and then using the newly compiled binaries is possible. Such a possibility is especially useful when working with data unknown during compile-time. Another example is the possibility of writing custom compilation errors. The new compiler is called Roslyn.

We used these features extensively in this project to achieve better performance. For example, we speed up hash functions 2 to 4 times (depending on the machine used) by using runtime-generated code [TODO ADD REFERENCE TO EXPERIMENTS]. Mainly due to these optimizations, we achieved about 130 million encodings per second while maintaining high code genericity.

We do not expect a common reader to know all the little nuances of C# or DotNet. Thus, we decided to provide readers with an introduction to them. However, we assume familiarity with some C-type language.

3.1 Dotnet and Just-In-Time Compilation

Every programming language needs to be translated into machine code. There are two main approaches: compiling and interpreting. Both approaches have their benefits and drawbacks regarding performance, security, and portability. By portability, we mean the ability to run our software easily on many different machines and architectures.

Compiled languages are translated to the assembly before the program starts. This has two advantages. First, security: we may send our clients just the machine code for their specific machine, keeping the source code private. Second, performance: the processor gets the code already translated and doesn't need to do any additional work. The compiler can also optimize the code to run faster. The main problem is that not all processors have the same instruction sets, so we have to compile our software for each of them.

Interpreted languages use an interpreter to read their instructions and translate them to machine code during runtime. This costs time. However, any machine equipped with such an interpreter can run the native code.

There is also a third option: JIT (Just-In-Time) compilation, which lies somewhere in the middle. It involves both compiling and interpreting. It first takes our code written in the language and translates it to IL (Intermediate Language). IL contains most of the instructions used by modern processors. However not all machines have those, thus the IL code still needs to be translated

to the true machine code instruction. This is done during runtime when our program first needs the code.

C# is a JIT language. Native code is first translated to CIL (Common Intermediate Language), and then when the program runs, the CLR (Common Language Runtime) does the heavy work of compiling from CIL to machine code during runtime.

3.2 Memory

3.2.1 Heap and Stack

There are two main types of memory: heap and stack. One should take care not to be confused by the name heap, as it has very little to do with the data structure used for finding the minimum in logarithmic time.

Stack

Stack and heap work quite differently. Stack functions as we would expect a stack to function: data is pushed to the top and also taken from the top. This is useful for recursive parts of our program and is mainly used by methods. One should easily see that methods are, in their nature, recursive. Methods have basically only two options: call some other method or return. When we call a method, we first save the values held by accumulators (registers used by the processor) so they are free to be used by the called method. These values are saved on the top of the stack. Then, after the called method finishes, the saved data is loaded back. Methods may also put their own data on the stack, but it is important to remember that every piece of data on the stack is not safe to use after the method that placed it there has returned. As the stack pointer is now under these data, any method called later can and probably will overwrite them.

Heap

Sometimes, we want to save data between method calls. The heap is used for this purpose. We request memory of a certain size from the heap, and then we are given such memory that is guaranteed not to be used. When we no longer need this memory, we return it to the heap. This process is called allocating and deallocating memory.

This sounds simple, but there are two problems. The first is that we may run out of memory if we do not deallocate. The second is that when we free some memory, some part of the program may still treat that part of the memory as if it still held original data, but that may or may not be true. These are real problems and may cause dangerous bugs and security issues, especially the latter.

3.2.2 Value and Reference Types

We started with heap and stack for a reason. In C#, there may be many different types, but there are only two main kinds of them: value and reference types. They both serve the same purpose of bundling data and operations together; they only differ in the place where they are located in memory. Value types live on

the stack and reference types on the heap. As heap and stack differ greatly, value and reference types behave differently. It is important to know these nuances as they may cause performance issues or unexpected behavior leading to bugs. Also, before we explain the main differences, we take a little detour to terminology. When we are speaking about instances of type, we use term values for instances of value types and objects for instances of reference types.

Value Types

The best example of a value type is an integer or any other numerical type in C#. However, not all numerical types are implemented in C#. For example, complex numbers may be missing or some other. As such, C# provides us with the possibility to define new value types using struct keyword:

```
struct ComplexNumber{
    public double Real = 0;
    public double Imaginary = 0;
    public ComplexNumber(){
    //Constructor
    }
}
```

The main intuition behind structs is that they are merely syntax sugar for grouping fields together.

```
//This is the same ->
double cReal = 0;
double cImaginary = 0;
//To this ->
var c = new ComplexNumber();
```

This leads to some interesting properties of value types. When we call the method, values are passed by copy into it. This means every single field of the value being passed into the method has to be copied. We can see this as a new instance of such a value type created. We show an example:

```
var x  =  new ComplexNumber(){Real = 1, Imaginary = 1};
var y  = new ComplexNumber(){Real = 1, Imaginary = 1};
var z = Add(x, y);

ComplexNumber Add(ComplexNumber a, ComplexNumber b){
    a.Real += b.Real;
    a.Imaginary += b.Imaginary;
    return a;
}
// z.Real is 2
// x.Real is 1
```

In this example, `z.Real` is 2, but `x.Real` is still 1. This is because the fields were copied; thus, when we added `b.Real` to `a.Real`, `x.Real` was not changed.

This behavior is caused by the fact that values live on a stack. Thus, values' lifespan is limited by the life of the method creating them.

For example, if we passed the value type by a pointer to the place where such a value lies on the stack, we could not use the value safely after the method that created it returned, as it could be overwritten at any time.

As copying the whole value type is required, we may sometimes face bottlenecks when using them, especially when the values copied contain many fields.

Lastly, value types are always initiated. For numerical types, C# gurrantees contain 0; if some value type contains a reference type field, it will hold a null value.

Reference Types

We use reference types to model more complex ideas. They are often useful for modeling non-generic objects with global states, such as persons, database entries, and database connections.

Value types are constrained by the lifespan of the method creating them. This is not an issue for reference types as space on the heap is guaranteed to them for eternity or at least until deallocation. As such, copying their fields is unnecessary when passing them to methods. We just need to pass information about where the object lies on the heap. This information is called a reference, and thus this process is called argument passing by reference

Similarly to value types, we might want to define our reference types using the keyword class. We will use nearly the same example as when discussing value types. We only changed the structure to class according to the definition of ComplexNumber.

```
class ComplexNumber{
    public double Real;
    public double Imaginary;
    public ComplexNumber(){
    //Constructor
    }
}

var x  =  new ComplexNumber(){Real = 1, Imaginary = 1};
var y  = new ComplexNumber(){Real = 1, Imaginary = 1};
var z = Add(x, y);

ComplexNumber Add(ComplexNumber a, ComplexNumber b){
    a.Real += b.Real;
    a.Imaginary += b.Imaginary;
    return a;
}
// z.Real is 2
// x.Real is 2
```

Although the code is the same, the result is not. `z.Real` is still 2, but `x.Real` is now 2. The reason is that `x` and `z` are just references pointing to the same object.

Pass by reference has the advantage that when we pass an object to a method, the only cost is a copy of the reference. One may ask why we should use value types when they are more expensive to pass to methods. The answer is that allocations to the heap may be problematic and come with their own costs.

3.2.3 Garbage Collector

In the first section, we learned about the problems with the heap, especially the problem with deallocating memory. Luckily for us, C# is a memory-safe language. This means it solves both problems for us by using a garbage collector (GC). The garbage collector keeps track of all objects allocated by our program.

Remark. This is technically not completely true, as C# allows opting out of the safe context. However, this is something that is used very rarely and only when performance or the need for interop with other languages is required. One seeking such behavior should check unsafe context and Memory<T>.

When the program needs more memory, it enters the garbage-collecting phase, during which some objects with no reference points are found, and the memory used by them is freed. The word some is important, as the process is not deterministic, and we cannot be sure whether or when some unused objects will be collected.

This process is also expensive. This is normally not a problem, but when we allocate a lot in a hot path, it will probably hurt us in terms of performance. (Allocating a new object is also not costless). Also, some garbage collectors need to stop all computations. This may not be feasible in some applications where even short delays are not acceptable, for example, in high-frequency trading.

This problem has only one solution: allocate less. It is possible to achieve this in many ways. Some of the solutions are used quite often. Some solutions are very obscure and used very sporadically. Using values instead of objects is very common; one just needs to consider that passing by copy might be expensive for values having many fields. The next trick in the book is to recycle objects after they will no longer be used. This is usually done via pools, and for some objects that are generally expensive to allocate, we are given an implementation in the base library. One interested should check ArrayPool<T> or ThreadPool<T>.

3.3 Abstraction, Virtual Methods, Delegates, Interfaces

3.3.1 Inheritance and Virtual Methods

Imagine implementing a probabilistic algorithm. This algorithm has a very low chance of cycling and never ending. We aim to use some heuristics to determine whether it is a good idea to continue computation. A very simple heuristic is to set a maximum number of steps. We can implement the algorithm like this:

```
class ProbAlg {
    //Some fields ...
    int _nRounds = 0;
    int _maxNRounds = 100;

    bool StopCondition(){
        return _nRounds < _maxNRounds;
    }

    void Run(){
        while (StopCondition()){
            //Some computation
            _nRounds += 1;
        }
    }
}
```

Later we might want to implement a different stop condition and compare how the algorithms perform. One option would be to use virtual methods and

inheritance.

```
class ProbAlg {
    //Some fields ...
    int _nRounds = 0;
    int _maxNRounds = 100;

    virtual internal bool StopCondition(){
        return _nRounds < _maxNRounds;
    }

    void Run(){
        while (StopCondition()){
            //Some computation
            _nRounds += 1;
        }
    }
}

class ProbAlgBetterStopCondition : ProbAlg {
    override bool StopCondition(){
        return SuperGoodHeuristic();
    }
}
```

The virtual keyword marks the method as virtual, and the internal keyword marks the method as internal, so they may be used by classes inheriting from ProbAlg but not by any other class. Virtual methods' implementations may be overridden by their children. This is important as, during compile time, the compiler does not know which implementation will be used beforehand. This may cause some performance issues as the compiler cannot perform some of its optimizations.

```
ProgAlg x = Create();
//Is x ProgAlg or ProbAlgBetterStopCondition?
//We do not know during compile time.
//x could possibly be either.

ProgAlg Create(){
    return //Creates some object of type AlgProg
}
```

3.3.2 Inheritance on Value Types

Value types cannot be inherited. This makes sense as values are passed using copies, but children may declare new fields, thus needing more memory than their parents. This is not a problem for reference types as references always have the same size.

3.3.3 Interfaces

Let us say we need to work with groups but do not want to implement a special function for each group type. In this case, interfaces are the way to go.

```
interface IGroupElement {
```

```
    IGroupElement Add(IGroupElement a);
    IGroupElement Inverse();
}

struct Z5 : IGroupElement {
    public readonly int Value;
    // Constructor
    public Z5(int value) {
        Value = value;
    }

    IGroupElement Add(IGroupElement a) {
        return new Z5((Value + ((Z5)a).Value) % 5);
    }

    IGroupElement Inverse() {
        return new Z5((5 - Value) % 5);
    }
}

IGroupElement AddAll(List<IGroupElement> elementsToAdd) {
    // Aggregate is C#'s version of fold
    return elementsToAdd.Aggregate((a, b) => a.Add(b));
}
```

'AddAll' is going to work for any implementation of 'IGroupElement'. However, there are some issues. The first is that 'List<IGroupElement>' may hold any value or object implementing 'IGroupElement'. This can result in runtime errors since we haven't defined behavior for adding elements from two different groups.

There's also a performance issue regarding using interfaces over value types. 'List<IGroupElement>' holds references. This is problematic because value types live on the stack and not on the heap, meaning they have to be moved to the heap, creating a new object. This process, called boxing, goes against the reason for using value types: to minimize heap allocations.

3.3.4 Generics

The aforementioned problem has a solution called generics. Generics in C# are similar to templates in C++. Let's look at how to solve this problem using generics.

```
interface IGroupElement<T> {
    T Add(T a);
    T Inverse();
}

struct Z5 : IGroupElement<Z5> {
    public readonly int Value;
    // Constructor
    public Z5(int value) {
        Value = value;
    }

    Z5 Add(Z5 a) {
        return new Z5((Value + a.Value) % 5);
    }
```

```
    Z5 Inverse() {
        return new Z5((5 - Value) % 5);
    }
}

T AddAll<T>(List<T> elementsToAdd) where T : IGroupElement<T> {
    // Aggregate is C#'s version of fold
    return elementsToAdd.Aggregate((a, b) => a.Add(b));
}
```

We have now defined a generic version of ‘AddAll‘. ‘T‘ can be substituted by any type that fulfills all constraints. Currently, we have defined only one constraint: that ‘T‘ implements ‘IGroupElement<T>‘. Now, ‘List<T>‘ can hold only objects or values of one type or their children. This solves the issue of different types in ‘List<IGroupElement>‘ causing runtime errors.

Let's look at how generics behave for value and reference types. For every reference type, there is just one shared CIL code. Every value type has its own CIL code. The code is compiled for the specific type when it is first needed during runtime. This is important because the value type does not need to be boxed. [**Generics1**]

3.4 Expression Trees

Trees Expression are very useful C# API. They allow us to get duck-typing in C# without needing reflection (reflection is expensive and sometimes it may not be feasible due to its costs). This was especially useful before generic math. Expression trees allow us to build code during runtime and then compile and execute it. This may be useful when, for example, we would like to optimize our function for some parameters or we just do not know how the data will look; we want as much performance as possible.

3.4.1 Simple example

If we look at code, we may see it having a tree structure (generally, it is DAG), where nodes are some functions and leaves are constant or parameters. For example:

```
int IfOddAddOne(int input) {
    if (input % 2 == 1){
        return input + 1
    }
    return input;
}
```

Might get translated to this:

```
//Constant nodes
one <- CONSTANT(1)
two <- CONSTANT(2)
//Parameter nodes
input <- PARAMETER()
//Operator nodes
moduloByTwo <- MODULO(input, two)
```

```
addOne <- ADD(input, 1)
isOdd <- EQUALS(one, moduloByTwo)
ifOddAddOne <- IF_ELSE(isOdd, addOne, input)
//Function node
F <- FUNCTION(input, ifOddAddOne)
//Now we can use F as node
two <- F(one)
```

C# allows us to build such expressions during runtime and then to compile them and use them as any other delegate. If we wanted an expression tree for the previous code, we could do that with something like this:

```
//This code was partially generated by ChatGpt
using System.Ling.Expressions
//Parameters
var input = Expression.Parameter(typeof(int), "input");
//Constants
var constOne = Expression.Constant(1);
var constTwo = Expression.Constant(2);
// mod = input % 2
var mod = Expression.Modulo(input, constTwo);
// if-else statement
var condition = Expression.Equal(mod, constOne);
var addOne = Expression.Add(input, constOne);
var ifTrue = addOne;
var ifFalse = input;
//Condition
var ifElse = Expression.Condition(condition, ifTrue, ifFalse);
Expression<Func<int, int>> lambda
= Expression.Lambda<Func<int, int>>(ifElse, input);
Func<int, int> compiled = lambda.Compile();
```

One can see that no output has been assigned. This is because, in Expression Trees, the returned value is always the value of the last expression.

3.4.2 Construct, Compile

Expression trees have two phases. In the first phase, we construct the tree from smaller expressions or other Expression Trees. Expression trees are immutable, and this construction may be done during runtime. Then we may compile the Expression Tree to get a lambda. Compiling is an expensive operation, and we should try to reuse compiled lambdas as much as possible, but the code is going to be optimized in the same way the code was written beforehand.

3.4.3 Base API

We are going to explain some of Expression Trees' base APIs. We are not planning to cover the topic completely but to give a wilful reader a little peek into the concept. There are several different types of expressions: Constants, Parameters, Scopes, Operators, Callers, and Flow Controllers.

Constants

Constants represent a constants This constant needs to be known when we compile the Expression Tree.

```
//Creates a constant expression of type string
var text = Expression.Constant("text");
//Creates a constant expression of type int
var one = Expression.Constant(1);
//We may specify the type explicitly
one = Expression.Constant(1, typeof(int));
//This fails during the compile phase as "q" has a type string and
//no implicit conversion to int
text = Expression.Constant("q", typeof(int));
```

Parameters

Parameters may be both parameters and variables. They represent an expression to which a value can be written or read; thus, they work as variables in normal C# code. There are no differences between them, and they may be used interchangeably.

```
//Create a parameter expression with type int
var x = Expression.Parameter(typeof(int));
// We may assign a value
var assignment = Expression.Assign(x, Expression.Constant(1));
//It is important to remember, that assignment represents
```

Scopes

Scopes are important; they group expressions together. There are basically two types of them Block and Lambdas. They both contain a list of parameters and a list of expressions. Expressions are executed sequentially and may use only parameters in their own parameter list or in any other parameter list of Scope transitively containing it. Example:

```
var x = Expression.Parameter(typeof(int));
var assigment = Expression.Assign(x, Expression.Constant(1));

//Scope behavior
//Works correctly
var firstScope = Expression.Block(new {x}, new {assigment});
//May fail during the compile phase as assignment second
//scope does not contain x
var secondScope = Expression.Block(new {}, new {assigment});
//Works correctly the containing scope provides such a variable
var thirdScope = Expression.Block(new {x}, new {secondScope});
//Fails during compile phase as x already is already declared
//in fourthScope
var fourthScope = Expression.Block(new {x}, new {firstScope});

//Lambdas
//It is useful to remember, that last
//executed expression is also going to be the returned value
var f = Expression.Lambda<Func<int>>(firstScope);
//This returns function always returns 1
Func<int> F = f.Compile();
var f2 = Expression.Lambda<Func<int,int>>(secondScope, x);
////This returns function always returns 1
Func<int, int> F2 = f2.Compile();
```

Operators

Operators allow us to work with all C# operators like =,==,+,-,« etc. For example:

```
var x = Expression.Parameter(typeof(int));
var one = Expression.Constant(1);
var two = Expression.Constant(2);


// x = 1
var assign = Expression.Assign(x, one);
//x += 1
var addAssign = Expression.AddAssign(x, one);
// 1 * 2
var multiply = Expression.Mutiply(one, two);
// (1 * 2) + 2
var add = Expression.Add(multiply, two);
var minus = Exprssion.Minus(multiply, x);
// ((1 * 2) + 2) - x
```

Callers

Sometimes, we need to work with some classes or structures. They may have some fields, properties, or methods. Then, we may use expressions like Call, Field, or Property to call or get some value. It should be noted that in C#, properties might be both methods and fields. This may lead to some nasty bugs if one is unsuspecting of this behavior. One can use the PropertyOrField expression to solve this.

```
//Calls parameterless constructor
var list = Expression.New(typeof(List<int>));
//Call a method with name Add -> a.Add(1)
var listAdd = Expression.Call(list, new {int}, "Add", 1);
//We can get a value from property or field by using PropertyOrField
var listCount = Expression.PropertyOrField(list, "Count");
```

Flow Controls

Control flow is the basis of our programming. Thus, we may use expressions like if, then, loop, and goto. We show an example of a simple loop.

```
var i = Expression.Parameter(int);
var zeroToi = Expression.Parameter(0);
//Assign 0 to 1
var condition = Expression.LessThan(i, Expression.Constant(10));
//Add one to i
var actionToDo = Expression.AddAssign(i, Expression.Constant(1);
var breakLabel = Expression.Label("LoopBreak");
Expression loop = Expression.Loop(
    Expression.IfThenElse(
        condition,
        actionToDo,
        Expression.Break(breakLabel)
        ),
    //Defines break label to jump to
    breakLabel
```

```
    );

var scope = Expression.Scope(new {i}, new {zeroToi, loop });
```

This is equivalent to this code:

```
int i = 0;
while(true){
    if(i < 10) break;
    else i += 1;
}
```

3.4.4 Issues with Expression Trees

One very visible issue with them is that they contain a lot of boilerplate code and are not very readable. If we look at the two previous examples, the latter is clearly more readable. This is a recurring problem with them. Also, the previous example is part of the project's codebase, but it took many iterations to get it right. The native C version was written in seconds with no bugs. This is not only a boilerplate problem but also a problem of no or very low static checking. This means that our code is mainly going to fail during runtime. We give an example of some possible issues that may lead to runtime errors.

```
using System.Ling.Expressions
//Parameters
var input = Expression.Parameter(typeof(int), "input");
//Constants
var constOne = Expression.Constant(1);
var constA = Expression.Constant('A');
var add = Expression.Add(constOne, constA);

Expression<Func<int, int>> lambda
= Expression.Lambda<Func<int>>(add);

lambda.Compile() //Fails there

// but x = 1 + 'A' does not compile
```

One very recurring error is forgetting to increment a value in a while loop, looking like this:

```
int i = 0;
while(i < 10){
    //Do something
    i++;
}
```

Normally, we get a warning that some of our code is unreachable, as we will be forever stuck in the loop. It is, however, not possible to do our code generated using Expression Tree, which exists only in runtime when it is too late to fix such a problem.

3.5 Compiler Optimizations

As C# has a compiler, one should be aware of some optimizations and potential performance bottlenecks.

Cost of Method Call

Calling methods is not free. We need to do some preparations, mainly saving values from accumulators to the stack and then loading them back. This cost is very small but can add up if the method body is small. For example:

```
int AddTwo(int x) {
    return AddOne(AddOne(x));
}

int AddOne(int x) {
    return x + 1;
}
```

Inlining

The compiler can solve this problem by inlining. It may replace the method call with its body, saving the cost of the method call. So we get something like this:

```
int AddTwo(int x) {
    return x + 2;
}

int AddOne(int x) {
    return x + 1;
}
```

Inlining is very powerful as it allows us to write very small methods without losing performance. However, not all methods can be inlined. A method can only be inlined when the compiler knows which method is going to be called.

Simple Example

```
List<int> FilterOdd(List<int> input) {
    var answer = new List<int>();
    foreach(var number in input) {
        if (IsOdd(number)) answer.Add(number);
    }
    return answer;
}
```

This code works, but what if we now need to implement a filter that filters prime numbers? We might modify the preceding code:

```
List<int> FilterPrime(List<int> input) {
    var answer = new List<int>();
    foreach(var number in input) {
        if (IsPrime(number)) answer.Add(number);
    }
    return answer;
}
```

If we look closely, we see that the code of ‘FilterPrime‘ and ‘FilterOdd‘ are very similar. Therefore, it would be nice to abstract the behavior of the filter. We can do something like this:

Solution Using Delegates

```
List<int> Filter(
    List<int> input,
    Func<int, bool> filterCondition
) {
    var answer = new List<int>();
    foreach(var number in input) {
        if (!filterCondition(number)) continue;
        answer.Add(number);
    }
    return answer;
}
```

Now, ‘filterCondition‘ may still be a very short method, but the compiler does not know it at the time of compilation, so it cannot be inlined. Similarly, this would happen if we were using an interface.

Solution Using Interface

```
interface IFilter {
    bool Invoke(int number);
}

List<int> Filter(
    List<int> input,
    IFilter filterCondition
) {
    var answer = new List<int>();
    foreach(var number in input) {
        if (!filterCondition.Invoke(number)) continue;
        answer.Add(number);
    }
    return answer;
}
```

There is a solution to this problem. We’ll first show the code and then explain why it works.

Solution Using Generic Over Struct Interface Method

```
interface IFilter {
    bool Invoke(int number);
}

List<int> Filter<Filter>(
    List<int> input,
    Filter filterCondition
) where Filter : struct, IFilter {
    var answer = new List<int>();
    foreach(var number in input) {
        if (!filterCondition.Invoke(number)) continue;
        answer.Add(number);
    }
    return answer;
}
```

We used a struct constraint to enforce that ‘Filter‘ is a value type. This means we get value behavior on generics, thus generating unique CIL code for every type of filter. Also, since ‘filter‘ is a struct, it cannot have children, so there is no way to override the ‘Invoke‘ method. This means the compiler knows there is only one possible implementation of such a method, thus it can inline it.

3.5.1 When the Compiler Inlines

The .NET compiler uses heuristics to determine whether to inline and where to inline, as inlining may not always be beneficial. There is no way to enforce inlining in C#, but we can give hints to the compiler. This is done using attributes. One can use ‘[MethodImpl(MethodImplOptions.AggressiveInlining)]‘ to increase the chance of a method being inlined and ‘[MethodImpl(MethodImplOptions.NoInlining)]‘ to forbid inlining for that method.

3.6 Constant Optimization

When we use constants, the compiler may optimize their use. For example:

```
int AddTwo(int a) {
return a + 1 + 1;
}
```

Is optimized to:

```
int AddTwo(int a) {
return a + 2;
}
```

This saves us some unnecessary calculations that would otherwise need to be repeated. Modern compilers are quite advanced and can replace some expensive arithmetic operations, such as division and modulo, with less expensive ones, like multiplication. This can result in a significant performance boost, as demonstrated by the following experiment:

Sometimes, we do not know the value to divide with beforehand, but we would still like to benefit from compiler optimizations, as we expect to do many divisions with the same divisor. We know only one method to achieve such functionality without too many obstacles. It is expression trees. One could also try source generation or System.Reflection.Emit namespace to emit CIL during runtime. However, we are not going to describe those techniques.

4 LittleSharp

4.1 Motivation

We used expression trees extensively in the project, but they have a few issues. The first and foremost is that the native Expression Tree library is not statically checked. This leads to many hard-to-debug errors. Also, code written in Expression Trees is complex to read, making debugging such code much harder. As such, we decided to create a wrapper around Expression Trees that would provide some static checking but not at the cost of versatility. We created a standalone library called LittleShape. Even though its main purpose was to serve the needs of our project, we think it may be useful for others. However, it is mainly limited to operations with numeric types.

4.2 How to use

In this section, we introduce the library briefly. First, we will write a short function that declares c with value 5, adds a,b, and c, and returns the sum.

```
// Declare a function with two parameters, a and b
var f = CompiledFunctions
    //Create<TIn1, ... , TInn, TOut>(out var input1_, ...)
    //creates a function with arguments with types TIn1 to Tinn
    //and return type TOut
    //when one wants to create action,
    //they should use CompiledActions class
    .Create<int, int, int>(out var a_, out var b_);
// a and b have types Variable<int>
// f.S is a scope that is going to be executed when f is called
// S is an abbreviation for Scope
// c = 5;
f.S.DeclareVariable<int>(out var c_, 5)
// c have also type Variable<int>
// return a + b + c
  .Assign(f.Output, a.V + b.V + c.V);
// f.Output is the value that is going to be returned
// .Assign(a, b) is the same as a = b;
// a.V is equal to SmartExpression<T>, V is abbreviation for Value
Expression<Func<int, int, int>> addExpression = f.Construct();
Func<int, int, int> add = addExpression.Compile();
//By calling Compile, the expression tree is then compiled
```

We may also use flow control statements, like while and if. Imagine having a buffer B and some number N to determine the number of values in the buffer. We would like to sum all values together, then we could use this code:

```
// Declare a function with two parameters, a and b
var f = CompiledFunction
    .Create<int[], int, int>(out var B_, out var N_);
f.S.DeclareVariable(out var i_, 0)
    //We declared variable i with value 0 of type int
    .While(
        i_.V < N.V,
        new Scope()
```

```
            .DeclareVariable(
                out var value,
                // By calling ToTable<int>(), we hint compiler
                // that the underlying type is a table,
                // thus we may use [] syntax to access fields
                // B_.V.ToTable<int>() is going to have
                // type ArrayAccess<int, int[]>
                B_.V.ToTable<int>()[i_.V]
                )
            .Assign(f.Output, f.Output.V + value.V)
            .Assign(i_, i_.V + 1)

Func<int[], int, int> F = f.Construct().Compile()
```

This code is equivalent to following:

```
// Declare a function with two parameters a and b

var F = (int[] B, int N) => {
    int i = 0;
    int output = 0;
    while(i < N){
        int value = B[i];
        output += value;
    }
    return output;
}
```

4.2.1 Naming Conventions

As we introduced some new concepts, we also introduced Naming conventions for them. Variables should end with underscore _; when the variable is converted to some special type, it should end with _NAME or its abbreviation. For example:

```
var a = CompiledAction.Create<int[]>(out var table_)
a.S.Macro(var out table_T, table_.V.ToTable<int>());
```

4.2.2 Debugging

Debuging expression trees is hard. There are Visual Studio extensions for that. Two libraries can be used to visualize them: Readable Expressions and Expression Tree Visualizer. [**ExpressDebug**] Sadly, Readable Expression does not work with LittleSharp. One can also use some tools provided by the library itself. For example, it is possible to use Print(), to log some data. For example:

```
// Logger has to implement method WriteLine(string line)
class Logger{
    public List<string> Data = new ();
    void WriteLine{string line}{
        Data.Add(line);
    }
}

var logger = new Logger();
```

```
//This is going to change a logger for all instances of Scopes
// As such it should be used only for debugging
// When DebugOutput is not set,
//that it prints to standard output
Scope.SetDebugOutput(logger);

var a = CompiledAction.Create()
a.S.Print("Start")
    .Declare(out var v_, 0)
    .Print(v_.ToStringExpression())
    //There is important not call ToString() as
    //then we would call ToString on v_, what is Variable<int>
    // not the underlying type

a.Construct().Compile()()
// Logger.Data contains "Start" and "0"
```

4.3 Implemenation details

4.3.1 Architecture

There are three main types: Variables, SmartExpression, Scopes

Variables

Variable wraps two possible expressions - ParameterExpression and Variable Expression. This means a Variable should be seen as an expression that holds a value, and a value might be assigned to it. There is a generic version of Variable, and this is Variable<T>. This should be preferred to use as it contains information about the type of variable that is held by Variable<T>.

SmartExpression

SmartExpression wraps all expressions. As Expressions are the main building block of Expression Trees, SmartExpression is going to fit this function building block. SmartExpression also has its generic version, SmartExpression<T>. This generic version also holds an expression of type T.

Scope

Scopes are the most interesting and complex of the types provided. They hold together SmartExpressions and Variables. One should imagine as a normal scope in C#, would look like. They should make it easier not to use variables from different scopes. A generic version, Scope<T>, also has a return type of T.

4.3.2 SmartExpression operators

As our goal is to provide safe arithmetic operations, we simplified SmartExpression to hold some binary numeric type. As such, we overloaded nearly all binary operators to work on the same types. We show an implementation of an overload for addition.

```
public class SmartExpression<TValue>{
    public readonly Expression Expression;
    //More stuff
    public static SmartExpression<TValue> operator+
    (
        SmartExpression<TValue> a,
        SmartExpression<TValue> b
    )
    {
        //Expression.Add( ome_expression, another_expression)
        //is from expression trees api
        return  new SmartExpression<TValue>(
            Expression.Add(a.Expression, b.Expression)
            );
    }
    //More stuff
}
//This allows us to do this
SmartExpression<int> a = 1;
SmartExpression<int> b = 2;
SmartExpression<int> result = a + b;
In the language of expression trees, this can be translated to this
var a = Expression.Constant(1);
var b = Expression.Constant(2);
var result = Expression.Add(a, b);
```

It may not be clear why the latter approach is so bad that developing a new library is a good option. The problem becomes visible with more complex code. The more expressions in the code, the more problems. The problems are mainly with variables, as one needs to make sure they all have the right types and are all initiated. The library solves many of these problems. Typing gives more safety and partially solves problems with assigning values of bad types and then having to deal with runtime errors. For example:

```
var S = new Scope();
S.DeclareVariable(a_, 0);
S.DeclareVariable(b_, 1);
S.DeclareVariable(c_, "a");
S.DeclareVariable(d_, 0l);
// d_ has SmartExpression<long> type

a_ + b_; //Compile error
a_.V + b_.V; // Compiles
a_.V + c_.V; // Compile error as + is overloaded only for
//SmartExpresion<T>s with the same T
a_.V + d.V; // Compile error for the same reason
a_.V.Convert<long>() + d.V; // Compiles
//This could be translated to this code (long) a + d

a_.V.ForceType<long>() + d.V;
// Compiles, fails during runtime
// ForceType cast only SmartExpression<T> to different type,
// but not the underlying expression
a_.Assign(S, b_.V); //Compiles
a_.Assign(S, d_.V); //Does not compile as to Variable<T>
// may be only assigned value with type T
```

This behavior provides enough safety to catch most errors, especially bad assign-

ments.

4.3.3 Scopes

Scopes are more interesting. They hold both variables and expressions to be executed. We may imagine that they hold a list of expressions and a list of variables. Expressions are executed when in the order in which they were added to the scope. It is not executed when we do not add an expression to the scope. All variables are declared before all expressions. When declaring a variable with some assigned value, one should not forget that this assigned value is, in reality, an expression. This means we are doing two things: declaring a variable and assigning a value. Example:

```
var S = new Scope();
S
    .DeclareVariable(a_, 0)
    .Macro(out var x_, a_ + 10)
    .DeclareVariable(b_)
    .Function( (x) => x + 1, x_, out var z_)
    //Upper function is never executed
    //as z_ is then never used.
    .Function( (x) => x + 13, x_, out var y_)
    .Assign(b_, y_)

//First we declared variables a_ and b_
//Then a_ = 0
//Then b_ = 13
```

Scopes may be contained in other scopes and may also contain them. Variables may be used only from the scope they are declared in or any scope contained in that scope. The property being contained is transitive. This behavior is the same as expected from normal scopes in C#.

Scopes may or may not have a return value.

4.4 Future work

This library is mostly experimental and serves more as a glimpse into a concept than a fully functional library. This makes sense as it was a side project created out of a need for better API for Expression Trees.

There is some technical debt in architecture. The main problem is that when a Scope cannot be added as a Scope to another Scope, it needs to be made a SmartExpression first, which is problematic as it makes analyzing such code much harder. To solve this problem, decupling scope trees from expression trees could be a good idea.

LittleSharp would benefit enormously from using Roslyn more extensively, especially source generators and analyzers, to build a less error-prone experience. Analyzers could be used to write custom compile errors, thus helping users debug code written using SmartExpression better. Code Generators could be used to create a more advanced type system supporting custom types.

5 FlashHash

5.1 Motivation

The project required access to families of fast hashing functions. Sadly, no libraries were offering such. The base C# library provides only cryptographic hashing functions. Other libraries offered hashing only from byte arrays. This meant we had to convert integer values to byte arrays, which caused performance bottlenecks. This led to the decision to implement, such a library ourselves.

We would like the library to provide fast hashing functions from long to long for some hashing families.

5.2 How to use

FlashHash implements two hashing function families - Linear Congruence and Multiply Shift.It isIHashingFunctionFamily There are two important concepts to understand:(Hashing Function) Schemes and (Hashing Function) Families.

Schemes

Schemes describe one hashing function. One may build a hashing function out of it:

```
var scheme = new LinearCongruenceScheme(10, 5, 7)
// This scheme yields the following hashing function:
//  (10*x + 5) % Prime % 7
// Prime represents the closest Mersene prime
  //greater than the size (7)

Expression<Func<ulong, ulong>> expression = scheme.Create()
// Building an expression tree is not computationally expensive

Func<ulong, ulong> hashingFunction = expression.Compile();
// However, compilation is resource-intensive
```

Families

Families are merely factories of schemes:

```
var family = new LinearCongruenceFamily();
family.SetRandomness(new Random(42));
// Custom randomness can be set
ulong size = 10;
Func<ulong, ulong> hashingFunction =
    family.GetScheme(size).Create().Compile()
// Retrieves a random hashing function
//from the linear congruence family
```

5.3 Implementation

5.3.1 Simplest Solution

There is very simple solution:

```
var CreateLinearCongruenceHashingFunction =
    (long size) =>
    {
        ulong a = Random.GetRandomUInt64();
        ulong b = Random.GetRandomUInt64();
        ulong p = Random.GetGoodPrime(size);
        return (ulong x) => (a * x + b) % size % prime;
    }
```

This approach has one very large advantage: it is simple. On the other hand, we pay for this simplicity by performance, as constant optimization and inlining are not possible. One can easily see that linear congruence uses modulo operators. These are quite expensive, and even more so if they are not constant optimized. This implementation is circa $2 - 4$ times slower, depending on processor, than when a,b, size, and prime were known in advance.

There are two possible solutions to this in C#: Expression Trees and Runtime Types with Struct Delegates. We talked about these in [number] chapter. We chose a solution via expression trees, but to interested readers, we first describe the solution by Runtime Types with Struct Delegates.

5.3.2 Possible solution via Runtime Types with Struct Delegates

First, we have this implementation:

```
//For sharplab
using SharpLab.Runtime;

interface IConstant{
    ulong Get();
}

//Constants are going to look something like this
struct One : IConstant
{
    public ulong Get() => 1L;
}

struct Six : IConstant
{
    public ulong Get() => 6;
}

struct Prime : IConstant
{
    public ulong Get() => 11;
}
//This is just for sharplab
[JitGeneric(typeof(One), typeof(Six), typeof(Prime),typeof(Six)) ]
struct HashingFunction<TA, TB, TSIZE, TPRIME>
```

```
where TA : struct, IConstant
where TB : struct, IConstant
where TSIZE :  struct, IConstant
where TPRIME : struct, IConstant
{
    public HashingFunction(){}

    readonly TA a = default(TA);
    readonly TB b = default(TB);
    readonly TSIZE size = default(TSIZE);
    readonly TPRIME prime = default(TPRIME);
    public ulong Invoke(ulong value)
        =>
    (a.Get() * value + b.Get())% prime.Get() %size.Get();
}

// Then hf.Invoke = (x) => (1*x + 6) %  11 % 6;
// But compiler knows this during compilation
```

This works. There is one small problem: We need access to families of hashing functions, but we cannot implement a constant type for every number. The solution to this is IL code and runtime types. It is possible in C# to create types during runtime, and then we may use reflection to build hashing functions out of the created constant. Implementation of the Linear Congruence family would thus look like this:

```
static class ConstantFactory(){
    Type GetConstant(ulong value){
    //Magic
    }
}

class LinearCongruenceFamily {
    IHashingFunction GetHashingFunction(size){
        //Prepare
        ulong a = Random.GetRandomUInt64();
        ulong b = Random.GetRandomUInt64();
        ulong prime = GetGoodPrime(size);
        Type A = ConstantFactory.Get(a);
        Type B = ConstantFactory.Get(b);
        Type PRIME = ConstantFactory.Get(prime);
        Type SIZE = ConstantFactory.Get(SIZE);
        //Reflection
        var maker = typeof(LinearCongruence<>).MakeGenericType(
            new {A,B, PRIME, SIZE}
        );
        return (IHashingFunction) Activator.CreateInstance(maker);
    }
}
```

Constant Factory

The implementation of such library function may look something like this:

```
public static Type CreateRuntimeConstant(long value)
{
    //Assembly to which we add the type
    AssemblyName assemblyName = new AssemblyName("DynamicAssembly");
```

```
    AssemblyBuilder assemblyBuilder =
        AssemblyBuilder.DefineDynamicAssembly(assemblyName, AssemblyBuilderAc
    ModuleBuilder moduleBuilder =
        assemblyBuilder.DefineDynamicModule("DynamicModule");

    //Type builder with struct semantics
    TypeBuilder tb =
        moduleBuilder.DefineType(
            $"long_{value}",
            TypeAttributes.Public | TypeAttributes.Sealed |
            TypeAttributes.SequentialLayout
            | TypeAttributes.BeforeFieldInit,
            typeof(ValueType),
            new Type[] { typeof(IConstant) }
        );

    var method = tb.DefineMethod("GetValue",
    MethodAttributes.Public |
    MethodAttributes.Virtual |
    MethodAttributes.HideBySig, typeof(long), Type.EmptyTypes);
    var il = method.GetILGenerator();
    il.Emit(OpCodes.Ldc_I8, value);
    il.Emit(OpCodes.Ret);
    tb.DefineMethodOverride(
        method,
        typeof(IConstant).GetMethod("GetValue")
        );
    return tb.CreateType();
}
```

5.3.3 Solution using Expression Trees (LittleSharp)

We are going to use LittleSharp for work with expression trees. Reasons for developing this library are given in the chapter about LittleSharp.

Using LittleSharp the simplest solution looks like this:

```
var CreateLinearCongruenceHashingFunction =
    (long size) =>
    {
        ulong a = Random.GetRandomUInt64();
        ulong b = Random.GetRandomUInt64();
        ulong p = Random.GetGoodPrime(size);
        var f = CompiledFunctions.Create<ulong, ulong>
        (out var input_, out var hash_);
        f.S.Assign(f.Output, (a * input_ + b) % size % prime);
        return f.Construct().Compile();
    }
    //Returns Func<ulong, ulong>
    //This code is constant optimized
```

5.3.4 Comparing aproaches

Both approaches add significant complexity to the project. The main issue is that switching the context between runtime-generated and normal code is not smooth. One needs to use reflection calls in case of IL-generated code or delegates

calls for Expression Trees. Also, using runtime-generated code will cause type safety issues. This is because types cannot be checked before the code is generated, which happens during runtime. This may cause many hard-to-find bugs that may cause runtime errors.

Both approaches are comparable. They both lead to constant optimized code and both them allows inlining with some addtional work. Expression Trees are easier to work with because their API is simpler, and they require much less reflection code. Some type of safety can also be added; this is discussed in the chapter LittleSharp. Finally, expression trees are probably used more by the community. This means there are more sources to learn about them. Taking this all into account, we have chosen the approach using Expression Trees.

5.4 Peformance

ToDo

5.5 Future work

We see the library as very complete, but it could provide more hashing functions families. It could also posibly provide a possibilty to hash from any integer type to any integer type. This would be actually very simple to implement. Some sort of cashing for compiled expression tree hashing function could be useful.

6 RedaFasta

6.1 Motivation

Our goal is to write a reader of FASTA files that does not cause bottlenecks to our main algorithm. As such, it needs to be reasonably fast. We are willing to sacrifice genericity for performance.

6.2 Preliminaries

6.2.1 DNA and kMers

For our purposes, one may imagine DNA as a string of four letters: A, C, G, and T. Two and two form pairs, this being A with T and C with G. To every DNA string S, there exists its complement C. This complement may be obtained by first replacing all letters with the other letter in pairs and then reversing the string. This means "AAAT" has a complement "ATTT".

DNA strings may be long, and getting them may be hard. As such, we often work on sets of kMer. kMer is a substring of length k. Similarly, as DNA has a complement, kMers also have complements. We would sometimes need to have just one representation of kMer; thus, we would define a canonical kMer as being lower in lexicographical ordering.

6.2.2 Fasta Files

When we have a set of keys, it is k-times more expensive to store them than storing a DNA string. The problem is that building DNA from set of kMer is hard. Thus fasta file format is used. On the first line of the file is configuration, and on the next line is the "DNA" string, but its letter may be upper or lower. Then, the kMer set stored is equal to all substrings of such "DNA" string beginning with an upper symbol.

6.3 How to use

The library provides two classes, FastaFile and FastaFileReader. FastaFile serves to initialize and recover configurations from FASTA files. FastaFileReader is the main class doing all reading, buffering, and parsing kmers.

Canonical Order

Reader returns canonical kMers.

Opening a FastaFileReader

The first line of the fasta file is expected to be a configuration. There must be two parameters l and k. l is the number of, chars in sequence, and k is the number of chars in a kmer. then it is expected to be a line with all chars in sequence.

The sequence should contain only A,C,G,T or a, c, t, g. The reader does not check this; different symbols may be treated as any char. A reader reads only l characters from the line.

```
//Open fasta file
string path = "file.fasta"
//Open TextReader
TextReader tr = new StringReader(path);
//Get configuration
var config = FastaFile.Open(tr);
//Open reader
var fastaReader =
    new FastaFileReader(
        config.KMerSize,
        config.nCharsInFile,
        tr
    );
```

Users may use FastaFileReader in two ways: using its own buffer or borrowing and then returning a buffer. Both methods have advantages and disadvantages. It is not recommended to mix both methods, as this may lead to highly underfilled buffers. FastaFileReader does not guarantee that KMer is read in order.

Using own buffer

This method always tries to fill the buffer as much as possible. This may be beneficial if we need more consistent counts of numbers in the buffer. This is, however, at the cost of an additional copy, causing it to be less performant, than borrowing

```
FastaFileReader fastaReader;
//We may use our own buffer and let it fill
var buffer = new ulong[1024];
int nKMerInBuffer = fastaReader.FillBuffer(buffer);
//0 means there is no KMer to be read and the stream ended
```

Borrowing a buffer

This buffer returns some buffer. Buffer has three fields: Data, Size, and Used. Size the number of KMer in the buffer; Used is the number of KMer already used. This means when Used has value N, the Data[N] is the first valid value. Used should be 0, but using the method FillBuffer() may cause some buffers to have Used fields with different values

```
FastaFileReader fastaReader;
//We may borrow a buffer from the reader
var borrowedBuffer = fastaReader.Borrow();
//We then should return the buffer back to the fasta reader
//If do not return a buffer,
//fasta file reader may be forced to allocate a new one.
//This may cause performance issues.
if (borrowedBuffer is not null)
    fastaReader.Recycle(borrowedBuffer);
```

6.3.1 Disposing

FastaFileReader does not dispose of the TextReader it was provided with. The user should do this themselves by calling the Dispose() method.

6.4 Future work

This library has a very specific purpose, and it fits that purpose well. It could be upgraded by adding some sort of checks and then allowing users to choose whether to use them.

7 SymmetricDifferenceFinder

7.1 Motivation

We aimed to develop a library to test different approaches to solve set recovery problems. We decided we would like performance and modifiability, even at the cost of speed and ease of development. Performance is essential because the expected use case may deal with massive data sets. We need modifiability to be able to change or replace some components easily.

7.2 Overview

SymmetricDifferenceFinder is our project's core library. It is a small framework for building solutions to the set recovery problem. It allows users to easily modify or replace many of its components and parts, allowing them to test different approaches without repeating themselves.

7.2.1 Encoding-Decoding Pipeline

There are two main phases: encoding and decoding. During encoding, we encode our data using some small sketches. There should be one sketch for each set. During decoding, we first take sketches of sets we want to find symmetric differences, merge them, and then run a decoding algorithm on the result of merging. This step may fail. If it fails, we then need to change the parameters and rerun the algorithm from the start.

7.2.2 Sketch/Table

The most important thing is to choose a Sketch/Table to use. We first make a little detour into the difference between Tables and Sketches. Tables are used for encoding, and Sketches are utilized for decoding. Most implementations may be the same, but the library's architecture allows the implementation of different Sketches for the same Table.

If not said explicitly, we will use Sketch with the meaning of both Table and Sketch.

One should view Sketch as an array with overloaded methods to "add" and "remove" items.

This is, for example, an ITable interface. Encoding relies on eveSketchtch to implement this interface.

```
public interface ITable
{
    void Add(Hash key, Key value);
    int Size();
}
```

7.2.3 Hashing Functions

Hashing functions are used in both phases. One must use the same set hashing function for encoding and decoding. This set is expected to implement IEnumerable<Expression<Func<ulong, ulong> > interface. This may, for example, be a List<Expression<Func<ulong, ulong> >. Order is not important. Some basic hashing functions are provided by the FlashHash library. One can also implement their own.

7.2.4 Encoder

Encoders are used in the encoding phase. There are two implementations of Encoders: Encoder<TTable> and NoConflictEncoder<TTable>. TTable has to implement an ITable interface and be a value type (this is to get inline). NoConflictEncoders expect that no two hashing functions that have intersecting ranges exist. There are two possible interfaces for implementing IEncoder and IParralelEncoder.

7.2.5 Decoders

Decoders are used in the decoding phase. They should be given a sketch and then return decoded values and information on whether decoding was successful. There are two possible decoders: HPWDecoder, based on the HWP algorithm, and HyperGraphDecoder, which uses standard hypergraph decoding. It should be noted that these two decoders may not be used interchangeably because they need different contracts from Sketch.

7.2.6 HWPDecoder

HPWDecoder is based on the HPW algorithm. It allows the removal of items that are not present in the set or that are not going to be preset after the encoding ends. Due to its nature, it cannot distinguish whether some element in the symmetric difference of sets A and B was an element of A or an element of B.

7.2.7 HyperGraphDecoder

It uses a [algorithm] to find the largest 2 core of the hypergraph. In its base version, it cannot remove items that are not present in the sketch or that will be present. We also implemented an improved version of such an algorithm, that supports such operations.

7.2.8 Massagers

Massagers try to take a Decoder in an unrecoverable state and shake little with the data to increase recoverability. Their work only on some data. We need to guess some items, possibly present in the symmetric difference, from already decoded keys. They are especially useful when only a few items remain undecoded. Their effectiveness depends largely on how good our guesses are.

7.2.9 Data sources

The library implements multiple possible data sources. RedaFasta may be used to provide a Fasta file as a data source, and multiple random data generators are available for testing purposes.

7.2.10 Testing

One can also find some useful tools for writing TODO.

7.3 Building a Custom Solution

This section will describe how to use this library to create a custom solution to set recovery problems. For example, we would like to improve a solution using HyperGraphDecoder using a space-saving dictionary.

7.3.1 Creating a table

A table needs to conform to the ITable interface interface. We are going to implement a simpler version of the table. We expect that items will only be removed from tSketchtch during the decoding phase. This is true if we have two sets A and B and $B \subseteq A$.

```
//Tables need to be structs
struct IBLTTableImprovedMemory: ITable
{
    ulong[] _keySums;
    SpaceSavingDictionary _counts;
    public IBLTTableImprovedMemory(int size){
        _data = ulong[size];
        _counts = new SpaceSavingDictionary(size);
        _hashSum = new SpaceSavingDictionary(size);
    }
    // Hash is allias for ulong
    // Key is allias for ulong
    // This means writing ulong or Hash or Key is equivalent
    void Add(Hash key, Key value){
        _keySums[key] += value;
        _counts.Add(1);
    }
}
```

7.3.2 Choosing Hash Functions

After choosing a table or writing its implementation, one should select a set of hashing functions. This set should implement an IEnumerable<T> interface, where T is Expression<Func<ulong, ulong> >. The hash function range should not exceed the table size; otherwise, the program fails during runtime.

```
using FlashHash;
//Simplest solution is this
ulong size = 10;
//This creates an array holding one hash function using modulo
```

```
//This approach has a problem with performance
//One should rather FlashHash library
Expression<Func<ulong, ulong>>[] hfs = [(x) => x % size];
//Solution using FlashHash library
//This solution gets us two Linear Congruence hash function
//with not conflicting ranges
hfs=
[
HashingFunctionProvider
    .Get(typeof(LinearCongruenceFamily), size / 2, 0).Create(),
HashingFunctionProvider
    //5 sets an offset to the hash function
    //It means all values of hashes are going to be increased by 5
    .Get(typeof(LinearCongruenceFamily), size / 2, 5).Create(),
]
```

7.3.3 Choosing Encoder

The basic library currently offers two encoders: Encoder and NoConflictEncoder. NoConflictEncoder expects that no hash functions intersect in ranges. This allows for some optimizations during encoding; namely, it will enable parallel writing to the table.

```
Expression<Func<ulong, ulong>>[] hfs;
int size = 10;
var config =
    EncoderConfiguration<IBLTTableImprovedMemory>(hfs, size);
//Factory does most of the compiling, the user
// seeking performance should reuse Factory as much
// as possible
var factory = EncoderFactory<IBLTTableImprovedMemory>(
    config,
    //This is table factory
    (size) => new IBLTTableImprovedMemory(size)
)
IDecoder decoder = factory.Create()
```

7.3.4 Using Encoder

The user must write their method to wrap a data source and encoder. Here, we provide an example of using RedaFasta to read the files in [SomeFormat].

```
//Opening a [SomeFormat] file
string fastaFilePath = "test.test";
var config = FastaFile.Open(new StreamReader(fastaFilePath));
var reader = new FastaFileReader(
    config.kMerSize, config.nCharsInFile, config.textReader
    );

//Some instances implementing the IEncoder interface
IEncoder encoder;

//It may be a good idea to set the buffer to some size that is large enough
var buffer = new ulong[1024 * 1024];
while (true)
{
```

```
    var data = reader.BorrowBuffer();
    if (data is null)
    {
        break;
    }
    //ParralelEncode may speed the encoding
    encoder.ParralelEncode(buffer, data.Size);
    reader.RecycleBuffer(data);
}

encoder.GetTable() //Returns table
```

7.3.5 Sketch

We should revisit the definition of IBLTTableImprovedMemory. We decided to use HyperGraphDecoder. HyperGraphExpect a sketch to implement IHyperGraphDecoderSketch<TSketch>. Now, we have two choices implement a brand new Sketch and some method transforming Table to the new Sketch or extend IBLTTableImprovedMemory to implement ISketch interface. We are going to use the second way. We will not implement it entirely, but to show important parts.

```
struct IBLTTableImprovedMemory:
    ITable,
    IHyperGraphDecoderSketch<IBLTTableImprovedMemory>
{
    ulong[] _keySums;
    SpaceSavingDictionary _counts;
    public IBLTTableImprovedMemory(int size){
        _data = ulong[size];
        _counts = new SpaceSavingDictionary(size);
        _hashSum = new SpaceSavingDictionary(size);
    }
    void Add(Hash key, Key value){
        _keySums[key] += value;
        _counts.Add(1);
    }
    //This allow the decoding algorithm
    //to remove items from buckets
    void Remove(Hash key, Key value){
        _keySums[key] -= value;
        _counts.Add(-1);
    }
    int GetCount(Hash key) => _counts.Get(key);
    //Now, there is little bit a tricky part
    //we need to implement public looks pure
    //if this method is not defined decoder fails during runtime
    static Expression<Func<ulong, IBLTTable, bool>>
    GetLooksPure(HashingFunctions hashingFunctions)
    {
    var f = CompiledFunctions
            .Create<ulong, IBLTTable, bool>(
                out var key_,
                out var sketch_);

    f.S.Assign(f.Output, false)
        .DeclareVariable(
            out var count_,
```

```
                sketch_.V.Call<int>("GetCount", key_.V)
                )
            .IfThen(
                //Bucket is pure when there is only one item
                //hashed to it
                !(count_.V == 1),
                new Scope().GoToEnd(f.S)
                    )
        return f.Construct();
}
}
```

7.3.6 Choosing Decoder

If we decided to implement an IHyperGraphDecoderSketch<TSketch>, we might use only a decoder that can use such Sketch. There is only one implementation, and this is currently a HyperGraphDecoder. We first need to merge the two sketches of the two Sets; we would like to find symmetric differences.

```
//Using a hypergraph decoder
IBLTTableImprovedMemory sketch1;
IBLTTableImprovedMemory sketch2;
//Know we want to merge these into sketches
//Every sketch needs to implement SymmetricDifference()
//So, imagine we implemented such a method
//This implementation could loop something like this:
public IBLTTable SymmetricDifference(IBLTTable other)
{
    if (other.Size() != Size())
    {
        throw new InvalidOperationException(
            "Sketches do not have same sizes"
            );
    }
    IBLTCell[] data = new IBLTCell[Size()];
    _table.CopyTo(data, 0);
    //This means if some a was encoded in A and also B
    //Then it is not encoded in the merged sketch
    for (int i = 0; i < data.Length; i++)
    {
        IBLTCell otherCell = other.GetCell((ulong)i);
        _table[i].KeySum -= otherCell.KeySum;
        _count.Add(i, -otherCell.GetCount(i));
    }
    return new IBLTTable(data);
}
//Know we just need to merge the two sketches
sketch1.SymmetricDifference(sketch2);
```

After we get a merged sketch, we may create a decoder and use it.

```
IEnumerable<Expression<Func<ulong, ulong>> hfs;
var factory = HypergraphFactory(hfs);
var decoder = factory.Create(sketch);
decoder.Decode()
//We may examine the decoding state by
decoder.EncodingState
//This property may have many possible states
```

```
//there are three most useful:
// - *Success*
// - *Failure*
// - *Shotdown* (have not finished)

decoder.GetDecodedValues()
// Returns a hash set of values in the symmetric difference
//We may get some decoded values even
//when the algorithm is in a fail state.
```

7.4 Buffers size

Buffer sizes: Todo Optimize the right size of buffers

7.5 Future Work

8 Experimets

8.1 Compare hash function

-Whether they improve recovery -Test all LinCon a MultiplyShift, maybe test tabelation -How they influence speed of algorithm

8.2

-compare algorythms

8.3 Test massagers

-ciselne rady -kmery

Conclusion

List of Figures

List of Tables

List of Abbreviations

A Attachments

A.1 First Attachment