StringMatch: Parallel Rabin-Karp Algorithm for Exact String Matching

Project Report - COMS 4995 Parallel Functional Programming

Eumin Hong (eh2890@columbia.edu) Chris Yoon (cjy2129@columbia.edu)

ABSTRACT

This report presents a parallel Rabin-Karp algorithm implemented in Haskell for the exact string matching problem. We provide multiple versions for single-pattern matching, and evaluate each implementation's performance on the DNA exact substring matching problem on a reference human genome.

1 INTRODUCTION

String matching algorithms locate occurrences of a specific pattern within a larger string or text file. The exact string matching problem is an extensively studied problem in computer science, finding applications in numerous areas such as bioinformatics, network security, database systems, and document matching.

We focus on the *exact* string matching problem (as opposed to approximate); locating exact matches of a search pattern within a larger text file. In particular, we study the parallelization of the Rabin-Karp algorithm [4], a popular string matching algorithm that finds exact matches in $O(n)$ expected runtime.

2 BACKGROUND

2.1 Naive String Matching Algorithm

To understand the underlying reason of the expected linear runtime of the Rabin-Karp algorithm, we first note the naive (brute-force) string matching algorithm.

An intuitive brute-force method is as followed: Let w be the search space string where $|w| = n$ and p be the desired pattern string where $|p| = k$. Then for each substring $s \in w$ where $|s| = m$, check if $s = p$ character-wise. If the substring and desired pattern match, then record the position of the substring. Once all substrings of length m in w are considered, return all the recorded positions.

While seemingly efficient, this algorithm has the worst-case running time of $O(nk)$ as there are $O(n)$ substrings to consider and the naive method of checking if two strings s and p are equal takes $O(k)$ time.

2.2 The Rabin-Karp Algorithm

We now discuss the Rabin-Karp Algorithm in more detail. Suppose we seek to find matches of a pattern p of length k in a larger text. The algorithm uses a hash function to perform a initial comparison of strings. Since a substring does not match the pattern if their hash values disagree, comparing the hash values, an $O(1)$ operation, allows avoid the $O(k)$ time character-wise string comparison for such obvious non-matches.

As such, the core of the algorithm is the efficient computation of the hash values of each successive substrings. In particular, the hash function for an encoded string $s = s_0 \cdots s_{k-1}$ of length k is given by the *Rabin fingerprint*, given by the polynomial function

$$RF(s) = s_0 b^{k-1} + s_1 b^{k-2} + \cdots + s_{k-1}$$

where b is usually a prime, and all arithmetic is done in modulo q for some large prime q. Instead of computing the polynomial for every successive substring, Rabin-Karp computes the hash value in a sliding-window fashion, where given $RF(s)$, we can compute the hash value of the next substring $RF(s' = s_1 \cdots s_k)$ using only constant number of operations, via

$$RF(s') = \left(RF(s) - s_0 \cdot b^{k-1}\right) \cdot b + s_k$$

In particular, after the hash value computation of the very first substring, every successive computation will take $O(1)$ time. Using this rolling hash scheme, the Rabin-Karp algorithm can be outlined as followed:

Algorithm 1 (Sequential) Rabin-Karp Algorithm

```
1: Input: Pattern $p$ ($|p| = k$) and string $S$ ($|S| = N$) to be searched.
2: Initialize: empty list $L$ to store indices of matches
3: Compute hash value $h_p$ of pattern $p$
4: for $i = 0 \ldots, N - 1 - k$ do
5:     Compute hash value of substring $h_s = RF(s = s_i \cdots s_{i+k-1})$
6:     if $h_s = h_p$ then
7:         if $s = p$ character-wise then
8:             add $i$ to output list $L$.
9: Output $L$
```

Since we only need to compare the hash values to rule out obvious non-matches, this scheme significantly cuts down the text comparisons the algorithm must perform. Consequently, although the worst-case runtime is still $O(nk)$, a good hash function (one that is unlikely to produce false positives) reduces the expected runtime to $O(n)$.

3 PARALLELIZATION

Consider contiguous partitions $P_1, \ldots, P_n$ of the input text in which the input pattern to be searched. Clearly, each partition can be thought of as an independent string. In that sense, the rolling hash computation on a partition is not dependent on that of any other partition. This gives a natural way of parallelizing the Rabin-Karp algorithm, where the rolling hash computation on every partition can be done in parallel.

However, one issue we must resolve is that such naive partitioning excludes instances of the pattern string that cross the border between two partitions; for instance, the window (of length $k = |p|$) starting at the last character of a partition will not have its hash value computed. To remedy this, we extend each partition to have

COMS 4995: Parallel Functional Programming,
2021.

access to the first $k-1$ characters of the next partition (hence allowing the partitions to overlap).

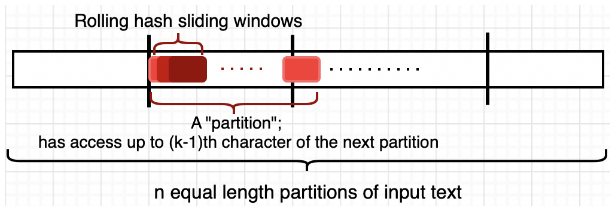
Rolling hash sliding windows
A "partition";
has access up to (k-1)th character of the next partition
n equal length partitions of input text

Figure 1: Overlapping partitions

Using this partitioning scheme, we can perform a full search of the input text in parallel, by chunks. We defer the actual Haskell implementation of this scheme to section 4.2.

4 HASKELL IMPLEMENTATION

4.1 Sequential Algorithm Implementation

Our base (sequential) implementation for the Rabin-Karp algorithm follows the pseudocode in Algortihm 1, but with a slight modification to simplify the code logic: as we perform the rolling hash, we first collect all substrings whose hash value matches that of the pattern before comparing them character-wise with the pattern. We then identify exact matches after all hash-value matches are collected. We made this choice since a good enough hash function would theoretically make the memory overhead associated with storing the matches negligible.

To that end, we first implemented the function `rabinKarpRoll`, which performs a rolling-hash on the input text and outputs a list of all substrings with there indices whose hash value matches that of the pattern:

```
rabinKarpRoll :: String -> HashValue -> Int -> [(Int, String)]
rabinKarpRoll text targetHash ws = roll text "" 0 0
  where
    roll [] subStr hashC idx
      | targetHash == hashC = [(idx, subStr)]
      | otherwise      = []
    roll (x:xs) subStr hashC idx
      | isShorter = roll xs (subStr ++ [x]) hashC' idx
      | isMatch   = (idx, subStr) : roll xs (ss ++ [x]) hashR (idx + 1)
      | otherwise = roll xs (ss ++ [x]) hashR (idx + 1)
     where
      isShorter = length subStr < ws
      isMatch   = length subStr == ws && targetHash == hashC
      hashC'    = modM $ ex + modM (hashC * b)
      hashR     = modM $ ex + modM ((hashC - es * (modExp b (ws - 1) m)) * b)
      modM val  = val `mod` m
      (s:ss)    = subStr
      (ex, es)  = (DC.ord x, DC.ord s)
      (b, m)    = (31, 100003)
```

Then, we implement the function `rabinKarpMatch` to identify exact matches in the collected substrings,

```
rabinKarpMatch :: String -> [(Int, String)]-> [Int]
rabinKarpMatch pattern candidates = DL.map (\(idx, _) -> idx) matches
  where
    matches = DL.filter (\(_, str) -> str == pattern) candidates
```

and used the two functions the to implement the full `rabinKarp` function:

```
rabinKarp :: String -> String -> [Int]
rabinKarp pattern text = rabinKarpMatch pattern candidates
  where
    candidates    = rabinKarpRoll text patternHash patternLength
    patternHash   = polyHash 31 100003 pattern
    patternLength = length pattern
```

4.2 Parallel Algorithm Implementation

As described in section 3, our approach for parallelization relies on efficiently reading partitions of the input text. Inspired by the HipgRap project [3] from PFP 2019, we also use the POSIX way of file reading in Haskell. In particular, we use the `fdPread` function from `System.Posix.IO.ByteString.Lazy` which calls the `pread` function in the C programming language via `foreign import ccall safe` [5]; it takes the number of bytes to read and the offset from the start of the string. Using this functionality and its laziness, we were able to partition and read from the input text at different locations without having to read the entire file:

```
readPartition :: FilePath -> Int -> Int -> Int -> IO DBLC.ByteString
readPartition filePath numParts patternLength partNum = do
    fileSize <- getFileSize filePath
    let fileMode   = Just (CMode 0440)
        partSize   = fileSize `div` (fromIntegral numParts)
        partOffset = partSize * (fromIntegral partNum)
        readSize   = if partNum == (numParts - 1)
                         then partSize
                         else partSize + fromIntegral (patternLength - 1)
        readSizeB  = (fromIntegral readSize) :: ByteCount
    fd <- PIO.openFd filePath PIO.ReadOnly fileMode PIO.defaultFileFlags
    chunk <- PIOB.fdPread fd readSizeB partOffset
    return chunk
```

With this function, we load P partitions of the input text to a list, and `map` a curried `rabinKarp` function on each partition. Naturally, use the `parList` strategy with `rdeepseq` to completely evaluate the list of `rabinKarp` tasks in parallel. After all results are evaluated, we perform necessary post-processing (such as offset correction for each partition) and return the indices of exact-matches. This parallelization is implemented as followed:

```
parRabinKarpN :: String -> String -> Int -> IO [Int]
parRabinKarpN pattern filePath n = do
    fileSize <- getFileSize filePath
    partitions <- mapM (readPartition filePath n (length pattern)) [0..(n-1)]
    let partsB  = map DBL.unpack partitions
        matches = runEval $ do
            let ms = (map (rabinKarp pattern) partsB) `using` parList rdeepseq
            return ms
    let indicesByPart         = zip [0..(n-1)] $ map (map fromIntegral) matches
        partSize              = (fromIntegral fileSize) `div` n
        applyOffset (np, idcs) = map (np * partSize +) idcs
        offsetCorrected       = map applyOffset indicesByPart
    return $ concat offsetCorrected
```

5 EVALUATION

5.1 Benchmark Set

As written earlier, the various implementations of the Rabin-Karp exact string matching algorithm were tested on the human reference genome in FASTA format [1], which is roughly 3.1 GB of the four base pairs: "A", "T", "C", and "G" along with "N", which is a placeholder for an unknown base pair. The specific pattern searched for while testing was the sequence "CTAGATTTGAT".

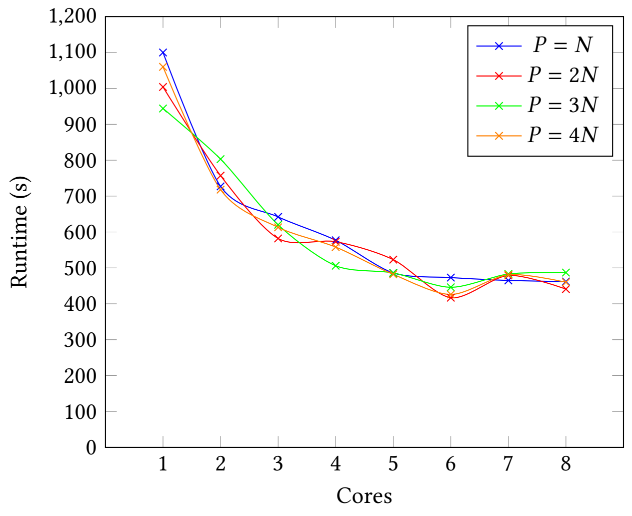

Figure 2: Runtime evaluations for various different runs.

Cores	2	3	4	5	6	7	8
Speedup	1.37	1.67	1.85	2.08	2.33	2.30	2.22

Figure 3: Average speedup for N cores relative to $N = 1$.

5.2 Results

All results in this paper were obtained using a 2021 MacBook Pro with a Apple M1 Max chip and 10 cores (8 performance and 2 efficiency), but only up to 8 cores were used during testing. The runtime of each instance (an implementation with a specific number of cores N and number of partitions P) was measured by taking the minimum runtime from three iterations, as the lowest value better approximates the fastest possible runtime for the given instance [2]. For `parRabinKarpN`, with N threads, $P \in \{N, 2N, 3N, 4N\}$ partitions of the search string were tested.

As the number of threads increases from $N = 1 \longrightarrow N = 2$, the performance significantly increases (a speed up of 1.37 times on average). However, as the number of cores increases, the speedup does not increase as much, as adding another core results in diminishing returns in speedup. Figure 3 contains the average speedups for N cores, and the best-case speedup is observed when $N = 6$ for a speedup of 2.33 times. Furthermore, there seems to be no statistically significant difference between the number of partitions.

The threadscope analysis in Figure 4 of running `parRabinKarpN` with $N = 8$ and $P = 24$ partitions indicates that the workload is evenly distributed amongst the four cores.

For the parallel Rabin-Karp Haskell implementation proposed, each spark, which represents a partition of the original search string, is deeply evaluated using `rdeepseq`. As a result, most sparks are converted, as seen in Figure 5.

5.3 Amdahl's Law

Amdahl's law is used to measure the theoretical maximum amount of speedup achievable through parallelism. To do this, the parallelizable fraction of the task from the equation $S = \frac{1}{(1-P)+\frac{P}{N}}$ was empircally determined to be approximately 0.62. Therefore, the theoretical maximum speedup was found to be $\lim_{N\to\infty} S = 2.63$ times.

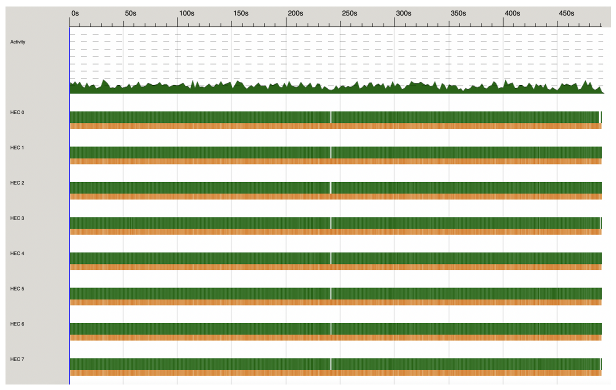

Figure 4: Threadscope analysis of the parallel algorithm with $N = 8$ threads and $P = 24$ partitions.

Total	Converted	Overflowed	Dud	GC'd	Fizzled
24	23	0	0	0	1

Figure 5: Spark results for the parallel algorithm with $N = 8$ threads and $P = 24$ partitions.

6 CONCLUSION

We implemented both sequential and parallel versions of the Rabin-Karp algorithm in Haskell for the exact string matching problem. We evaluated our implementation on DNA substring matching, finding a 10-character long pattern in a human reference genome FASTA file of 3.1 GB. While Amdahl's law reveals a theoretical maximum speedup of 2.63 times, with $N = 6$ cores, our implementation was able achieve a speedup of 2.33 times.

We include the source code of our project in the appendix below. We include detailed instructions on how to run the program with sample data in the `README.md` file of our code submission. While we have also implemented the Rabin-Karp algorithm for (fixed-length) multiple-pattern matching, we have focused on single pattern matching for this report. Future work may include rigorously investigating and optimizing process utilization of our program. Moreover, we may also explore approximate matching or matching with wild-cards.

REFERENCES

[1] 1000GenomesProject. 2008. "Humanreferencegenome". ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/technical/reference/humang1kv37.fasta.gz
[2] Wonhyuk Choi and Andrew de Soler. 2020. SAT: Parallel SAT Solver with DPLL - Report. http://www.cs.columbia.edu/~sedwards/classes/2020/4995-fall/reports/SAT.pdf
[3] Bicheng Gao and Gangwei Lin. 2019. HipgRap - Report. http://www.cs.columbia.edu/~sedwards/classes/2019/4995-fall/reports/HipgRap.pdf
[4] Richard M. Karp and Michael O. Rabin. 1987. Efficient randomized pattern-matching algorithms.
[5] Wren G. Romano. 2010. System.Posix.IO.ByteString. https://hackage.haskell.org/package/unix-bytestring-0.3.7.6/docs/System-Posix-IO-ByteString.html#g:2

CODE LISTING

We give a brief overview of the project source code structure:

- **StringMatch/RabinKarp.hs**: Contains the base (sequential) Rabin-Karp algorithm detailed in section 4.1.
- **StringMatch/Parallel.hs**: Contains the parallelization of the implemented `rabinKarp` function, as detailed in section 4.2.
- **StringMatch/FileReader.hs**: Implements `readPartition`, which allows to lazy (hence efficient) reading of partitions of an input text.
- **StringMatch/Match.hs**: Contains high-level functions that users would use.
- **app/Main.hs**: Main file for running `StringMatch`.

We also provide example usages to illustrate how to run the program. The general usage is:

```
stack run ["pf"] [pattern or path_to_pattern] [path_to_search_space_text] [num_partitions]
```

a) To run a sequential StringMatch where the pattern is given as a string, do

```
stack run pattern path_to_text
```

b) To run a parallel StringMatch where the pattern is given as a string, do

```
stack run p pattern path_to_text num_partitions
```

c) To run a sequential StringMatch where the pattern is in a file, do

```
stack run f path_to_pattern path_to_text
```

d) To run a sequential StringMatch where the pattern is in a file, do

```
stack run pf path_to_pattern path_to_text num_partitions
```

For more detailed instruction on how to use the code with a sample data included in the code submission, please read the `README.md` file in the submission.
All code is available on GitHub: https://github.com/cyoon1729/StringMatch.

StringMatch/RabinKarp.hs

```haskell
{-
 - Implements the Rabin-Karp algorithm for single and multiple
   pattern matching
-}
module StringMatch.RabinKarp
    (
      rabinKarp
    , rabinKarpMulti
    )  where


import qualified Data.Char as DC
import qualified Data.List as DL
import qualified Data.Set as DS
import qualified Data.Map as DM
import qualified Data.Bits as DB


type HashValue = Int


-- | Modular exponentiation, taken from https://gist.github.com/trevordixon/6788535
modExp :: Int -> Int -> Int -> Int
modExp b 0 m = 1
modExp b e m = t * modExp ((b * b) `mod` m) (DB.shiftR e 1) m `mod` m
  where
    t = if DB.testBit e 0 then b `mod` m else 1


-- | polynomial hash for rabin-karp hashing pattern
polyHash :: Int -> Int -> String -> HashValue
polyHash b m str = foldl (\acc c -> polyMod c acc) 0 str
  where
    polyMod c acc = modM $ (DC.ord c) + modM (acc * b)
    modM val     = val `mod` m


{- Modules for single-pattern Rabin-Karp String Matching -}

-- | Internal Rabin-Karp rolling hash function helper that discards non-matches.
rabinKarpRoll :: String -> HashValue -> Int -> [(Int, String)]
rabinKarpRoll text targetHash ws = roll text "" 0 0
  where
    roll [] subStr hashC idx
      | targetHash == hashC = [(idx, subStr)]
      | otherwise     = []
    roll (x:xs) subStr hashC idx
      | isShorter = roll xs (subStr ++ [x]) hashC' idx
      | isMatch  = (idx, subStr) : roll xs (ss ++ [x]) hashR (idx + 1)
      | otherwise = roll xs (ss ++ [x]) hashR (idx + 1)
      where
        isShorter = length subStr < ws
        isMatch  = length subStr == ws && targetHash == hashC
        hashC'   = modM $ ex + modM (hashC * b)
        hashR    = modM $ ex + modM ((hashC - es * (modExp b (ws - 1) m)) * b)
        modM val = val `mod` m
        (s:ss)   = subStr
        (ex, es) = (DC.ord x, DC.ord s)
        (b, m)   = (31, 100003)


-- | Outputs indices that match pattern.
rabinKarpMatch :: String -> [(Int, String)]-> [Int]
rabinKarpMatch pattern candidates = DL.map (\(idx, _) -> idx) matches
  where
    matches = DL.filter (\(_, str) -> str == pattern) candidates


-- | Rabin-Karp with decoupled candidate selection and matching.
rabinKarp :: String -> String -> [Int]
rabinKarp pattern text = rabinKarpMatch pattern candidates
  where
    candidates   = rabinKarpRoll text patternHash patternLength
    patternHash  = polyHash 31 100003 pattern
    patternLength = length pattern


{- Modules for multi-pattern Rabin-Karp string matching -}

-- | Internal Rabin-Karp rolling hash function helper that discards non-matches.
rabinKarpRollMulti :: String -> DS.Set HashValue -> Int -> [(Int, String)]
rabinKarpRollMulti text targets ws = roll text "" 0 0
  where
    roll [] subStr hashC idx
      | DS.member hashC targets = [(idx, subStr)]
      | otherwise             = []
    roll (x:xs) subStr hashC idx
      | length subStr < ws = roll xs (subStr ++ [x]) hashC' idx
      | isMatch          = (idx, subStr) : roll xs (ss ++ [x]) hashR (idx + 1)
      | otherwise        = roll xs (ss ++ [x]) hashR (idx + 1)
      where
        isMatch = length subStr == ws && DS.member hashC targets
        hashC'  = modM $ ex + modM (hashC * b)
        hashR   = modM $ ex + modM ((hashC - es * (mExp b (ws - 1))) * b)
        modM val = val `mod` m
        mExp p q = modExp p q m
        (s:ss)  = subStr
        (ex, es) = (DC.ord x, DC.ord s)
        (b, m)  = (31, 100003)


-- | Outputs indices that match patterns.
rabinKarpMatchMulti :: DS.Set String -> [(Int, String)] -> [(String, [Int])]
rabinKarpMatchMulti patterns candidates = sortIdxs matches
  where
    sortIdxs     = DL.map (\(patt, idxs) -> (patt, DL.sort idxs))
    matches      = DL.filter (\(a, _) -> DS.member a patterns) candidateIdxs
    candidateIdxs = DM.toList $ DM.fromListWith (++) flipPadded
    flipPadded   = DL.map (\(idx, str) -> (str, [idx])) candidates


-- | Performs fixed-length multi-pattern rabin-karp matching.
rabinKarpMulti :: [String] -> String -> [(String, [Int])]
rabinKarpMulti patterns text = rabinKarpMatchMulti patternSet candidates
  where
    patternSet   = DS.fromList patterns
    candidates   = rabinKarpRollMulti text patternHashes patternLength
    patternHashes = DS.fromList $ DL.map (polyHash 31 100003) patterns
    patternLength = length (head patterns)
```

StringMatch/Parallel.hs

```haskell
{-
 - Implements parallelization of the Rabin-Karp algorithm
-}
module StringMatch.Parallel
    (
      parRabinKarpN
    ) where

import qualified Data.ByteString.Lazy.Char8 as DBL
import Control.Parallel.Strategies (using, parList, runEval, rseq, rdeepseq)
import Control.DeepSeq

import StringMatch.RabinKarp (rabinKarp)
import StringMatch.FileReader (getFileSize, readPartition)


-- | Run Rabin Karp in parallel, splitting text into n partitions.
parRabinKarpN :: String -> String -> Int -> IO [Int]
parRabinKarpN pattern filePath n = do
    fileSize <- getFileSize filePath
    partitions <- mapM (readPartition filePath n (length pattern)) [0..(n-1)]
    let partsB = map DBL.unpack partitions
        matches = runEval $ do
            let ms = (map (rabinKarp pattern) partsB) `using` parList rdeepseq
            return ms
    let indicesByPart       = zip [0..(n-1)] $ map (map fromIntegral) matches
        partSize            = (fromIntegral fileSize) `div` n
        applyOffset (np, idcs) = map (np * partSize +) idcs
        offsetCorrected      = map applyOffset indicesByPart
    return $ concat offsetCorrected
```

StringMatch/Match.hs

```haskell
{-
 - Implements utilities to parse program arguments and
   perform match
-}
module StringMatch.Match
    (
      doMatch
    ) where


import System.IO
import StringMatch.Parallel (parRabinKarpN)
import StringMatch.RabinKarp (rabinKarp)


usage :: [String]
usage = [
         "Usage:"
        , "- To match non-parallel, where input pattern is given as string:"
        , "     stack run pattern path_to_text"
        , "- To match parallel, where input pattern is given as a string:"
        , "     stack run p pattern path_to_text num_partitions"
        , "- To match non-parallel, where input pattern is given as a file path:"
        , "     stack run f path_to_pattern path_to_text"
        , "- To match parallel, where input pattern is given as a file path:"
        , "     stack run pf path_to_pattern path_to_text"
        ]

foundPattern :: String -> String
foundPattern pattern = "Found \"" ++ pattern ++ "\" in locations:"


-- | Match in sequential, where pattern is given as a string
matchStr :: String -> String -> IO [Int]
matchStr pattern filePath = do
    text   <- readFile filePath
    return $ rabinKarp pattern text


-- | Match in parallel, where pattern in given as a string
matchStrPar :: String -> String -> Int -> IO [Int]
matchStrPar pattern filePath numPartitions = do
    matches <- parRabinKarpN pattern filePath numPartitions
    return matches


-- | Match in sequential, where pattern is given as a file
matchFile :: String -> String -> IO [Int]
matchFile patternPath filePath = do
    text    <- readFile filePath
    patternRaw <- readFile patternPath
    let pattern = filter (/='\n') patternRaw
    return $ rabinKarp pattern text


-- | Match in parallel, where pattern is given as a file
matchFilePar :: String -> String -> Int -> IO [Int]
matchFilePar patternPath filePath numPartitions = do
    text <- readFile filePath
    patternRaw <- readFile patternPath
    let pattern = filter (/='\n') patternRaw
    matches <- parRabinKarpN pattern filePath numPartitions
    return matches


-- | Parse program arguments and perform specified match
doMatch :: [String] -> IO [()]
doMatch args = do
    case args of
        [pattern, fPath] -> do
            matches <- matchStr pattern fPath
            putStrLn $ foundPattern pattern
            mapM putStrLn $ map show matches
        ["p", pattern, filePath, numPartitions] -> do
            let numParts = read numPartitions :: Int
            matches <- matchStrPar pattern filePath numParts
            putStrLn $ foundPattern pattern
            mapM putStrLn $ map show matches
        ["f", pattPath, filePath] -> do
            patternRaw <- readFile pattPath
            let pattern = filter (/='\n') patternRaw
            matches <- matchFile pattPath filePath
            putStrLn $ foundPattern pattern
            mapM putStrLn $ map show matches
        ["pf", pattPath, filePath, numPartitions] -> do
            patternRaw <- readFile pattPath
            let pattern = filter (/='\n') patternRaw
                numParts = read numPartitions :: Int
            matches <- matchFilePar pattPath filePath numParts
            putStrLn $ foundPattern pattern
            mapM putStrLn $ map show matches
        _ -> do
            mapM putStrLn usage
```

app/Main.hs

```
module Main where

import System.IO
import System.Environment (getArgs)
import Lib (doMatch)


main :: IO [()]
main = do
    args <- getArgs
    doMatch args
```