МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЁТ

по лабораторной работе №4

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритм Кнута-Морриса-Пратта

Студент гр. 8303 _____ Парфентьев Л.М.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы

Изучение алгоритма Кнута-Морриса-Пратта поиска образца в строке.

Задание

- Задание 1

  Реализуйте алгоритм КМП и с его помощью для заданных образца P ($|P| \leqslant 15000$) и текста T ($|T| \leqslant 5000000$) найдите все вхождения P в T.

  Вход:

  > Первая строка – P
  >
  > Вторая строка – T

  Выход:

  > Индексы начал вхождений P в T, разделенных запятой, если P не входит в T, то вывести -1.

  Sample Input:

  ```
  ab
  abab
  ```

  Sample Output:

  ```
  0,2
  ```

- Задание 2

  Заданы две строки A ($|A| \leqslant 5000000$) и B ($|B| \leqslant 5000000$).

  Определить, является ли A циклическим сдвигом B (это значит, что A и B имеют одинаковую длину и A состоит из суффикса B, склеенного с префиксом B). Например, `defabc` является циклическим сдвигом `abcdef`.

  Вход:

Первая строка – A

Вторая строка – B

Выход:

Если A вляется циклическим сдвигом B, индекс начала строки B в A, иначе вывести -1. Если возможно несколько сдвигов вывести первый индекс.

Sample Input:

```
defabc
abcdef
```

Sample Output:

```
3
```

Вариант 1

Подготовка к распараллеливанию: работа по поиску разделяется на k равных частей, пригодных для обработки k потоками (при этом длина образца гораздо меньше длины строки поиска).

Описание алгоритма

Алгоритм Кнута-Морриса-Пратта используется для поиска всех вхождений образца в строке за время, линейно зависящее от длины строки. Для этого выполняется предварительная обработка образца, а также при обработке очередного символа учитывается информация, полученная при обработке предыдущих символов.

Префиксом длины k строки s называется подстрока длины k, которая совпадает с началом s длины k, т. е. с первыми k символами в s. Аналогично, *суффиксом* длины k строки s называется подстрока длины k, которая совпадает с хвостом s.

Границей длины k строки s назовём строку длины k, являющуюся одновременно и префиксом, и суффиксом s (но не совпадающей с s).

Префикс-функция строки s и индекса i равняется длина наибольшей границы подстроки $s[0 : i - 1]$ ($s[a : b]$ обозначим подстроку s от символа с номером a до b включительно). Значение префикс-функции строки s и индекса i обозначим как p_i.

Если известны значения префикс функции для строки s и всех индексов от 0 до i включительно, можно получить её значение для индекса $i + 1$ следующим образом:

1. Попробуем расширить максимальную границу. Её ширина равняется значению префикс-функции в i, т. е. p_i. Тогда мы можем расширить границу если $s_{i+1} = s_{p_i}$ (нумерация символов в строке идёт с нуля). В этом случае размер максимальной границы в строке $s[0 : i + 1]$ будет равен $p_i + 1$ ($p_{i+1} = p_i + 1$), т. е. мы просто расширим границу на 1 символ.

2. Если расширить границу не удаётся, мы берём другую границу и пытаемся расширить её. Здесь используется следующее свойство границ: если u – граница v, а v – граница s, то u также является границей s.

   Таким образом, мы хотим найти максимальную границу текущей максимальной границы, и попытаться её расширить. Её длина будет равна p_{p_i-1}, и это также индекс символа, с которым мы теперь будем сравнивать текущий.

3. Далее, мы будем брать максимальную границу той границы, которую мы пытаемся расширить, пока или не найдём такую, которую мы *можем* расширить, или не обраружим, что мы не можем расширить даже границу длины 0. В этом случае $p_{i+1} = 0$.

Чтобы получить вхождения образца t в строку s, мы можем найти значения префикс функции для каждого индекса в строке ts, но с ограничением на длину границы (т. е. на значение префикс-функции) $|t|$. Этого ограниче-

ния можно добиться, вставив между t и s специальный символ, который не встречается ни в t, ни в s.

Вхождения t в s будут заканчиваться там, где значение префикс-функции будет равно $|t|$. Тогда вхождение, заканчивающееся на индексе i, будет начинаться на индексе $i - |t| + 1$.

Также поиск можно реализовать не конкатенируя строки, что позволяет хранить значения префикс-функции только для t, но не для s, т. к. будут использоваться только значения для t и предыдущего индекса в s. Ограничение на значение префикс-фукнции придётся вводить отдельно. Собственно, нахождение значений префикс-функции для образца и есть его предварительная обработка.

В этом случае, граница станет префиксом образца и суффиксом пройденной части строки, в которой выполняется поиск, а не префиксом и суффиксом одной и той же строки.

Распараллеливание алгоритма

Разделим строку s, в которой осуществляется поиск на n равных (или почти равных) участков. Эти участки пересакаются, причём ширина пересакающихся частей равна $|t| - 1$ (t – образец). Тогда мы получим n участков, на которых можно независимо запустить алгоритм, и получить все вхождения, причём каждое вхождение будет найдено ровно 1 раз.

Поскольку алгоритм находит вхождение, когда доходит до его конца, ему нужно пройти $|t|-1$ символов прежде чем он найдёт хотя бы одно вхождение. В нашем разбиении, вхождения, заканчивающиеся на одном из этих $|t| - 1$ символов, будут найдены экземпляром, обрабатывающим предыдущий участок, поэтому вхождения не теряются. Разумеется, в первых $|t| - 1$ символах исходной строки s вхождения t заканчиваться не могут.

Естесственно, что каждый участок не может быть короче, чем образец, поэтому наибольшее число участков, на которые можно разбить строку, будет $|s| - (|t| - 1)$. В этом случае длина каждого участка будет равна $|t|$.

Поиск циклического сдвига

Поиск циклического сдвига можно рассматривать как поиск в удвоенной строке. Если мы проверяем, является ли s циклическим сдвигом t, мы можем запустить алгоритм поиска подстроки s в tt.

При реализации алгоритма строку не обязательно на самом деле удваивать, достаточно просто продолжить поиск с начала, но в качестве значения префикс-функции в предыдущем символе взять её значение в последнем символе при первом проходе.

Описание функций

Программа была написана на языке Go, поскольку в нём есть удобные механизмы для запуска лёгких потоков (*горутин*) и коммуникации между ними (*каналы*). Была написана параллельная реализация алгоритма в обоих заданиях (и поиск подстроки, и поиск циклического сдвига). Выбор количества потоков осуществляется при помощи ключа командной строки `-j`; по-умолчанию используется один поток.

Несколько функций используются и в программе поиска подстроки, и в поиске циклического сдвига:

- `func prefixAt(n string, np []int, c byte, i int) int`

  Возвращает значение префикс-функции при поиске образца. `n` – образец, `np` – префикс-функция для каждого индекса в образеце, `c` – текущий символ, `i` – номер символа в образеце, с которого надо начинать сравнивать.

  Эта функция используется как при поиске образца, так и при построении массива значений префикс-функции для одной и той же строки. В последнем случае, `n` – это та же строка, к которой строится массив, `np` – этот самый массив, а `i` – значение префикс-функции в предыдущем символе.

- `func prefix(s string) []int`

Строит массив значений префикс-функции для строки `s`.

- ```
  func findMatches(
      haystack, needle string,
      n_prefix []int,
      initial int,
      offset int,
  ) ([]int, int)
  ```

  Ищет вхождения `needle` в `haystack`. Другие аргументы:

  – `n_prefix` – значения префикс-функции для `needle`;

  – `initial` – значение префикс-функции в символе, идущем перед строкой, используется при поиске циклического сдвига. Если поиск происходит с начала строки, аргумент должен быть равен нулю.

  – `offset` – Сдвиг, который нужно прибавить к индексу начала вхождения. Используется в параллельной реализации (там он равен индексу начала данного участка) и во втором проходе при поиске циклического сдвига (иначе индекс начала вхождения будет отрицательным).

  Эта функция возвращает массив (`[]int`, в Go называется slice) индексов вхождений, а также значение префикс-функции в последнем символе `haystack`, которое используется при составлении нескольких вызовов этой функции (это значение в таком случае нужно будет передать в качестве значения параметра `initial`).

- `func splitKmpWork(len_s, len_substr, n_ranges int) []Range`

  Возвращает массив участков, на которые нужно разделить строку `len_s` при поиске подстроки длины `len_substr`. Функция выдаёт не более `n_ranges` участков.
  ```

Если `n_ranges` $> n_{\max} =$ `len_s` $-$ `len_substr` $+ 1$ (максимальное число участков), всё равно будет возвращено $n_{\max}$ участков.

Если `len_s` $-$ `len_substr` $+ 1$ не делится нацело на число участков, то получить участки равной длины невозможно и первые r участков будут на 1 символ длинее остальных (r – остаток от деления).

Тип `Range` состоит из индекса начала участка и его длины:

```
type Range struct {
    offset, length int
}
```

Следующие функции используются при поиске подстроки:

- ```
  func parallelFindMatches(
      haystack, needle string,
      n_prefix []int,
      ranges []Range,
  ) []int
  ```

  Возвращает массив вхождений образца `needle` в строку `haystack`. Параметр `n_prefix` – значения префикс-функции для образца. `ranges` – участки, на которые надо разбить строку. Каждый участок будет обрабатываться в отдельном потоке.

  Эта функция работает следующим образом:

  1. Для каждого участка запускается `findMatches` в отдельной горутине. Результаты выполнения складываются в канал `finish_ch`.
  2. Собираются результаты. Порядок, в котором нужно объединять массивы индексов вхождений известен, т. к. вместе с ними возвращается номер участка.
  3. Полученные результаты объединяются в один массив.

- ```
  func ParallelSearchSubstring(
      haystack, needle string,
  ```

```
    n_ranges int,
) []int
```

Вычисляет префикс-функцию для образца `needle` при помощи `prefix`, вычисляет участки (не более `n_ranges`), на которые можно разделить строку поиска при помощи `splitKmpWork`, а затем вызывает функцию `parallelFindMatches`.

Функции поиска циклического сдвига:

- ```
  func findCyclicMatch(
      hd, tl, orig string,
      orig_prefix []int,
  ) int
  ```

  Ищет `orig` в строке `hd` + `tl`. Конкатенация строк на самом деле не происходит, вместо этого делаются два вызова `findMatches`, причём второе возвращаемое значение из первого вызова (значение префикс-функции в последнем символе `hd`) передаётся во второй вызов в качестве параметра `initial`.

  Если совпадение найдено не было, возвращает -1.

- `func ParallelCheckCyclic(str, orig string, n_pieces int) int`

  Работает по схожему с `parallelFindMatches` принципу, но сама получает префикс-функцию для `orig` и разбиение на участки (при помощи тех же функций `prefix` и `splitKmpWork`).

  Перед началом работы проверяет, что ей переданы строки одинаковой длины.

  Вначале запускаются горутины, которые запишут возвращаемое значение в канал `finish_ch`, а затем из результатов, которые они возвращают, выбирается наименьший. Таким образом, эта функция вернёт минимальный индекс если возможно несколько сдвигов.

Используются также несколько утилит:
  ```

- `func printOffsets(off []int)`

  Записывает числа из `off` по порядку, через запятую, на стандартный вывод. Если слайс `off` пустой, выводит `-1`. В любом случае выводит перевод строки в конце.

- `func readLine(rd *bufio.Reader) (string, error)`

  Считывает строку с буферизованного потока `rd`. Возвращает считанную строку и ошибку, если она есть. Строка не содержит символа перевода строки.

  Если ошибка является `io.EOF`, строка была считана, но вместо символа перевода строки последовал конец файла. В этом случае строку всё равно можно использовать, если после неё ничего не ожидается.

Обе программы можно запустить с ключом `-v`, который включит вывод промежуточных данных.

Сложность алгоритма

Алгоритм обрабатывает каждый символ строки s. Для каждого символа он ищет наибольшую границу, которую можно расширить. Затем он или увеличивает длину текущей границы на 1, или делает её равной нулю.

Отбросить границу можно не большее число раз, чем было проведено расширение границы, а расширить границу можно не более $|s|$ раз. Таким образом, получаем линейную сложность прохода по s $O(|s|)$.

Кроме прохода по s, перед поиском вхождений t в s нужно выполнить предварительную обработку образца – получить значения префикс-функции для каждого символа в t. Это то же самое, что мы делаем с s, поэтому сложность предобработки $O(|t|)$. Поскольку $|t| < |s|$, можно считать, что суммарная сложность поиска подстроки по времени $O(|s|)$.

Для алгиритма требуется хранить образец и значения префикс-функции для каждого его символа, это $O(|t|)$. Хранить строку, в которой происходит

поиск не обязательно (хотя в данной реализации она целиком загружается в память), хранить индексы вхождений также необязательно (их можно обрабатывать сразу при нахождении, но в данной реализации они сохраняются), поэтому в теории сложность алгоритма по памяти может быть $O(|t|)$. Сложность по памяти в данной реализации $O(|s|)$.

Сложность поиска циклического сдвига, где длины обеих строк равны n равна сложности поиска подстроки длины n в строке длины $2n$. Таким образом, сложность по времени $O(n)$, сложность по памяти также $O(n)$.

Тестирование

- Тесты поиска подстроки (программа search.go)

1. Ввод:

```
ab
abababababababbbabababaabba
```

   Вывод:

```
0,2,4,6,8,10,12,14,18,20,22,25
```

2. Ввод:

```
the
the quick brown fox jumps over the lazy dog
```

   Вывод:

```
0,31
```

3. Ввод:

```
bao
foo bar baz zoo
```

   Вывод:

```
-1
```

4. Ввод:

```
abaa
abaabaa
```

   Вывод:

```
0,3
```

5. Ввод:

```
abracadabra
abrakadabracadabracadabra
```

Вывод:

```
7,14
```

- Тесты поиска циклического сдвига (программа rotation.go)

1. Ввод:

```
abraabracad
abracadabra
```

Вывод:

```
4
```

2. Ввод:

```
bababababaaba
abababababab
```

Вывод:

```
1
```

3. Ввод:

```
foobar
foobar
```

Вывод:

```
0
```

4. Ввод:

```
foobar
bazfoo
```

Вывод:

```
-1
```

5. Ввод:

```
foobar
roobaf
```

Вывод:

```
-1
```

6. Ввод:

```
abaa
aaba
```

Вывод:

```
3
```

Выводы

В ходе выполнения лабораторной работы был изучен алгоритм Кнутта-Морриса-Пратта для нахождения подстроки в строке, а также схема его распараллеливания. Написана параллельная реализация на языке Go.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: search.go

```
package main

import (
        "io"
        "os"
        "bufio"

        "flag"

        "fmt"
        "log"
        "strings"
)

var verbose bool

// --- KMP algorithm implementation

// n: the string to search in
// np: prefix function computed for n. Only np[:i+1] is ever accessed
// c: character/byte at the current position
// i: first index in ‘n’ in ‘c’ with
func prefixAt(n string, np []int, c byte, i int) int {
        if verbose {
                fmt.Fprintf(os.Stderr, "\tLooking for: '%c'\n", c)
        }
        for {
                if verbose {
                        if i >= len(n) {
                                fmt.Fprintf(os.Stderr, "\tOut of bounds\n")
                        } else {
                                fmt.Fprintf(os.Stderr,
                                        "\tCurrent char: '%c'\n", n[i])
                        }
                }

                switch {
                case i < len(n) && c == n[i]:
                        if verbose {
                                fmt.Fprintf(os.Stderr,
                                        "\tMatching char at %d\n", i)
                        }

                        return i + 1
                case i == 0:
```

```
                        if verbose {
                                fmt.Fprintf(os.Stderr, "\tNo match\n")
                        }

                        return 0
                }
                i = np[i - 1]

                if verbose {
                        fmt.Fprintf(os.Stderr, "\tFalling back to %d\n", i)
                }
        }
}

// Compute the prefix function at each offset in 's'
func prefix(s string) []int {
        if len(s) == 0 {
                return nil
        }
        prefix := make([]int, len(s))
        prefix[0] = 0
        if verbose {
                fmt.Fprintf(os.Stderr, "At 0: 0 (automatically)\n")
        }
        for i := 1; i < len(s); i++ {
                start := prefix[i - 1]

                if verbose {
                        fmt.Fprintf(os.Stderr, "At %d:\n", i)
                        fmt.Fprintf(os.Stderr, "\tStarting at %d\n", start)
                }

                prefix[i] = prefixAt(s, prefix, s[i], start)

                if verbose {
                        fmt.Fprintf(os.Stderr, "At %d: %d\n", i, prefix[i])
                }
        }
        return prefix
}

// haystack: the string to search in
// needle: the string to search for
// n_prefix: prefix function for each offset in needle
// initial: initial match length. Useful when chaining multiple calls
// offset: the value to add to matches. Useful when splitting search
func findMatches(
        haystack, needle string,
        n_prefix []int,
        initial int,
        offset int,
) ([]int, int) {
```

```
        var matches []int
        prev := initial
        n := len(needle)
        for i := 0; i < len(haystack); i++ {
                real_idx := offset + i

                if verbose {
                        fmt.Fprintf(os.Stderr, "At %d:\n", real_idx)
                        fmt.Fprintf(os.Stderr, "\tPrev = %d\n", prev)
                }

                prev = prefixAt(needle, n_prefix, haystack[i], prev)

                if verbose {
                        fmt.Fprintf(os.Stderr, "At %d: %d\n", real_idx, prev)
                }

                if prev == n {
                        match_idx := real_idx - n + 1
                        matches = append(matches, match_idx)

                        if verbose {
                                fmt.Fprintf(os.Stderr, "Match at %d-%d\n",
                                        match_idx, real_idx)
                        }
                }
        }
        return matches, prev
}

// --- Parallel implementation

type Range struct {
        offset, length int
}

// len_s: length of haystack
// len_substr: length of needle
// n_ranges: max number of ranges to return
func splitKmpWork(len_s, len_substr, n_ranges int) []Range {
        prematch_area := len_substr - 1

        // match_area: max count of matches
        match_area := len_s - prematch_area

        // In the extreme case, each thread will be checking its own offset.
        if n_ranges > match_area {
                n_ranges = match_area
        }

        //
```

```
        match_one_min := match_area / n_ranges
        match_one_rem := match_area % n_ranges

        // Each thread's match area begins at offset + prematch_area.
        offset := 0
        // '+1': we distribute the remainder (r) among the first r processes.
        match_len := match_one_min + 1

        // Allocate beforehand, so no reallocation
        ranges := make([]Range, 0, n_ranges)
        for i := 0; i < n_ranges; i++ {
                if i == match_one_rem {
                        match_len -= 1
                }
                ranges = append(ranges, Range{
                        offset,
                        match_len + prematch_area,
                })
                offset += match_len
        }

        return ranges
}

func parallelFindMatches(
        haystack, needle string,
        n_prefix []int,
        ranges []Range,
) []int {
        type searchResult struct {
                idx int
                mat []int
        }

        // Spawn a goroutine for each range. Each goroutine sends its index and
        // matches from its range to finish_ch.
        finish_ch := make(chan searchResult)
        for i, rng := range ranges {
                offset := rng.offset
                end := offset + rng.length
                piece := haystack[offset : end]
                go func (i int) {
                        m, _ := findMatches(piece, needle, n_prefix, 0, offset)
                        finish_ch <- searchResult{i, m}
                }(i)

                if verbose {
                        fmt.Fprintf(os.Stderr,
                                "Goroutine %d started in range from %d to %d\n",
                                i, offset, end)
                }
        }
```

```
        // Fetch results. We need matches to be in the correct order, so we put
        // match slices in order.
        results := make([][]int, len(ranges))
        total := 0
        for i := 0; i < len(ranges); i++ {
                res := <-finish_ch
                results[res.idx] = res.mat
                total += len(res.mat)

                if verbose {
                        fmt.Fprintf(os.Stderr, "Goroutine %d finished: %v\n",
                                res.idx, res.mat)
                }
        }

        if verbose {
                fmt.Fprintf(os.Stderr, "Total: %d matches\n", total)
        }

        // Produce a slice of total matches. We won't have to reallocate because
        // we specify the required capacity.
        matches := make([]int, 0, total)
        for _, res := range results {
                matches = append(matches, res...)
        }

        return matches
}

func ParallelSearchSubstring(haystack, needle string, n_ranges int) []int {
        if verbose {
                fmt.Fprintf(os.Stderr,
                        "Calculating prefix function of search pattern...\n")
        }

        n_prefix := prefix(needle)

        if verbose {
                fmt.Fprintf(os.Stderr,
                        "Prefix function of search pattern: %v\n", n_prefix)
        }

        ranges := splitKmpWork(len(haystack), len(needle), n_ranges)
        return parallelFindMatches(haystack, needle, n_prefix, ranges)
}

func printOffsets(off []int) {
        if len(off) == 0 {
                fmt.Println(-1)
                return
        }
```

```
        fmt.Print(off[0])
        for _, x := range off[1:] {
                fmt.Printf(",%v", x)
        }
        fmt.Println()
}

func readLine(rd *bufio.Reader) (string, error) {
        str, err := rd.ReadString('\n')
        if err != nil {
                // if err == io.EOF, the string may non-empty and useful
                return str, err
        }

        return strings.TrimSuffix(str, "\n"), nil
}

func main() {
        var n_threads int

        flag.IntVar(
                &n_threads, "j", 1,
                "Number of `threads` to use",
        )
        flag.BoolVar(
                &verbose, "v", false,
                "Show detailed info on algorithm's execution",
        )
        flag.Parse()

        buf_rd := bufio.NewReader(os.Stdin)

        str_a, err := readLine(buf_rd)
        if err != nil {
                log.Fatal(err)
        }

        str_b, err := readLine(buf_rd)
        if err != nil && err != io.EOF {
                log.Fatal(err)
        }

        res := ParallelSearchSubstring(str_b, str_a, n_threads)

        printOffsets(res)
}
```

Название файла: rotation.go

```
package main
```

```
import (
        "io"
        "os"
        "bufio"

        "flag"

        "fmt"
        "log"
        "strings"
)

var verbose bool

// --- KMP algorithm implementation

// n: the string to search in
// np: prefix function computed for n. Only np[:i+1] is ever accessed
// c: character/byte at the current position
// i: first index in ‘n’ in ‘c’ with
func prefixAt(n string, np []int, c byte, i int) int {
        if verbose {
                fmt.Fprintf(os.Stderr, "\tLooking for: '%c'\n", c)
        }
        for {
                if verbose {
                        if i >= len(n) {
                                fmt.Fprintf(os.Stderr, "\tOut of bounds\n")
                        } else {
                                fmt.Fprintf(os.Stderr,
                                        "\tCurrent char: '%c'\n", n[i])
                        }
                }

                switch {
                case i < len(n) && c == n[i]:
                        if verbose {
                                fmt.Fprintf(os.Stderr,
                                        "\tMatching char at %d\n", i)
                        }

                        return i + 1
                case i == 0:
                        if verbose {
                                fmt.Fprintf(os.Stderr, "\tNo match\n")
                        }

                        return 0
                }
                i = np[i - 1]

                if verbose {
```

```
                        fmt.Fprintf(os.Stderr, "\tFalling back to %d\n", i)
                }
        }
}

// Compute the prefix function at each offset in 's'
func prefix(s string) []int {
        if len(s) == 0 {
                return nil
        }
        prefix := make([]int, len(s))
        prefix[0] = 0
        if verbose {
                fmt.Fprintf(os.Stderr, "At 0: 0 (automatically)\n")
        }
        for i := 1; i < len(s); i++ {
                start := prefix[i - 1]

                if verbose {
                        fmt.Fprintf(os.Stderr, "At %d:\n", i)
                        fmt.Fprintf(os.Stderr, "\tStarting at %d\n", start)
                }

                prefix[i] = prefixAt(s, prefix, s[i], start)

                if verbose {
                        fmt.Fprintf(os.Stderr, "At %d: %d\n", i, prefix[i])
                }
        }
        return prefix
}

// haystack: the string to search in
// needle: the string to search for
// n_prefix: prefix function for each offset in needle
// initial: initial match length. Useful when chaining multiple calls
// offset: the value to add to matches. Useful when splitting search
func findMatches(
        haystack, needle string,
        n_prefix []int,
        initial int,
        offset int,
) ([]int, int) {

        var matches []int
        prev := initial
        n := len(needle)
        for i := 0; i < len(haystack); i++ {
                real_idx := offset + i

                if verbose {
                        fmt.Fprintf(os.Stderr, "At %d:\n", real_idx)
```

```
                        fmt.Fprintf(os.Stderr, "\tPrev = %d\n", prev)
                }

                prev = prefixAt(needle, n_prefix, haystack[i], prev)

                if verbose {
                        fmt.Fprintf(os.Stderr, "At %d: %d\n", real_idx, prev)
                }

                if prev == n {
                        match_idx := real_idx - n + 1
                        matches = append(matches, match_idx)

                        if verbose {
                                fmt.Fprintf(os.Stderr, "Match at %d-%d\n",
                                        match_idx, real_idx)
                        }
                }
        }
        return matches, prev
}

// --- Find rotation

// Note: for a non-parallel case, hd would be the same string as tl
func findCyclicMatch(hd, tl, orig string, orig_prefix []int) int {
        n := len(orig)

        if verbose {
                fmt.Fprintf(os.Stderr, "Running findMatches in the tail...\n")
        }

        _, last := findMatches(hd, orig, orig_prefix, 0, 0)
        switch last {
        case n:
                if verbose {
                        fmt.Fprintf(os.Stderr,
                                "Full match with tail -> strings are equal\n")
                }
                return 0
        case 0:
                if verbose {
                        fmt.Fprintf(os.Stderr,
                                "No partial match, no need to check further\n")
                }
                return -1
        }

        if verbose {
                fmt.Fprintf(os.Stderr,
                        "Running findMatches in the head with prev=%d...\n",
                        last)
```

```
        }
        matches, _ := findMatches(tl, orig, orig_prefix, last, n)
        if len(matches) != 0 {
                return matches[0]
        } else {
                return -1
        }
}

// --- Parallel implementation

type Range struct {
        offset, length int
}

// len_s: length of haystack
// len_substr: length of needle
// n_ranges: max number of ranges to return
func splitKmpWork(len_s, len_substr, n_ranges int) []Range {
        prematch_area := len_substr - 1

        // match_area: max count of matches
        match_area := len_s - prematch_area

        // In the extreme case, each thread will be checking its own offset.
        if n_ranges > match_area {
                n_ranges = match_area
        }

        //
        match_one_min := match_area / n_ranges
        match_one_rem := match_area % n_ranges

        // Each thread's match area begins at offset + prematch_area.
        offset := 0
        // '+1': we distribute the remainder (r) among the first r processes.
        match_len := match_one_min + 1

        // Allocate beforehand, so no reallocation
        ranges := make([]Range, 0, n_ranges)
        for i := 0; i < n_ranges; i++ {
                if i == match_one_rem {
                        match_len -= 1
                }
                ranges = append(ranges, Range{
                        offset,
                        match_len + prematch_area,
                })
                offset += match_len
        }

        return ranges
```

```
}

func ParallelCheckCyclic(str, orig string, n_pieces int) int {
        if (len(str) != len(orig)) {
                if verbose {
                        fmt.Fprintf(os.Stderr,
                                "Strings of different lengths; quitting\n")
                }

                return -1
        }

        if verbose {
                fmt.Fprintf(os.Stderr,
                        "Calculating prefix function of original string...\n")
        }

        orig_prefix := prefix(orig)

        if verbose {
                fmt.Fprintf(os.Stderr,
                        "Original string's prefix function: %v\n", orig_prefix)
        }

        n := len(orig)
        ranges := splitKmpWork(2*n - 1, n, n_pieces)

        // Spawn a goroutine for each range. Each goroutine will send one number
        // into finish_ch -- either a rotation or -1.
        finish_ch := make(chan int)
        for i, rng := range ranges {
                offset := rng.offset
                end := offset + rng.length - n

                go func () {
                        if verbose {
                                fmt.Fprintf(os.Stderr,
                                        "findCyclicMatch from %d...\n", offset)
                        }

                        rot := findCyclicMatch(
                                str[offset:], str[:end],
                                orig, orig_prefix,
                        )
                        finish_ch <- rot
                }()

                if verbose {
                        fmt.Fprintf(os.Stderr,
                                "Goroutine %d started in range %d to %d\n",
                                i, offset, n+end)
                }
```

```
        }

        // Collect results
        res := -1
        for i := 0; i < len(ranges); i++ {
                x := <-finish_ch

                if verbose {
                        fmt.Fprintf(os.Stderr, "Goroutine finished: %d\n", x)
                }

                if x >= 0 && (res < 0 || x < res) {
                        res = x
                        if verbose {
                                fmt.Fprintf(os.Stderr,
                                        "New leftmost result: %d\n", res)
                        }
                }
        }
        return res
}

func readLine(rd *bufio.Reader) (string, error) {
        str, err := rd.ReadString('\n')
        if err != nil {
                // if err == io.EOF, the string may non-empty and useful
                return str, err
        }

        return strings.TrimSuffix(str, "\n"), nil
}

func main() {
        var n_threads int

        flag.IntVar(
                &n_threads, "j", 1,
                "Number of `threads` to use",
        )
        flag.BoolVar(
                &verbose, "v", false,
                "Show detailed info on algorithm's execution",
        )
        flag.Parse()

        buf_rd := bufio.NewReader(os.Stdin)

        str_a, err := readLine(buf_rd)
        if err != nil {
                log.Fatal(err)
        }
```

```
        str_b, err := readLine(buf_rd)
        if err != nil && err != io.EOF {
                log.Fatal(err)
        }

        res := ParallelCheckCyclic(str_a, str_b, n_threads)

        fmt.Println(res)
}
```