Циклический сдвиг (одномерного) массива.

Суть задачи формулируется просто: на вход подается одномерный массив длины N и число K (0 < K < N); требуется сместить все элементы на K позиций влево, причём массив считаем “зацикленным” - перед начальным элементом массива стоит последний. При этом результат должен получиться на том же самом месте. Например, если массив `[1, 2, 3, 4, 5, 6, 7, 8]` сдвинуть циклически на 3 позиции влево, то получим массив `[4, 5, 6, 7, 8, 1, 2, 3]` .

Задача прекрасна сама по себе, но приятно, что она имеет очевидный практический смысл - ведь нам очень часто необходимо переставить более или менее значительный по размеру кусок данных в другое место, да хотя бы переместить кусок текста в текстовом редакторе - а ведь циклический сдвиг как раз и состоит в том, что мы меняем местами два последовательных куска массива данных.

Задача классическая, так что о ней в сети понаписано много где, естественно в большинстве своём понаписано всякого дерьма, так что я лучше дам нормальный линк. На этом линке более или менее подробно изложен материал главы 2 книги *Дж. Бентли. Жемчужины программирования”. Я помещаю pdf-ку с фрагментом книги прямо здесь же (рядом с текущим файлом), так что картинки можно смотреть в нём. Во второй главе, помимо задачи о циклическом сдвиге, речь идёт и о половинном делении, и довольно много говорится о пользе сортировки (но не об алгоритмах сортировки - о них подробно написано в другом месте этой же книги). О сортировке мы сегодня не говорим, но у меня не повернулась рука резать эту главу, уж больно хороша!

Я тоже буду, в основном, придерживаться вышеупомянутой книги, но изложу алгоритмы поподробнее и изменю порядок изложения.

1. сдвиг через перестановку блоков

Цитата из Бентли: “*Можно предложить и другой алгоритм, который возникает из рассмотрения задачи с другой точки зрения. Циклический сдвиг массива x сводится фактически к замене AB на BA, где A— первые K элементов массива, а B — оставшиеся элементы. Предположим, что A короче B. Разобьем B на B_{left} и B_{right}, где B_{right} содержит K элементов (столько же, сколько и A). Поменяем местами A и B_{right}, получим $B_{right}B_{left}A$. При этом A окажется в конце массива — там, где и полагается. Поэтому можно сосредоточиться на перестановке B_{right} и B_{left}. Эта задача сводится к начальной, поэтому алгоритм можно вызывать рекурсивно. Программа, реализующая этот*

алгоритм, будет достаточно красивой, но она требует аккуратного написания кода, а оценить ее эффективность непросто“.

Эта цитата всё-таки требует некоторых комментариев. Во-первых, рекурсивный вызов в данном случае совершенно ни к чему - процесс чудеснейшим и естественнейшим образом реализуется циклом. Во-вторых, сложность анализа эффективности преувеличена катастрофически - там всё совсем просто, но об этом чуть позже. В-третьих, ничего не сказано об окончании процесса, а он закончится тогда, когда A и B будут иметь одинаковую длину. Этот момент непременно настанет, об этом чуть позже - в анализе алгоритма. А пока я нарисую схемку, иллюстрирующую вышеописанный подход:

```
|---------------|---------------------------|---------------|
| A  A  A  A  A | BL BL BL BL BL BL BL BL BL | BR BR BR BR BR |     ------->
|---------------|---------------------------|---------------|

|---------------|---------------------------|---------------|
| BR BR BR BR BR | BL BL BL BL BL BL BL BL BL | A  A  A  A  A |
|---------------|---------------------------|---------------|
```

А вот что будет в случае, если часть A длиннее части B:

```
|---------------|---------------------------|---------------|
| AL AL AL AL AL | AR AR AR AR AR AR AR AR AR | B  B  B  B  B |     ------->
|---------------|---------------------------|---------------|

|---------------|---------------------------|---------------|
| B  B  B  B  B | AR AR AR AR AR AR AR AR AR | AL AL AL AL AL |
|---------------|---------------------------|---------------|
```

После чего остаётся переставить местами AL и AR.

И теперь код

```
//shift_1a.go
package main

import  "fmt"

func swap(x []int, start1 int, start2 int, length int)  {
    i1, i2:= start1, start2
```

```
    for i:= 0; i < length; i++  {
        x[i1], x[i2] = x[i2], x[i1]
        i1++
        i2++
    }
}

func shift(x []int, left0 int, right0 int, rightK int)  {
//    x - изменяемый массив
//    left0 - начало левого фрагмента
//    right0 - начало правого фрагмента
//    rightK - конец правого фрагмента
    for  {
        lengthL := right0 - left0  // длина левой части
        lengthR := rightK - right0 + 1  // длина правой части
        if lengthL < lengthR  {  // левая часть короче правой
            swap(x, left0, rightK - lengthL + 1, lengthL)
            rightK = rightK - lengthL
        }  else
        if lengthL > lengthR  {  // левая часть длиннее правой
            swap(x, left0, right0, lengthR)
            left0 = left0 + lengthR
        }  else  {
        // длины частей равны
            swap(x, left0, right0, lengthL)
            break
        }
    }
}

func main()  {
    var  (
        n, k int
    )

    fmt.Print("Введите длину массива: ")
    fmt.Scanln(&n)
    fmt.Print("Введите величину сдвига: ")
    fmt.Scanln(&k)

    x:= make([]int, n, n)
```

```
     for i:= 0; i < n; i++  {
        x[i] = i+1
    }
    fmt.Println(x)
    shift(x, 0,  k, n-1)
    fmt.Println(x)
}
```

Рассмотрим ещё один вариант того же самого подхода. В первом варианте мы меняли местами крайние части одинаковой длины. Попробуем теперь переставлять крайнюю часть со средней частью. Не буду расписывать словами, приведу схемы:

```
|---------------|---------------|---------------------------|
| A  A  A  A  A | BL BL BL BL BL | BR BR BR BR BR BR BR BR BR |     ------->
|---------------|---------------|---------------------------|

|---------------|---------------|---------------------------|
| BL BL BL BL BL | A  A  A  A  A | BR BR BR BR BR BR BR BR BR |
|---------------|---------------|---------------------------|
```

В результате часть BL встала на своё место, а нам остаётся переставить части A и BR.
А вот что будет, если часть A длиннее части B:

```
|---------------|---------------------|---------------------|
| AL AL AL AL AL | AR AR AR AR AR AR AR | B  B  B  B  B  B  B  |     ------->
|---------------|---------------------|---------------------|

|---------------|---------------------|---------------------|
| AL AL AL AL AL | B  B  B  B  B  B  B  | AR AR AR AR AR AR AR |
|---------------|---------------------|---------------------|
```

и остаётся перетавить AL и B.

Код, разумеется почти не изменился:

```
//shift_1b.go
package main

import  "fmt"
```

```
func swap(x []int, start1 int, start2 int, length int)  {
    i1, i2:= start1, start2
    for i:= 0; i < length; i++  {
        x[i1], x[i2] = x[i2], x[i1]
        i1++
        i2++
    }
}

func shift(x []int, left0 int, right0 int, rightK int)  {
//    x - изменяемый массив
//    left0 - начало левого фрагмента
//    right0 - начало правого фрагмента
//    rightK - конец правого фрагмента
    for  {
        lengthL := right0 - left0  // длина левой части
        lengthR := rightK - right0 + 1  // длина правой части
        if lengthL < lengthR  {  // левая часть короче правой
            swap(x, left0, right0, lengthL)                 // !!!
            left0, right0 = right0, rightK - lengthL + 1    // !!!
        }  else
        if lengthL > lengthR  {  // левая часть длиннее правой
            swap(x, left0, right0, lengthR)
            right0, rightK = left0 + lengthR, right0-1      // !!!
        }  else  {
        // длины частей равны
            swap(x, left0, right0, lengthL)
            break
        }
    }
}

func main()  {
    var  (
        n, k int
    )

    fmt.Print("Введите длину массива: ")
    fmt.Scanln(&n)
    fmt.Print("Введите величину сдвига: ")
    fmt.Scanln(&k)
```

```
    x:= make([]int, n, n)

    for i:= 0; i < n; i++  {
        x[i] = i+1
    }
    fmt.Println(x)
    shift(x, 0,  k, n-1)
    fmt.Println(x)
}
```

Отличия в коде минимальные - изменились только три строки, они отмечены комментарием // !!!

Анализ алгоритма перестановки блоков. А анализ прост до изумления. Каждый раз, когда мы меняем местами два элемента (в функции swap) один из них становится на свое окончательное место и после этого уже никогда не двигается. Так что весь алгоритм потребует не более N операций обмена двух элементов. Точнее говоря, даже меньше, поскольку при последнем вызове функции swap (когда длины частей равны) при каждом обмене на место становятся сразу оба элемента.

А теперь я приведу ещё один алгоритм, который у Бентли называется “Алгоритм #3: переворотами”.

2. Циклический сдвиг массива переворотами

Не буду ничего придумывать в данном случае, а просто процитирую Бентли:
начало цитаты
Задача кажется сложной, пока вас не осенит озарение («ага!»): итак, нужно преобразовать массив AB в BA. Предположим, что у нас есть функция reverse, переставляющая элементы некоторой части массива в противоположном порядке. В исходном состоянии массив имеет вид AB. Вызвав эту функцию для первой части, получим A_rB (прим. редактора: A_r - это модифицированная часть A, к которой применили фукнцию перестановки reverse). Затем вызовем ее для второй части: получим A_rB_r. Затем вызовем функцию для всего массива, что даст $(A_rB_r)_r$, а это в точности соответствует BA. Посмотрим, как будет такая функция действовать на массив abcdefgh, который нужно сдвинуть влево на три элемента:
псевдокод: Сдвиг через функцию перестановки reverse

```
1.    reverse(0, k-1)   /* cba|defgh */
2.    reverse(k, n-1)   /* cba|hgfed */
```

```
3.    reverse(0, n-1)   /* defgh|abc */
```

Дуг Макилрой (Doug McIlroy) предложил наглядную иллюстрацию циклического сдвига массива из десяти элементов вверх на пять позиций (рис. 2.3); начальное положение: обе руки ладонями к себе, левая над правой: картинка находится в файле reverse.png

Код, использующий функцию переворота, оказывается эффективным и малотребовательным к памяти, и настолько короток и прост, что при его реализации сложно ошибиться.

Б. Керниган и П. Дж. Плоджер пользовались именно этим методом для перемещения строк в текстовом редакторе в своей книге (B. Kernighan, P. J. Plauger, Software Tools in Pascal, 1981). Керниган пишет, что эта функция заработала правильно с первого же запуска, тогда как их предыдущая версия, использовавшая связный список, содержала несколько ошибок. Этот же код используется в некоторых текстовых редакторах, включая тот, в котором я впервые набрал настоящую главу. Кен Томпсон (Ken Thompson) написал этот редактор с функцией reverse в 1971 году, и он утверждает, что она уже тогда была легендарной…"
конец цитаты
Реализацию писать не стану - всё совсем просто и ясно, так что оставляем её в качестве **упражнения**.

3. последовательный сдвиг по одному элементу

Рассмотрим алгоритм из много цитировавшейся книги Дж. Бентли "Жемчужины программирования" (п. 2.3). В изложении считается, что нумерация массива начинается с 0.
начало цитаты
Алгоритм #1: последовательный обмен
Одним из вариантов решения будет введение дополнительной переменной. Элемент x_0 помещается во временную переменную t, затем x_k помещается в x_0,x_{2*k} — в x_k и так далее (перебираются все элементы массива x с индексом по модулю n). *примечание от меня: я боюсь, что здесь какая-то типографская накладка, явно имеется в виду "с индексом pk, p = 0, 1, 2,… по модулю n")*, пока мы не возвращаемся к элементу x_0, вместо которого записывается содержимое переменной t, после чего процесс завершается. Если i = 3, а n = 12, этот этап проходит следующим образом: см. рисунок

Если при этом не были переставлены все имеющиеся элементы, процедура повторяется, начиная с x[1] и так далее, до достижения конечного результата.
конец цитаты

Есть и ещё один рисунок, который иллюстрирует процесс для N=15, K=3.

Попробую изложить этот алгоритм поподробнее, заодно и поймём, откуда у Бентли в коде (я его здесь не привожу, но в материалах к предыдущему занятию есть полный текст соответствующей главы из книги) появляется слово gcd.

Давайте начнём вышеописанный процесс для N=15, K=3. Тогда 3-й элемент перейдёт на 0-е место, 6-й - на 3-е, 9-й - на 6-е, 12-й - на 9-е, и, наконец, 0-й - на 12-е. Переехали на свои новые места только 5 элементов. Повторяем процесс, начиная с элемента на 1-ом месте. В результате последовательно встанут на свои новые места 4-й, 7-й, 10-й, 13, и 1-й элементы. Наконец, совершив такой последовательный обмен, начиная со 2-го элемента, поставим на свои новые места 5-й, 8-й, 11-й, 14-й и 2-й элементы.

Интересное наблюдение: каждый раз (в данном примере каждый из трёх раз) процесс последовательного обмена останавливается на том элементе, с которого он начался. Почему так происходит, почему мы никогда не натыкаемся на уже обработанный элемент из другого внутреннего цикла? Ответ на этот вопрос легко виден из приводимого рисунка. В каждую ячейку ведут ровно два отрезка: по одному из них в эту ячейку “приехал новый жилец”, по другому - “уехал старый”. Понятно (просто из задачи), что других отрезков ни в какую ячейку не входит-выходит. И вот мы выходим из какой-то ячейки. Каждый раз мы входим в новую ячейку и выходим из неё, зайти в уже посещённую ячейку мы не можем - это будет третий отрезок, ведущий в неё. С другой стороны процесс должен закончится - мы на каждом шаге посещаем новую ячейку, а они в конце концов закончатся. Но закончить рисование нашей ломаной линии мы можем только одним единственным способом - вернуться в начальную вершину, в ту, из которой начался последовательный обмен.

Итак, первый цикл закончен, на рисунке ему соответствует замкнутая ломаная 1-4-7-10-13-1. Если в массиве остались еще не отмеченные ячейки, то среди них обязательно будет и ячейка номер 2. Иначе, если наш цикл привел нас из первой ячейки во вторую, то он, прежде, чем замкнуться, должен привести нас в третью, из третьей - в четвертую, и т.д., т.е. свободных ячеек не останется. В самом деле, просто повернём кружок так, чтобы двойка встала на место 1. Тогда ломаная будет вести нас из двойки в тройку. Ну, и т.д. Начинаем второй цикл со второй ячейки. Если второй цикл после себя оставит непомеченные ячейки, то выполним третий цикл, который, конечно же, начнется с третьей ячейки, и т.д.

И как долго и т.д.? А пока не переставим всё. Можем просто подсчитывать количество передвинутых элементов и заканчивать тогда, когда переставим N элементов. Этот вариант реализован в программе `shift_3.go` :

```
// shift_3.go
```

```
package main

import  "fmt"

func shift (x []int, k int)  {
    var  (
        tmp, current, next int
        count int = 0
    )

    for start := 0; count < len(x); start++  {
        tmp = x[start]
        current = start
        for  {
            next = (current + k)%len(x)
            if next == start  {
                x[current] = tmp
                count++
                break
            }
            x[current] = x[next]
            count++
            current = next
        }
    }
}

func main()  {
    var  (
        n, k int
    )

    fmt.Print("Введите длину массива: ")
    fmt.Scanln(&n)
    fmt.Print("Введите величину сдвига: ")
    fmt.Scanln(&k)

    x:= make([]int, n, n)
     for i:= 0; i < n; i++  {
        x[i] = i+1
    }
```

```
    fmt.Println(x)
    shift(x, k)
    fmt.Println(x)
}
```