Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Факультет безопасности информационных технологий

Дисциплина:
«Алгоритмы и структуры данных»

Лабораторная работа №2

Выполнил:

Беляков Никита Андреевич N3245

_____
(подпись)

Проверил:

Еврофеев С.А.

_____
(отметка о выполнении)

_____
(подпись)

Санкт-Петербург
2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В данной лабораторной работе мы разработаем программу битонной сортировки, используя стек в качестве основной структуры данных, а также оценим сложность алгоритма сортировки путем подсчета количества элементарных операций.

Для реализации стека будет использоваться структура данных типа struct, обладающая методами для основных операций над стеком, такими как добавление элемента (Push), извлечение элемента (Pop), определение длины стека (Len), сравнение элементов (Compare) и обмен элементов (Swap).

Кроме того, в рамках этой работы будет реализована функция битонной сортировки, которая принимает строку чисел в качестве входных данных, сортирует их с использованием стека и возвращает отсортированную строку чисел в виде результата.

1 ХОД РАБОТЫ

1.1 Блок схема

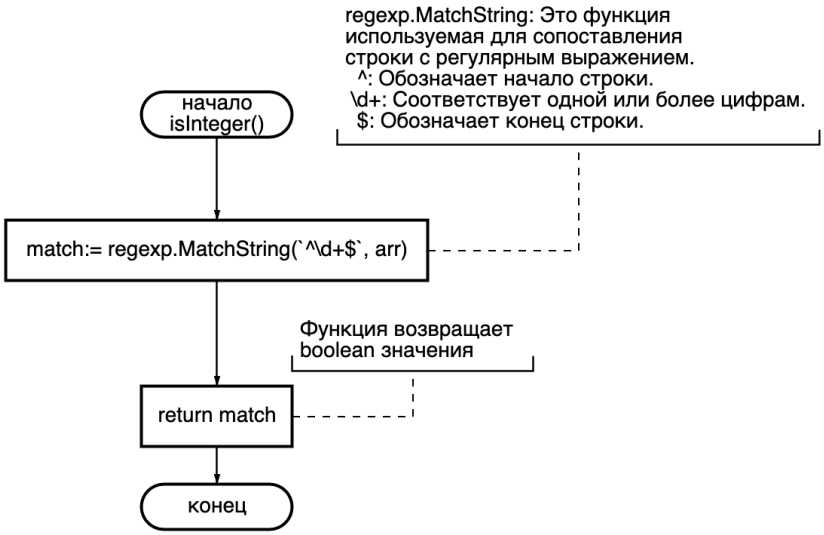


Рисунок 1.1 — Блок схема isInteger().

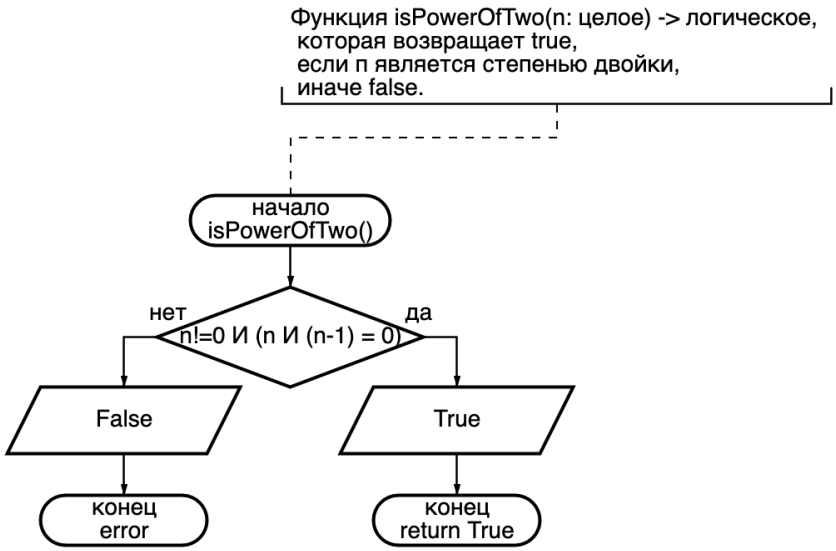
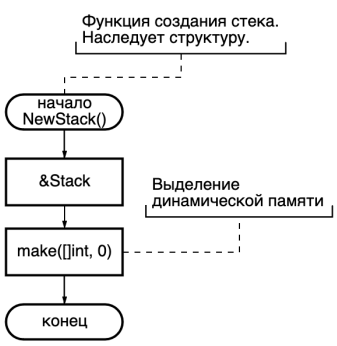
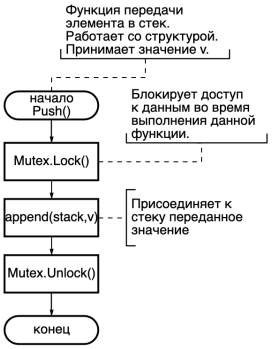


Рисунок 1.2 — Блок схема isPowerOfTwo().



(а) Блок схема NewStack()



(б) Блок схема Push()

Рисунок 1.3

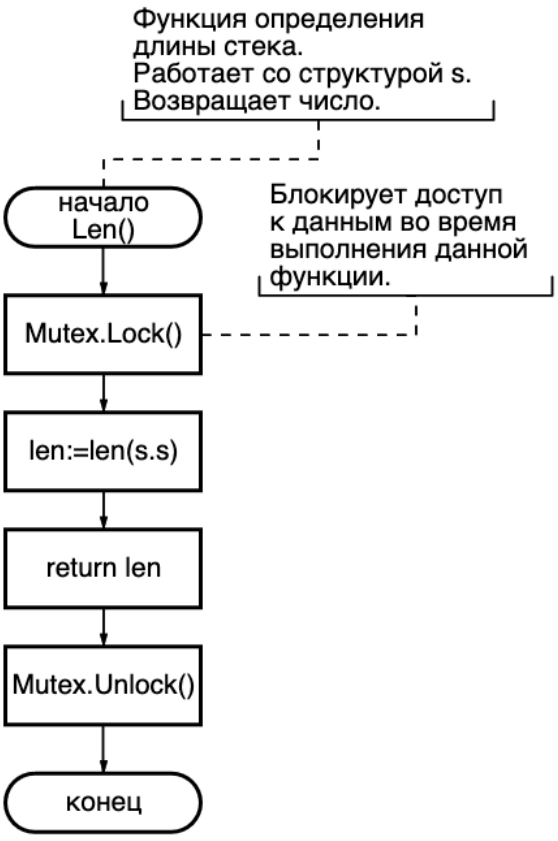


Рисунок 1.4 — Блок схема Len().

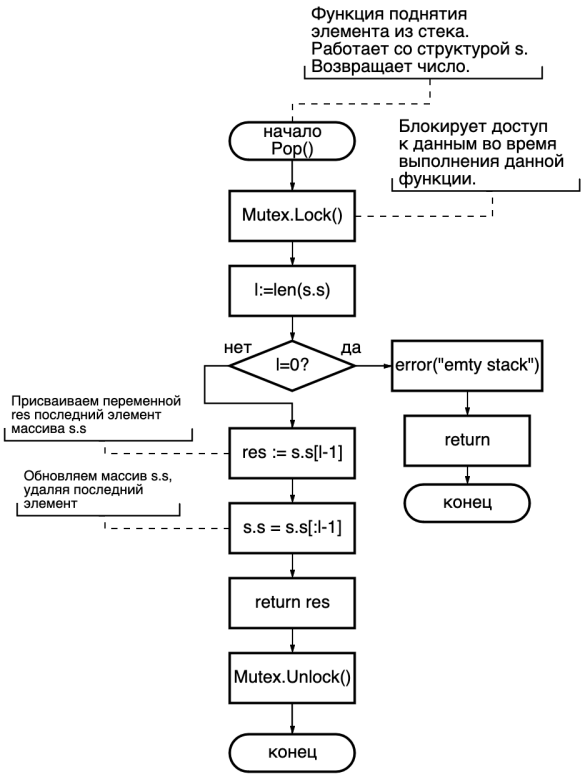


Рисунок 1.5 — Блок схема Pop().

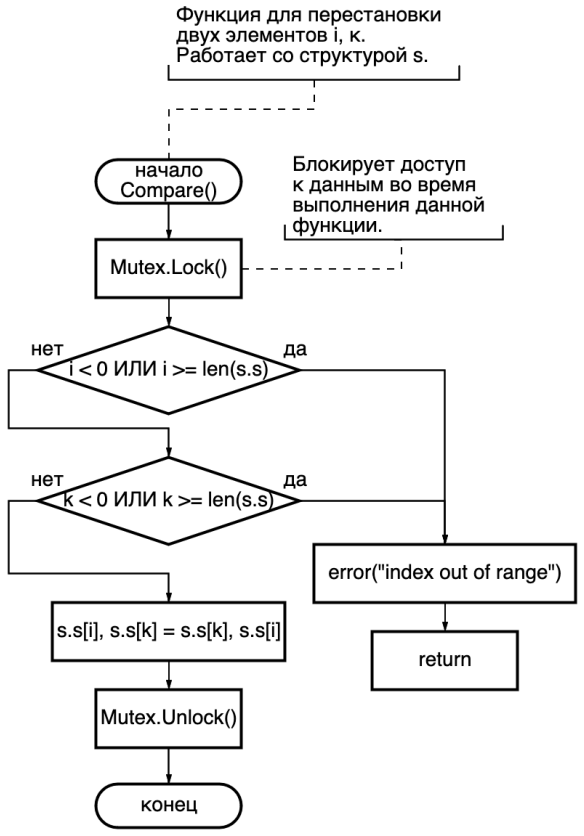


Рисунок 1.6 — Блок схема Swap().

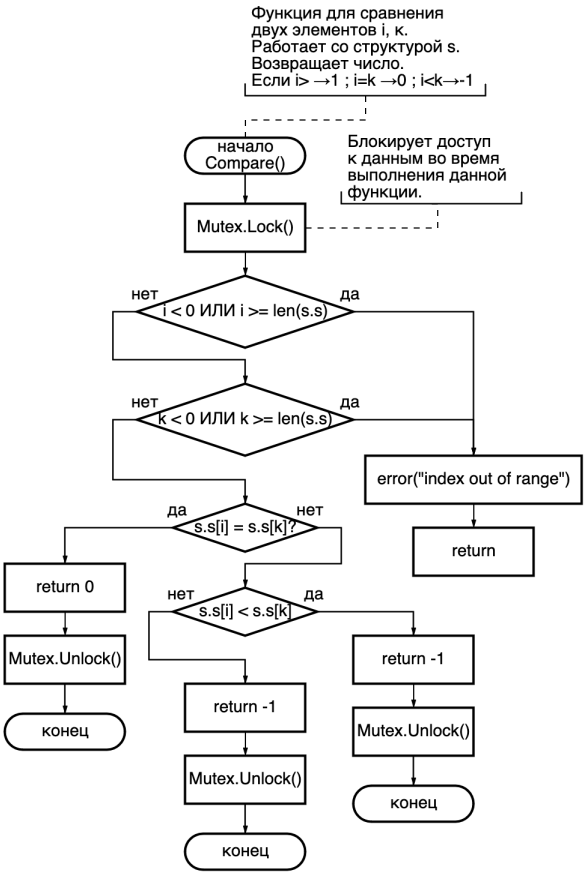


Рисунок 1.7 — Блок схема Compare().

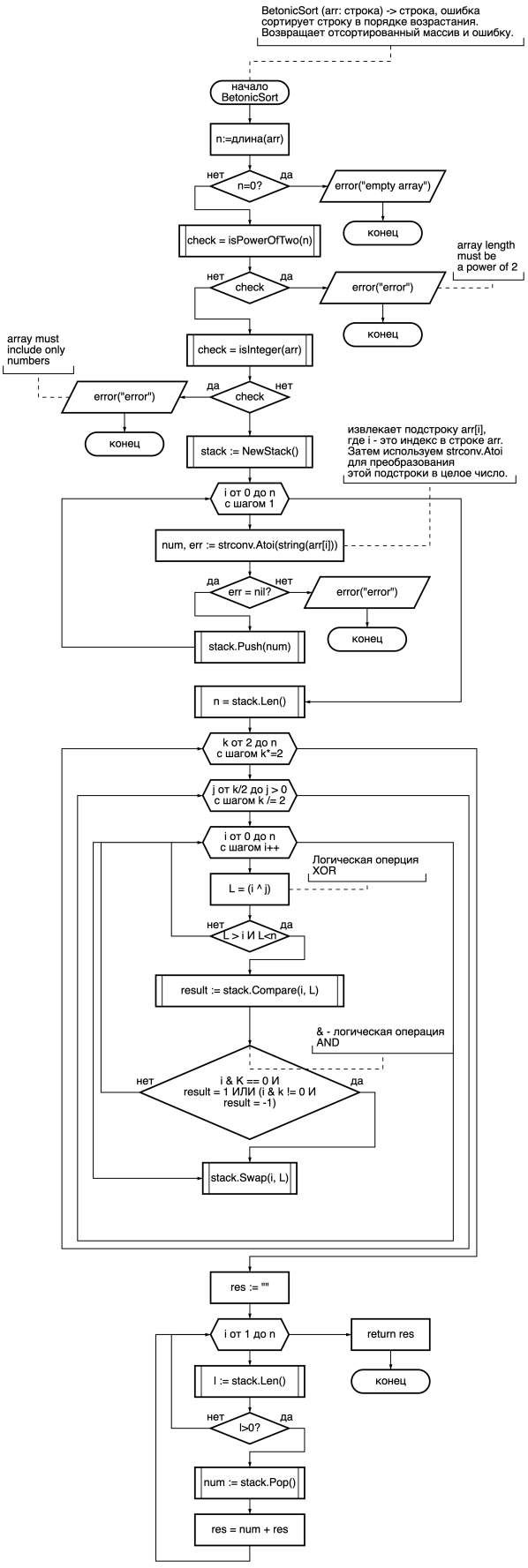


Рисунок 1.8 — Блок схема Betonic Sort.

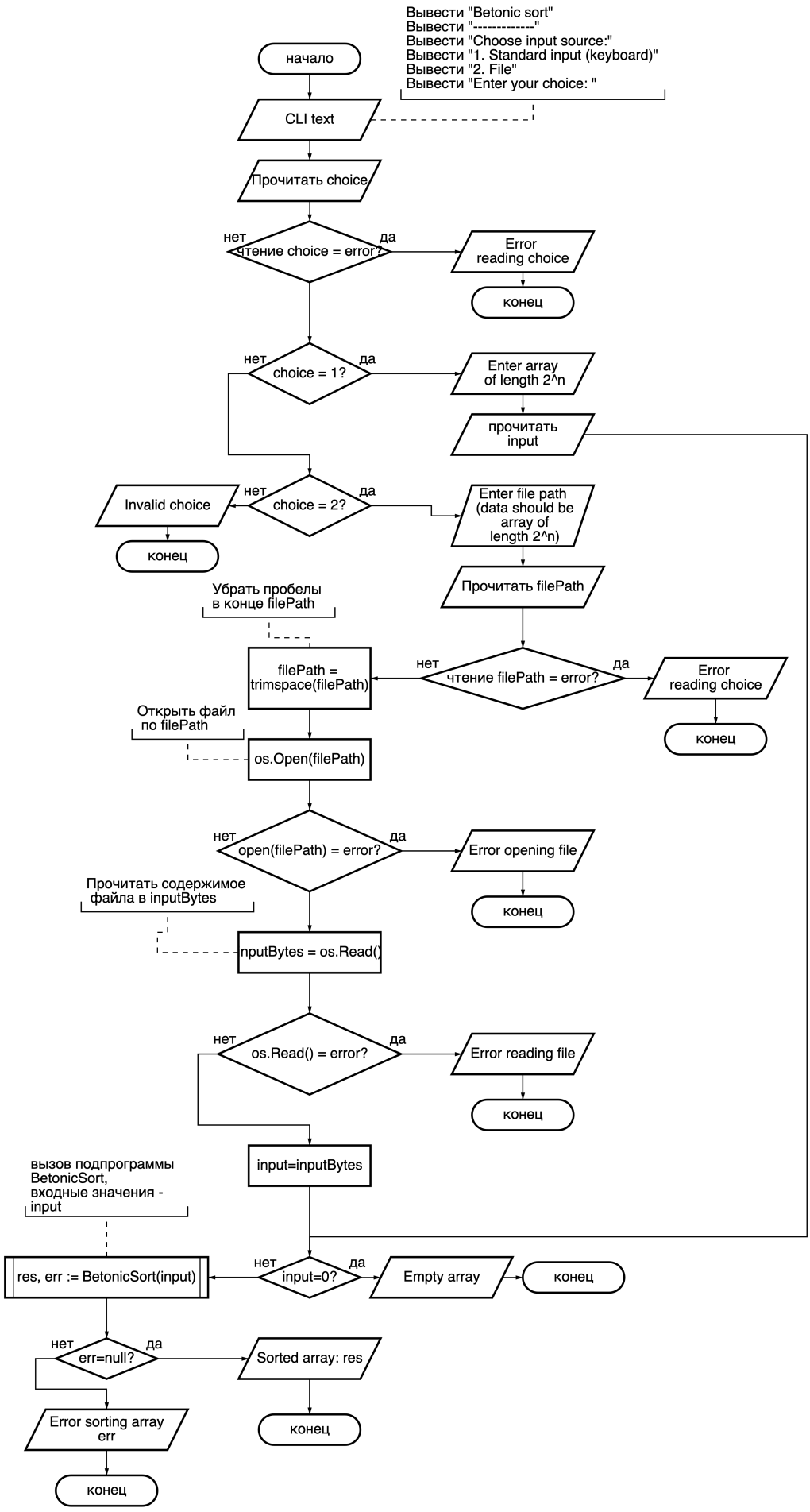


Рисунок 1.9 — Блок схема main.

1.2 Псевдокод

1.2.1 Псевдокод для структуры Stack

```
Структура Stack:
    Поле s - массив целых чисел
    Мьютекс - для обеспечения безопасности доступа к данным

Функция NewStack():
    Возвращает новый экземпляр структуры Stack с пустым массивом

Функция Push(v):
    Захватывает мьютекс
    Добавляет значение v в конец массива s
    Освобождает мьютекс

Функция Pop():
    Захватывает мьютекс
    Получает последний элемент массива s
    Удаляет последний элемент из массива s
    Освобождает мьютекс
    Возвращает полученное значение

Функция Len():
    Захватывает мьютекс
    Возвращает длину массива s
    Освобождает мьютекс

Функция Compare(i, k):
    Захватывает мьютекс
    Проверяет, что i и k не выходят за пределы массива s
    Если s[i] равно s[k], возвращает 0
    Если s[i] меньше s[k], возвращает -1
    Если s[i] больше s[k], возвращает 1
    Освобождает мьютекс

Функция Swap(i, k):
    Захватывает мьютекс
    Проверяет, что i и k не выходят за пределы массива s
    Меняет местами значения s[i] и s[k]
    Освобождает мьютекс
```

1.2.2 Псевдокод для функции BitonicSort

```
Функция BitonicSort(arr):
    n := Длина строки arr

    Если n равно 0, возвращаем ошибку "пустой массив"
    Если n не является степенью двойки, возвращаем ошибку "длина массива должна быть степенью двойки"
    Если arr содержит не только числа, возвращаем ошибку "массив должен содержать только числа"

    stack := Создать новый стек
    Для каждого элемента в строке arr:
        Преобразовать элемент в число и добавить его в стек

    Для k от 2 до n с шагом 2:
        Для j от k/2 до 1 с шагом k/2:
            Для i от 0 до длины стека:
                l := i XOR j
```

```
                Если l > i и l < длины стека:
                    result := Вызвать функцию Compare для элементов i и l
                    Если i & k равно 0 и result равно 1 или i & k не равно 0 и result равно -1:
                        Вызвать функцию Swap для элементов i и l

    res := Пустая строка
    Пока длина стека больше 0:
        num := Извлечь элемент из стека
        Преобразовать num в строку и добавить его в начало строки res

    Вернуть res и nil (результат отсортированной строки и отсутствие ошибок)
```

1.2.3 Псевдокод для функции main

```
Функция main():
    Вывести "Betonic sort"
    Вывести "-------------"
    Вывести "Choose input source:"
    Вывести "1. Standard input (keyboard)"
    Вывести "2. File"

    Чтение выбора пользователя choice

    Если choice равно 1:
        Вывести "Enter array of length 2^n:"
        Чтение строки input с клавиатуры
    Если choice равно 2:
        Вывести "Enter file path (data should be array of length 2^n):"
        Чтение пути к файлу filePath
        Чтение содержимого файла filePath в строку input

    Если длина input равна 0:
        Вывести "Empty array. Exit."
        Завершить программу

    Отсортировать input с помощью функции BitonicSort
    Вывести отсортированный массив
```

1.3 Маршрут

Маршрут выполнения программы на основе данного кода.

1.3.1 Маршрут для main.go

– Программа выводит на экран меню выбора источника ввода.
– Пользователь делает выбор между стандартным вводом (клавиатурой) и файлом.
– В зависимости от выбора, программа либо считывает массив с клавиатуры, либо запрашивает путь к файлу и считывает массив из файла.
– Проверяется, не является ли массив пустым. Если да, программа завершается с сообщением об ошибке.
– Проверяется, является ли длина массива степенью двойки. Если нет, программа завершается с сообщением об ошибке.

– Происходит вызов функции BetonicSort из пакета betonicSortStack для сортировки массива.
– Отсортированный массив выводится на экран.

1.3.2 Маршрут для betonicSort

– Принимает входной массив чисел в виде строки.
– Проверяет, не является ли массив пустым.
– Проверяет, является ли длина массива степенью двойки.
– Создает новый стек.
– Помещает элементы массива в стек.
– Производит битоническую сортировку элементов в стеке.
– Формирует отсортированную строку из элементов стека.
– Возвращает отсортированную строку и, при наличии ошибок, соответствующее сообщение об ошибке.

1.4 Спецификация переменных

Ниже представлена спецификация переменных.

Переменная	Тип	Минимум	Максимум	Значение
arr	string	0	2^{31}	входной массив чисел
n	int	-2147483648	2147483647	длина входного массива
choice	int	-2147483648	2147483647	выбор пользователя
input	string	0	2^{31}	входные данные (массив)
filePath	string	0	2^{31}	путь к файлу (если выбрано чтение из файла)
sortedArr	string	0	2^{31}	отсортированный массив чисел
res	string	0	2^{31}	результат сортировки
i, j, k, l	int	-2147483648	2147483647	индексы для итерации в алгоритме сортировки
num	int	-2147483648	2147483647	число для преобразования из строки
s	struct	–	–	стек для хранения чисел

1.5 Листинг программы

1.5.1 betonicSortStack.go

Ниже представлен листинг файла betonicSortStack.go

```
package betonicSortStack

import (
        "errors"
        "regexp"
        "strconv"

        "github.com/17HIERARCH70/stackGo"
)

// Check if string is an integer
func isInteger(arr string) bool {
        match, _ := regexp.MatchString(`^\d+$`, arr)
```

```
        return match
}

// Check if number is a power of 2
func isPowerOfTwo(n int) bool {
        return (n != 0) && (n&(n-1) == 0)
}

// Betonic sort for integers
func BitonicSort(arr string) (string, error) {
        n := len(arr)

        if n == 0 {
                return "", errors.New("empty array")
        }

        if !isPowerOfTwo(n) {
                return "", errors.New("array length must be a power of 2")
        }

        if !isInteger(arr) {
                return "", errors.New("array must include only numbers")
        }

        // Create a stack struct
        stack := stackGo.NewStack()

        // Push array to stack
        for i := 0; i < n; i++ {
                num, err := strconv.Atoi(string(arr[i]))
                if err != nil {
                        return "", err
                }
                stack.Push(num)
        }

        // Betonic sort for stack
        for k := 2; k <= n; k *= 2 {
                for j := k / 2; j > 0; j /= 2 {
                        for i := 0; i < stack.Len(); i++ {
                                l := i ^ j
                                if l > i && l < stack.Len() {
                                        result, err := stack.Compare(i, l)
                                        if err != nil {
                                                return "", err
                                        }

                                        if (i&k == 0) && (result == 1) || (i&k != 0) && (result == -1) {
                                                err := stack.Swap(i, l)
                                                if err != nil {
                                                        return "", errors.New("error with swapping")
                                                }
                                        }
                                }
                        }
                }
        }
        res := ""

        // Pop elements from the stack to construct the result
        for stack.Len() > 0 {
                num, _ := stack.Pop()
                res = strconv.Itoa(num) + res
```

```
        }

        // Return the sorted result as a string
        return res, nil
}
```

1.5.2 main.go

Ниже представлен листинг файла main.go

```
package main

import (
        "algosITMO/lab3/betonicSortStack"
        "bufio"
        "fmt"
        "io"
        "log"
        "os"
        "strings"
)

func main() {
        // CLI UI
        fmt.Println("Betonic sort")
        fmt.Println("-------------")
        fmt.Println("Choose input source:")
        fmt.Println("1. Standard input (keyboard)")
        fmt.Println("2. File")

        reader := bufio.NewReader(os.Stdin)

        var choice int
        fmt.Print("Enter your choice: ")
        _, err := fmt.Scan(&choice)
        if err != nil {
                log.Fatal("Error reading choice: ", err)
        }

        var input string

        switch choice {
        case 1: // read from keyboard
                fmt.Println("Enter array of length 2^n:")
                input, err = reader.ReadString('\n')
                if err != nil {
                        log.Fatal("Error reading from keyboard: ", err)
                }
                input = strings.TrimSpace(input)
        case 2: // read from file
                fmt.Print("Enter file path (data should be array of length 2^n): ")
                filePath, err := reader.ReadString('\n')

                if err != nil {
                        log.Fatal("Error reading file path: ", err)
                }
                filePath = strings.TrimSpace(filePath) // remove trailing newline character

                file, err := os.Open(filePath)
                if err != nil {
                        log.Fatal("Error opening file: ", err)
```

```
                }
                // close file on exit
                defer func() {
                        if err := file.Close(); err != nil {
                                log.Fatal("Error closing file: ", err)
                        }
                }()
                inputBytes, err := io.ReadAll(file)
                if err != nil {
                        log.Fatal("Error reading file: ", err)
                }
                input = string(inputBytes)

        default: // invalid choice
                log.Fatal("Invalid choice")
        }

        if len(input) == 0 {
                fmt.Println("Empty array. Exit.")
                return
        }

        // sort array
        sortedArr, err := betonicSortStack.BitonicSort(input)
        if err != nil {
                log.Fatal("Error sorting array: ", err)
        }

        // print sorted array
        fmt.Println("Sorted array:", sortedArr)
}
```

1.5.3 stackGo.go

Ниже представлен листинг файла stackGo.go

```
package stackGo

import (
        "errors"
        "sync"
)

type Stack struct {
        sync.Mutex // Embed the mutex for easier locking
        s          []int
}

func NewStack() *Stack {
        return &Stack{s: make([]int, 0)}
}

func (s *Stack) Push(v int) {
        s.Lock()
        defer s.Unlock()

        s.s = append(s.s, v)
}

func (s *Stack) Pop() (int, error) {
        s.Lock()
```

```
        defer s.Unlock()

        l := len(s.s)
        if l == 0 {
                return 0, errors.New("empty stack")
        }

        res := s.s[l-1]
        s.s = s.s[:l-1]
        return res, nil
}

func (s *Stack) Len() int {
        s.Lock()
        defer s.Unlock()

        return len(s.s)
}

func (s *Stack) Compare(i, k int) (int, error) {
        s.Lock()
        defer s.Unlock()

        if i < 0 || i >= len(s.s) || k < 0 || k >= len(s.s) {
                return 0, errors.New("index out of range")
        }

        if s.s[i] == s.s[k] {
                return 0, nil
        } else if s.s[i] < s.s[k] {
                return -1, nil
        } else {
                return 1, nil
        }
}

func (s *Stack) Swap(i, k int) error {
        s.Lock()
        defer s.Unlock()

        if i < 0 || i >= len(s.s) || k < 0 || k >= len(s.s) {
                return errors.New("index out of range")
        }

        s.s[i], s.s[k] = s.s[k], s.s[i]
        return nil
}
```

case1	case2	case3	case4
47384920	3647383917253679	9443	35748234
Step 0: 47834920	Step 0: 3674383917253679	Step 0: 4943	Step 0: 35742834
Step 1: 43874920	Step 1: 3674389317253679	Step 1: 4349	Step 1: 35742843
Step 2: 34874920	Step 2: 3674389317523679	Step 2: 3449	Step 2: 34752843
Step 3: 34784920	Step 3: 3674389317523697	Sorted array: 3449	Step 3: 34754823
Step 4: 34789420	Step 4: 3476389317523697		Step 4: 34574823
Step 5: 34289470	Step 5: 3476983317523697		Step 5: 34578423
Step 6: 34209478	Step 6: 3476983312573697		Step 6: 34578432
Step 7: 24309478	Step 7: 3476983312579637		Step 7: 34378452
Step 8: 20349478	Step 8: 3476983312579736		Step 8: 34328457
Step 9: 20347498	Step 9: 3467983312579736		Step 9: 32348457
Step 10: 02347498	Step 10: 3467983312579763		Step 10: 32345487
Step 11: 02344798	Step 11: 3437986312579763		Step 11: 23345487
Step 12: 02344789	Step 12: 3433986712579763		Step 12: 23344587
Sorted array: 02344789	Step 13: 3433986792571763		Step 13: 23344578
	Step 14: 3433986797571263		Sorted array: 23344578
	Step 15: 3433986797671253		
	Step 16: 3334986797671253		
	Step 17: 3334689797671253		
	Step 18: 3334679897671253		
	Step 19: 3334679897675213		
	Step 20: 3334679897675312		
	Step 21: 3334678997675312		
	Step 22: 3334678997765312		
	Step 23: 3334678997765321		
	Step 24: 3334578997766321		
	Step 25: 3334538997766721		
	Step 26: 3334532997766781		
	Step 27: 3334532197766789		
	Step 28: 3324533197766789		
	Step 29: 3321533497766789		
	Step 30: 3321533467769789		
	Step 31: 2331533467769789		
	Step 32: 2133533467769789		
	Step 33: 2133335467769789		
	Step 34: 2133335466779789		
	Step 35: 2133335466778799		
	Step 36: 1233335466778799		
	Sorted array: 1233334566778799		

1.7 Расчет сложности алгоритма

1.7.1 Проверка на целое число

Функция для проверки строки на целое число в Go:

```
func isInteger(arr string) bool {
    match, _ := regexp.MatchString(`^\d+$`, arr)
    return match
}
```

Сложность данной функции оценивается константно, так как `regexp.MatchString` в Go выполняется следующим образом:

```
func (re *Regexp) MatchString(s string) bool {
    return re.doMatch(nil, nil, s)
```

```
}
```

Функция `regexp.MatchString` сравнивает строку с регулярным выражением, проходя по строке `arr` и сравнивая ее с заданным регулярным выражением. Длина строки обозначается как n, таким образом, сложность операции будет $O(n)$.

1.7.2 Проверка на степень двойки

```
func isPowerOfTwo(n int) bool {
    return (n != 0) && (n & (n-1) == 0)
}
```

Анализ операций:

- Сравнение $n \neq 0$ — 1 операция.
- Вычисление $n - 1$ — 1 операция.
- Побитовое <И> — 1 операция.
- Сравнение результата — 1 операция.

Таким образом, общее количество операций равно 4.

1.7.3 Алгоритм сортировки

```
func BitonicSort(arr string) (string, error) {
    n := len(arr)
    ...
}
```

Эта операция выполняется за время $O(1)$, так как для получения длины строки необходимо только прочитать уже имеющуюся информацию о длине строки, что выполняется за постоянное время.

```
stack := stackGo.NewStack()

func NewStack() *Stack {
    return &Stack{s: make([]int, 0)}
}
```

1. `make([]int, 0)`: Это операция создания среза типа `int` с начальной емкостью 0. Это простая операция, выполняемая за константное время `O(1)`.
2. Возврат указателя на структуру `Stack`: Это также простая операция, которая выполняется за константное время `O(1)`.

Таким образом, общая сложность функции `NewStack` составляет $O(1)$, поскольку ее сложность не зависит от размера данных.

```
func (s *Stack) Push(v int) {
    s.Lock()
    defer s.Unlock()

    s.s = append(s.s, v)
}
```

1. `s.Lock()`: Это операция блокировки мьютекса. Константа.
2. `append(s.s, v)`: Это операция добавления элемента в срез. Если есть достаточно места в срезе, то это обычно занимает амортизированное константное время O(3).

Итак, общая сложность метода `Push` - O(3)
Пройдем по основному алгоритму:

```
for i := 0; i < n; i++ {
    num, err := strconv.Atoi(string(arr[i]))
    if err != nil {
       return "", err
    }
    stack.Push(num)
}
```

1. Цикл `for`: Этот цикл выполняется `n` раз, где `n` - длина строки `arr`. Следовательно, это требует времени O(n+3).
2. `strconv.Atoi(string(arr[i]))`: Это операция преобразования символа в строке в целое число. Это выполняется за время O(1), так как мы работаем только с одним символом.
3. Проверка , умножаем на n.
4. `stack.Push(num)`: Как мы обсудили ранее, операция `Push` добавления элемента в стек имеет сложность, близкую к O(1), с учетом блокировки и освобождения мьютекса.

```
func (s *Stack) Compare(i, k int) (int, error) {
    s.Lock() //1
    defer s.Unlock() //1
    //7
    if i < 0 || i >= len(s.s) || k < 0 || k >= len(s.s) {
        return 0, errors.New("index out of range")
    }
    //1
    if s.s[i] == s.s[k] {
        return 0, nil
    } else if s.s[i] < s.s[k] { //1
        return -1, nil
    } else {
        return 1, nil
    }
}
```

В функции `Compare()` выполняется 11 простейших операций, сложность которых O(11).

```
func (s *Stack) Swap(i, k int) error {
    s.Lock() //1
    defer s.Unlock() //1
    //7
    if i < 0 || i >= len(s.s) || k < 0 || k >= len(s.s) {
        return errors.New("index out of range")
    }
    //2
    s.s[i], s.s[k] = s.s[k], s.s[i]
    return nil
}
```

В функции `Swap()` выполняется 11 простейших операций, сложность которых O(11).

```
// Betonic sort for stack
for k := 2; k <= n; k *= 2 {
    for j := k / 2; j > 0; j /= 2 {
        for i := 0; i < stack.Len(); i++ {
            l := i ^ j
            if l > i && l < stack.Len() {
                result, err := stack.Compare(i, l)
                if err != nil {
                    return "", err
                }

                if (i&k == 0) && (result == 1) || (i&k != 0) && (result == -1) {
                    err := stack.Swap(i, l)
                    if err != nil {
                        return "", errors.New("error with swapping")
                    }
                }
            }
        }
    }
}
```

Давайте разберем каждую часть кода:

1. Внешний цикл: Этот цикл выполняется $\log_2 n$ раз, где n - длина стека. На каждой итерации размер сравниваемых последовательностей удваивается. Следовательно, этот цикл имеет сложность $O(\log n)$.
2. Первый внутренний цикл: Этот цикл выполняется $\log_2 \frac{k}{2}$ раз, где k - текущий размер сравниваемых последовательностей. На каждой итерации размер сравниваемых последовательностей уменьшается вдвое. Следовательно, сложность этого цикла также $O(\log n)$.
3. Второй внутренний цикл: Этот цикл выполняется O(n) раз, так как он проходит через все элементы стека. Следовательно, сложность этого цикла O(n).
4. Внутренние операции: В этом блоке выполняются следующие операции:
   - Четыре if statement. Умножаем на 4.
   - Вызов метода `stack.Compare(i, l)`: Этот вызов имеет сложность O(11), так как он просто сравнивает два элемента стека.
   - Вызов метода `stack.Swap(i, l)`: Этот вызов также имеет сложность O(11), так как он просто меняет местами два элемента стека.

Таким образом, общая сложность этого кода составляет $O((\log n)^2 * 4n + 11)$.

```
func (s *Stack) Pop() (int, error) {
    s.Lock()
    defer s.Unlock()

    l := len(s.s)
    if l == 0 {
        return 0, errors.New("empty stack")
    }

    res := s.s[l-1]
    s.s = s.s[:l-1]
    return res, nil
```

```
}
```

Метод `Pop` удаляет и возвращает последний элемент из стека.

1. `s.Lock()`: Это операция блокировки мьютекса для безопасного доступа к стеку. Сложность этой операции зависит от реализации синхронизации, но обычно она достаточно быстрая.
2. `defer s.Unlock()`: Это операция отложенного освобождения мьютекса после завершения функции.
3. `l := len(s.s)`: Получение длины стека. Эта операция выполняется за время O(1), так как длина среза обычно хранится вместе с самим срезом.
4. Проверка пустоты стека: Этот блок проверяет, пуст ли стек. Если стек пуст, возвращается ошибка. Эта проверка выполняется за время O(1).
5. Извлечение элемента из стека: Этот блок извлекает последний элемент из стека. Это выполняется за время O(1), так как срезы в Go позволяют эффективно извлекать последний элемент.
6. Уменьшение размера стека: Этот блок уменьшает размер стека на 1. Это также выполняется за время O(1), так как срезы в Go поддерживают операцию удаления из конца с амортизированной сложностью O(1).

Таким образом, общая сложность метода `Pop` также составляет O(1), а количество простейших операций внутри этого метода оценивается как O(6).

```
for stack.Len() > 0 {
    num, _ := stack.Pop()
    res = strconv.Itoa(num) + res
}
```

Извлечение элементов из стека имеет сложность $O(n)$.

Давайте соберём все слагаемые и упростим выражение:

1. O(1) + O(1) = O(1) (сумма констант)
2. O(n + 3) = O(n) (здесь мы просто оставляем самый большой член)
3. $O((\log n)^2 * 4n) = O(4n \log n^2)$ (оставляем без изменений)
4. O(n + 11) = O(n) (оставляем самый большой член)
5. O(n + 2) = O(n) (оставляем самый большой член)

Теперь объединим все слагаемые: $O(1) + O(1) + 2 + O(1) + O(2n) + O(4nlog(n)^2) + O(3n) = O(4nlog(n))^2$

Итак, итоговая сложность данного фрагмента кода составляет $O(nlog(n)^2)$

1.8 Тесты

1.8.1 betonicSortStackTest.go

```go
package betonicSortStack_test

import (
	"algosITMO/lab3/betonicSortStack"
	"errors"
	"strconv"
	"testing"
)

func TestBetonicSort(t *testing.T) {
	tests := []struct {
		input string
		want  string
		err   error
	}{
		{input: "1234", want: "1234", err: nil},
		{input: "4321", want: "1234", err: nil},
		{input: "1243", want: "1234", err: nil},
		{input: "3214", want: "1234", err: nil},
		{input: "12345678", want: "12345678", err: nil},
		{input: "87654321", want: "12345678", err: nil},
		{input: "12435678", want: "12345678", err: nil},
		{input: "87543212", want: "12234578", err: nil},
		{input: "12a4", want: "", err: errors.New("array must include only numbers")},
		{input: "1232332456789", want: "",
	    err: errors.New("array length must be a power of 2")},
		{input: "", want: "", err: errors.New("empty array")},
	}
	for _, test := range tests {
		got, err := BitonicSort(test.input)
		if (err == nil && test.err != nil)
	    || (err != nil && test.err == nil)
	    || (err != nil && test.err != nil
	   && err.Error() != test.err.Error()) {
			t.Errorf("betonicSort(%q) = %q,
		    %v; want %q, %v", test.input, got, err, test.want, test.err)
		}
		if got != test.want {
			t.Errorf("betonicSort(%q) = %q,
		    want %q, error: %v", test.input, got, test.want, test.err)
		}
	}
}

// BenchmarkBitonicSort benchmarks the BitonicSort function
func BenchmarkBitonicSort(b *testing.B) {
	// Create a test case with a large input array
	input := "12345678901234567890"

	// Run the BitonicSort function b.N times
	for i := 0; i < b.N; i++ {
		_, _ = BitonicSort(input)
	}
}

// Helper function to generate a random string of digits
func generateRandomString(length int) string {
	str := ""
	for i := 0; i < length; i++ {
		str += strconv.Itoa(i % 10) // Appending digits 0-9 cyclically
	}
	return str
}
```

```
// BenchmarkBitonicSortLargeInput benchmarks the BitonicSort function with a large input
func BenchmarkBitonicSortLargeInput(b *testing.B) {
        // Generate a large input array
        input := generateRandomString(1048576) // Change the length as needed

        // Run the BitonicSort function b.N times
        for i := 0; i < b.N; i++ {
                _, _ = BitonicSort(input)
        }
}
```

1.8.2 Вывод результатов

Тесты

```
Running tool: /usr/local/go/bin/go test -timeout 30s
-run ^TestBetonicSort$ algosITMO/lab3/betonicSortStack

=== RUN   TestBetonicSort
--- PASS: TestBetonicSort (0.00s)
PASS
ok      algosITMO/lab3/betonicSortStack 1.701s
```

Бенчмарки

Ввод 12345678901234567890 и запуск сортировки в длинном цикле.

```
Running tool: /usr/local/go/bin/go test -benchmem
-run=^$ -bench ^BenchmarkBitonicSort$ algosITMO/lab3/betonicSortStack

goos: darwin
goarch: arm64
pkg: algosITMO/lab3/betonicSortStack
=== RUN   BenchmarkBitonicSort
BenchmarkBitonicSort
BenchmarkBitonicSort-8          64744293                18.71 ns/op           16 B/op
1 allocs/op
PASS
ok      algosITMO/lab3/betonicSortStack 3.578s
```

Ввод случайных чисел длиной 2^{20}.

```
Running tool: /usr/local/go/bin/go test -benchmem
-run=^$ -bench ^BenchmarkBitonicSortLargeInput$ algosITMO/lab3/betonicSortStack

goos: darwin
goarch: arm64
pkg: algosITMO/lab3/betonicSortStack
=== RUN   BenchmarkBitonicSortLargeInput
BenchmarkBitonicSortLargeInput
BenchmarkBitonicSortLargeInput-8                1       63587984583 ns/op        1107934582064 B/op
2122531 allocs/op
PASS
ok      algosITMO/lab3/betonicSortStack 64.945s
```

1.8.3 Ручное тестирование

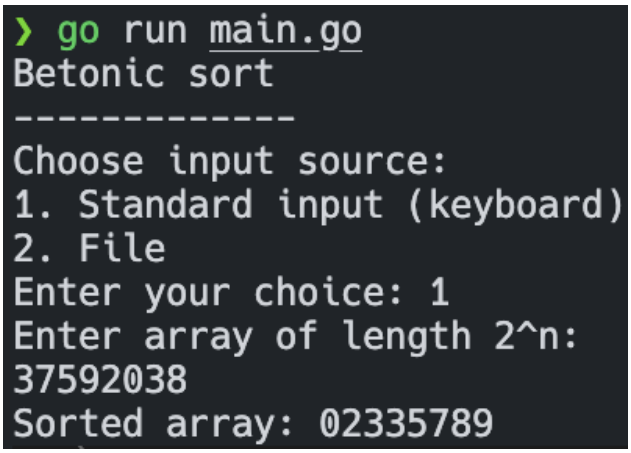


Рисунок 1.10 — Обычная ситуация для ввода с клавиатуры.

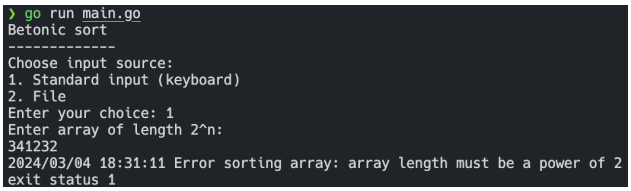


Рисунок 1.11 — Ввод массива не равным 2^n.

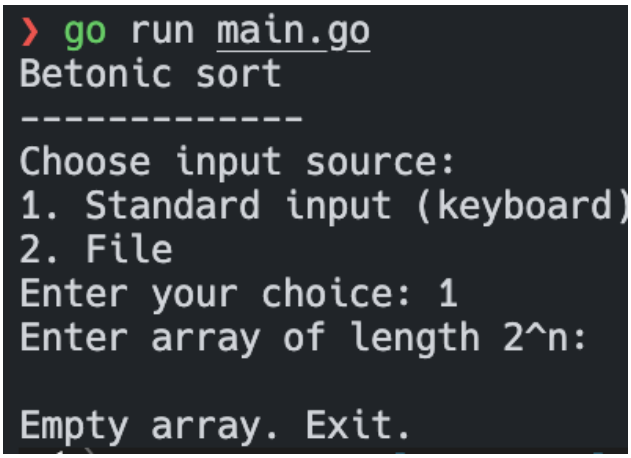


Рисунок 1.12 — Пустой ввод.

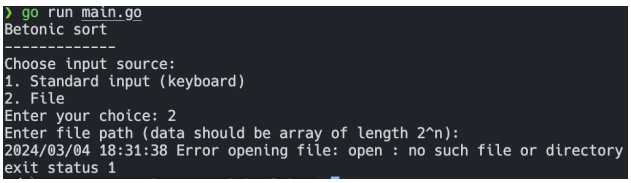


Рисунок 1.13 — Пустой ввод пути файла.

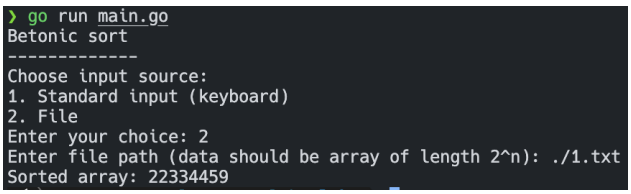


Рисунок 1.14 — Обычная ситуация для ввода с файла.

ЗАКЛЮЧЕНИЕ

Задача: Реализовать программу для битонической сортировки.
Реализованы функции:

1. Ввод массива (по выбору пользователя, как через файл, так и через консоль) и его проверка на корректность.
2. Выполнение битонической сортировки (в функции `BitonicSort`) используя структуру типа stack.

Кроме того, была реализована структура данных для выполнения сортировки через стек.

Среда запуска: Go 1.22.0 (выпущена 2024-02-06)
Редактор: Visual Studio Code
Все тесты были успешно пройдены.