Министерство науки и высшего образования РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Курский государственный университет»

Кафедра программного обеспечения и администрирования информационных систем

Направление подготовки математическое обеспечение и администрирование информационных систем

Форма обучения очная

Отчет
по лабораторной работе №3
«Сравнительный анализ алгоритма поиска»

Выполнил:
студент группы 213 Файтельсон А.А.

Проверил:
ассистент кафедры ПОиАИС Овсянников А.В.

Курск, 2024

Цель работы: Изучение алгоритмов поиска элемента в массиве и закрепление навыков в проведении сравнительного анализа алгоритмов.

Листинг программы

```
package main
import (
	"errors"
	"fmt"
	"math/rand"
	"slices"
	"testing"
	"time"
)
func generateRandomArray(N int) []int {
	// Seed the random number generator
	rand.Seed(time.Now().UnixNano())

	// Create a slice with a length of N
	array := make([]int, N)

	// Populate the array with random values
	for i := 0; i < N; i++ {
		array[i] = rand.Intn(10000000000) // Random numbers between 0 and
99
	}

	return array
}

func NaiveSearch(v int, slice []int) (int, error) {
	for i, value := range slice {
		if value == v {
```

```
                return i, nil
            }
        }
        return 0, errors.New("element not found")
}

func FastNaiveSearch(v int, slice []int) (int, error) {
        // slice = append(slice, v)
        // i := 0
        // for {
        //      if slice[i] == v {
        //              if i == (len(slice)) {
        //                      return 0, errors.New("element not found")
        //              }
        //              return i, nil
        //      }
        //      i++
        // }
        n := len(slice)

        slice = append(slice, v)

        i := 0
        for slice[i] != v {
            i++
        }

        if i == n {
            return 0, errors.New("element not found")
        }
        return i, nil
```

```
}

func FastNaiveSearchSorted(v int, slice []int) (int, error) {
	slice = append(slice, v)
	i := 0
	for {
		if slice[i] >= v {
			if i == (len(slice)) || slice[i] > v {
				return 0, errors.New("element not found")
			}
			return i, nil
		}
		i++
	}
}

func BlockSearch(v int, slice []int) (int, error) {
	if v <= 0 || len(slice) == 0 {
		return 0, errors.New("element not found")
	}
	medium := len(slice) / 2
	if slice[medium] == v {
		return medium, nil
	}
	if slice[medium] > v {
		res, err := BlockSearch(v, slice[0:medium])
		return res, err
	}
	res, err := BlockSearch(v, slice[medium:])
	return res, err
```

```
}

func BinarySearch(v int, slice []int) (int, error) {
	l, r := 0, len(slice)
	for l < r {
		mid := (l + r) / 2
		if slice[mid] == v {
			return slice[mid], nil
		}
		if slice[mid] > v {
			r = mid
		}
		if slice[mid] < v {
			l = mid
		}
	}
	return 0, errors.New("element not found")
}

var SizeOfCases = []int{50, 100, 150, 200, 300, 350, 400, 450}

func BlockSearch1(v int, slice []int) (int, error) {
	// Base case: If the slice is empty, return an error
	if len(slice) == 0 {
		return -1, errors.New("element not found")
	}

	// Find the middle index
	medium := len(slice) / 2

	if slice[medium] == v {
```

```
            return medium, nil
        }

        if slice[medium] > v {
            result, err := BlockSearch(v, slice[:medium])
            if err != nil {
                return -1, err
            }
            return result, nil
        } else {
            result, err := BlockSearch(v, slice[medium+1:])
            if err != nil {
                return -1, err
            }
            return medium + 1 + result, nil
        }
    }

    func main() {
        // Example usage
        slice := []int{1, 3, 5, 7, 9, 11, 13, 15}
        value := 7

        index, err := BlockSearch(value, slice)
        if err != nil {
            fmt.Println(err)
        } else {
            fmt.Printf("Element %d found at index %d\n", value, index)
        }
    }
```

```
func BenchmarkNaiveSearch(b *testing.B) {
	for _, t := range SizeOfCases {
		b.Run(fmt.Sprintf("size of case: %d", t), func(b *testing.B) {
			b.StopTimer()
			value := 355
			arr := generateRandomArray(t)
			b.StartTimer()
			for i := 0; i < b.N; i++ {
				NaiveSearch(value, arr)
			}
		})
	}
}

func BenchmarkFastNaiveSearch(b *testing.B) {
	for _, t := range SizeOfCases {
		b.Run(fmt.Sprintf("size of case: %d", t), func(b *testing.B) {
			b.StopTimer()
			value := 355
			arr := generateRandomArray(t)
			b.StartTimer()
			for i := 0; i < b.N; i++ {
				FastNaiveSearch(value, arr)
			}
		})
	}
}

func BenchmarkFastNaiveSearchSorted(b *testing.B) {
	for _, t := range SizeOfCases {
```

```
            b.Run(fmt.Sprintf("size of case: %d", t), func(b *testing.B) {
                b.StopTimer()
                value := 355
                arr := generateRandomArray(t)
                slices.Sort(arr)
                b.StartTimer()
                for i := 0; i < b.N; i++ {
                    FastNaiveSearchSorted(value, arr)
                }
            })
        }
}

func BenchmarkBlockSearch(b *testing.B) {
        for _, t := range SizeOfCases {
            b.Run(fmt.Sprintf("size of case: %d", t), func(b *testing.B) {
                b.StopTimer()
                value := 393
                arr := generateRandomArray(t)
                slices.Sort(arr)
                b.StartTimer()
                for i := 0; i < b.N; i++ {
                    BlockSearch(value, arr)
                }
            })
        }
}

func BenchmarkBinarySearch(b *testing.B) {
        for _, t := range SizeOfCases {
```

```
            b.Run(fmt.Sprintf("size of case: %d", t), func(b *testing.B) {
                b.StopTimer()
                value := 355
                arr := generateRandomArray(t)
                slices.Sort(arr)
                b.StartTimer()
                for i := 0; i < b.N; i++ {
                    BinarySearch(value, arr)
                }
            })
        }
    }
```

Результаты сравнительного анализа алгоритмов.
(в микросекундах)

Таблица 1 - Максимальное количество операций сравнения

Алгоритмы поиска	Количество элементов в массиве								
	50	100	150	200	250	300	350	400	450
1.а	50	100	150	200	250	300	350	400	450
1.б	50	100	150	200	250	300	350	400	450
2.а	50	100	150	200	250	300	350	400	450
2.б	14	20	25	29	32	35	37	40	42
2.в	6	7	8	8	8	9	9	9	9

Таблица 2 - Среднее количество операций сравнения

Алгоритмы поиска	Количество элементов в массиве								
	50	100	150	200	250	300	350	400	450
1.а	25	50	75	100	125	150	175	200	225
1.б	25	50	75	100	125	150	175	200	225
2.а	25	50	75	100	125	150	175	200	225
2.б	7	10	12	14	16	18	19	20	21
2.в	6	7	8	8	8	9	9	9	9

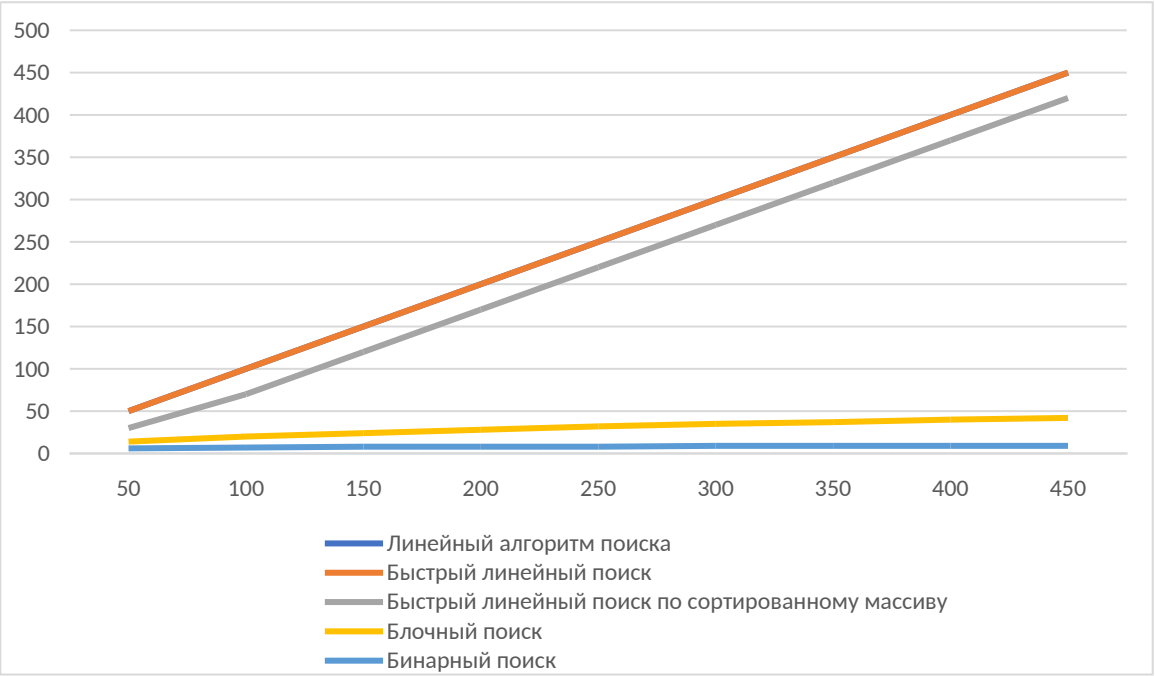

Рисунок 1 – Зависимость количества операций сравнения от количества элементов в массиве.

Вывод по работе: Для решения задачи поиска элемента в отсортированном массиве нужно использовать бинарный поиск, а в остальных случаях обычный поиск работает приемлемо.