**Міністерство освіти і науки України**

**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"**

**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

**Звіт**

з лабораторної роботи № 1 з дисципліни

«Проектування алгоритмів»

„ **Проектування і аналіз алгоритмів зовнішнього сортування**”

Варіант 1

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

ІП-з21 Стихун М.В.

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.М.*

Київ 2023

Зміст

1. Мета лабораторної роботи 3
2. Завдання 4
3. Виконання 6
   1. Алгоритм зовнішнього сортування 6
   2. Програмна реалізація алгоритму 6
      1. Вихідний код 7

Висновок 32

Критерії оцінювання 33

# Мета лабораторної роботи

Мета роботи – вивчити основні алгоритми зовнішнього сортування та способи їх модифікації, оцінити поріг їх ефективності.

# Завдання

Згідно варіанту (таблиця 2.1), розробити та записати алгоритм зовнішнього сортування за допомогою псевдокоду (чи іншого способу за вибором).

Виконати програмну реалізацію алгоритму на будь-якій мові програмування та відсортувати випадковим чином згенерований масив цілих чисел, що зберігається у файлі (розмір файлу має бути не менше 10 Мб, можна значно більше).

Здійснити модифікацію програми і відсортувати випадковим чином згенерований масив цілих чисел, що зберігається у файлі розміром не менше ніж двократний обсяг ОП вашого ПК. Досягти швидкості сортування з розрахунку 1Гб на 3хв. або менше. Достатньо штучно обмежити доступну ОП, для уникнення багатогодинних сортувань (наприклад використовуючи віртуальну машину).

Рекомендується попередньо впорядкувати серії елементів довжиною, що займає не менше 100Мб або використати інші підходи для пришвидшення процесу сортування.

Зробити узагальнений висновок з лабораторної роботи, у якому порівняти базову та модифіковану програми. У висновку деталізувати, які саме модифікації було виконано і який ефект вони дали.

Таблиця 2.1 – Варіанти алгоритмів

|  |  |
| --- | --- |
| **№** | **Алгоритм сортування** |
| 1 | Пряме злиття |
| 2 | Природне (адаптивне) злиття |
| 3 | Збалансоване багатошляхове злиття |
| 4 | Багатофазне сортування |
| 5 | Пряме злиття |
| 6 | Природне (адаптивне) злиття |
| 7 | Збалансоване багатошляхове злиття |
| 8 | Багатофазне сортування |
| 9 | Пряме злиття |
| 10 | Природне (адаптивне) злиття |
| 11 | Збалансоване багатошляхове злиття |
| 12 | Багатофазне сортування |
| 13 | Пряме злиття |
| 14 | Природне (адаптивне) злиття |
| 15 | Збалансоване багатошляхове злиття |
| 16 | Багатофазне сортування |
| 17 | Пряме злиття |
| 18 | Природне (адаптивне) злиття |
| 19 | Збалансоване багатошляхове злиття |
| 20 | Багатофазне сортування |
| 21 | Пряме злиття |
| 22 | Природне (адаптивне) злиття |
| 23 | Збалансоване багатошляхове злиття |
| 24 | Багатофазне сортування |
| 25 | Пряме злиття |
| 26 | Природне (адаптивне) злиття |
| 27 | Збалансоване багатошляхове злиття |
| 28 | Багатофазне сортування |
| 29 | Пряме злиття |
| 30 | Природне (адаптивне) злиття |
| 31 | Збалансоване багатошляхове злиття |
| 32 | Багатофазне сортування |
| 33 | Пряме злиття |
| 34 | Природне (адаптивне) злиття |
| 35 | Збалансоване багатошляхове злиття |

# Виконання

## Алгоритм зовнішнього сортування

* Визначити максимальний обсяг пам'яті, який може бути використаний для сортування у пам'яті (наприклад, 100MB).
* Відкрити вхідний файл для читання та виходовий файл для запису результату.
* Створити тимчасову пам'ять у вигляді буфера та змінної для відстеження використаної пам’яті.
* Прочитати дані з вхідного файлу починаючи з першого запису.
* Додавати записи у буфер до тих пір, поки використана пам'ять не досягне максимального обсягу, або досягнуто кінець вхідного файлу.
* Якщо буфер заповнений, сортувати його у пам’яті.
* Записати вміст буфера у тимчасовий файл.
* Очистити буфер та використану пам’ять.
* Повторювати кроки 4-8 до тих пір, поки не вичерпано вхідні дані.
* Якщо буфер не пустий після завершення обробки вхідних даних:
  + Сортувати його у пам'яті.
  + Записати вміст буфера у тимчасовий файл.
* Об'єднати всі тимчасові файли в один вихідний файл у правильному порядку.
* Закрити всі файли та звільнити ресурси.
* Видалити всі тимчасові файли, створені під час обробки.
* Повернути результат, якщо це потрібно.

## Програмна реалізація алгоритму

### Вихідний код

package main

import (

"flag"

"github.com/Nav1Cr0ss/algorithms/internal/domain/lab\_1"

"github.com/Nav1Cr0ss/algorithms/pkg/sort"

)

func main() {

var inputFileName string

var outputFileName string

var useExternalSort bool

flag.StringVar(&inputFileName, "input", "input.txt", "Input file name")

flag.StringVar(&outputFileName, "output", "output.txt", "Output file name")

flag.BoolVar(&useExternalSort, "external", false, "Use external sort (default is in-memory sort)")

flag.Parse()

var sorter lab\_1.SortProvider

switch useExternalSort {

case true:

sorter = sort.NewExternalSort()

default:

sorter = sort.NewMemorySort()

}

lab\_1.SortArrayOfInt(inputFileName, outputFileName, sorter)

}

package main

import (

"bufio"

"github.com/Nav1Cr0ss/algorithms/internal/domain/lab\_1"

"github.com/Nav1Cr0ss/algorithms/pkg/fs"

"github.com/Nav1Cr0ss/algorithms/pkg/sort"

"os"

"strconv"

"testing"

)

func TestSorting(t \*testing.T) {

tests := []struct {

name string

sortProvider lab\_1.SortProvider

inputFileName string

outputFileName string

totalNum int

}{

{

name: "MemorySort",

sortProvider: sort.NewMemorySort(),

inputFileName: "test\_memory\_input.txt",

outputFileName: "test\_memory\_output.txt",

totalNum: 1000000,

},

{

name: "ExternalSort",

sortProvider: sort.NewExternalSort(),

inputFileName: "test\_external\_input.txt",

outputFileName: "test\_external\_output.txt",

totalNum: 100000000,

},

}

for \_, tt := range tests {

t.Run(tt.name, func(t \*testing.T) {

fileSystem := fs.NewFS()

if err := generateInputFile(fileSystem, tt.inputFileName, tt.totalNum); err != nil {

panic(err)

}

defer func() {

if err := deleteOutputFile(fileSystem, []string{tt.inputFileName, tt.outputFileName}); err != nil {

panic(err)

}

}()

lab\_1.SortArrayOfInt(tt.inputFileName, tt.outputFileName, tt.sortProvider)

outputFile, err := os.Open(tt.outputFileName)

if err != nil {

t.Fatalf("Failed to open %s: %v", tt.outputFileName, err)

}

defer outputFile.Close()

scanner := bufio.NewScanner(outputFile)

numbers := make([]int, 100)

for i := 0; i < 100; i++ {

if !scanner.Scan() {

t.Fatalf("Failed to read 100 numbers from %s", tt.outputFileName)

}

num, err := strconv.Atoi(scanner.Text())

if err != nil {

t.Fatalf("Failed to parse number from %s: %v", tt.outputFileName, err)

}

numbers[i] = num

}

if !isSorted(numbers) {

t.Fatalf("The numbers in %s are not sorted", tt.outputFileName)

}

})

}

}

func isSorted(numbers []int) bool {

for i := 1; i < len(numbers); i++ {

if numbers[i-1] > numbers[i] {

return false

}

}

return true

}

func generateInputFile(fs \*fs.FS, fileName string, totalNum int) error {

err := fs.CreateFileWithArrOfInt(fileName, totalNum)

if err != nil {

return err

}

return nil

}

func deleteOutputFile(fs \*fs.FS, fileNames []string) error {

err := fs.DeleteFiles(fileNames)

if err != nil {

return err

}

return nil

}

package lab\_1

import "bufio"

type Item struct {

value int

index int

scanner \*bufio.Scanner

}

type PriorityQueue []\*Item

func (pq PriorityQueue) Len() int { return len(pq) }

func (pq PriorityQueue) Less(i, j int) bool {

return pq[i].value < pq[j].value

}

func (pq PriorityQueue) Swap(i, j int) {

pq[i], pq[j] = pq[j], pq[i]

pq[i].index = i

pq[j].index = j

}

func (pq \*PriorityQueue) Push(x interface{}) {

item := x.(\*Item)

item.index = len(\*pq)

\*pq = append(\*pq, item)

}

func (pq \*PriorityQueue) Pop() interface{} {

old := \*pq

n := len(old)

item := old[n-1]

item.index = -1

\*pq = old[0 : n-1]

return item

}

package lab\_1

import (

"fmt"

"github.com/Nav1Cr0ss/algorithms/pkg/utilz"

"time"

)

type SortProvider interface {

Sort(inputFile, outputFile string) error

}

func SortArrayOfInt(inputFile, outputFile string, sp SortProvider) {

start := time.Now()

err := sp.Sort(inputFile, outputFile)

if err != nil {

fmt.Println("Error on sorting file")

}

utilz.MeasureTime(start, "Sorting array")

}

package fs

import (

"bufio"

"fmt"

"log"

"math/rand"

"os"

"strconv"

"sync"

)

type FS struct{}

func NewFS() \*FS {

return &FS{}

}

func (f \*FS) DeleteFile(fileName string) error {

err := os.Remove(fileName)

if err != nil {

return err

}

return nil

}

func (f \*FS) ParseIntArrayFromFile(filename string) ([]int, error) {

file, err := os.Open(filename)

if err != nil {

return nil, err

}

defer file.Close()

scanner := bufio.NewScanner(file)

scanner.Buffer(make([]byte, 64\*1024), 1024\*1024)

var nums []int

for scanner.Scan() {

line := scanner.Text()

num, err := strconv.Atoi(line)

if err != nil {

fmt.Println("Error parsing number:", err)

continue

}

nums = append(nums, num)

}

if err := scanner.Err(); err != nil {

fmt.Println("Error scanning file:", err)

return nums, err

}

return nums, nil

}

func (f \*FS) WriteIntArrayToFile(filename string, arr []int) error {

outputFile, err := os.Create(filename)

if err != nil {

fmt.Println("Error creating file:", err)

return err

}

defer outputFile.Close()

writer := bufio.NewWriter(outputFile)

defer writer.Flush()

for \_, num := range arr {

\_, err := writer.WriteString(strconv.Itoa(num) + "\n")

if err != nil {

fmt.Println("Error writing to file:", err)

return err

}

}

return nil

}

func (f \*FS) WriteChunk(filename string, data []int) error {

file, err := os.Create(filename)

if err != nil {

return err

}

defer file.Close()

writer := bufio.NewWriter(file)

defer writer.Flush()

for \_, num := range data {

\_, err := writer.WriteString(strconv.Itoa(num) + "\n")

if err != nil {

return err

}

}

return nil

}

func (f \*FS) MergeChunks(chunkFiles []string, outputFile string) error {

output, err := os.Create(outputFile)

if err != nil {

return err

}

defer output.Close()

chunkData := make(chan string)

var wg sync.WaitGroup

for \_, chunkFile := range chunkFiles {

wg.Add(1)

go func(file string) {

defer wg.Done()

chunkReader, err := os.Open(file)

if err != nil {

log.Println(err)

return

}

defer chunkReader.Close()

scanner := bufio.NewScanner(chunkReader)

for scanner.Scan() {

chunkData <- scanner.Text()

}

if scanner.Err() != nil {

log.Println(scanner.Err())

}

}(chunkFile)

}

go func() {

writer := bufio.NewWriter(output)

defer writer.Flush()

for data := range chunkData {

fmt.Fprintln(writer, data)

}

}()

wg.Wait()

close(chunkData)

return nil

}

func (f \*FS) DeleteFiles(fileNames []string) error {

for \_, name := range fileNames {

err := f.DeleteFile(name)

if err != nil {

return err

}

}

return nil

}

func (f \*FS) CreateFileWithArrOfInt(filename string, totalNumbers int) error {

outputFile, err := os.Create(filename)

if err != nil {

fmt.Println("Error creating file:", err)

return err

}

defer func(outputFile \*os.File) {

err = outputFile.Close()

if err != nil {

panic("panic on closing file")

}

}(outputFile)

writer := bufio.NewWriter(outputFile)

defer func(writer \*bufio.Writer) {

err = writer.Flush()

if err != nil {

panic("panic on flushing file")

}

}(writer)

for i := 0; i < totalNumbers; i++ {

randomNum := rand.Intn(1000000)

\_, err := writer.WriteString(strconv.Itoa(randomNum) + "\n")

if err != nil {

fmt.Println("Error writing to file:", err)

return err

}

}

return nil

}

package main

import (

"flag"

"github.com/Nav1Cr0ss/algorithms/pkg/fs"

//fs2 "github.com/Nav1Cr0ss/algorithms/pkg/fs"

"github.com/Nav1Cr0ss/algorithms/pkg/utilz"

"time"

)

func main() {

var fileName string

flag.StringVar(&fileName, "file\_name", "input.txt", "file name")

files := fs.NewFS()

start := time.Now()

err := files.DeleteFile(fileName)

if err != nil {

return

}

utilz.MeasureTime(start, "Deleting")

}

package main

import (

"flag"

"github.com/Nav1Cr0ss/algorithms/pkg/fs"

//fs2 "github.com/Nav1Cr0ss/algorithms/pkg/fs"

"github.com/Nav1Cr0ss/algorithms/pkg/utilz"

"time"

)

func main() {

var fileName string

var numsToGenerate int

flag.StringVar(&fileName, "file\_name", "input.txt", "file name")

flag.IntVar(&numsToGenerate, "output", 100000, "Nums to generate")

files := fs.NewFS()

start := time.Now()

err := files.CreateFileWithArrOfInt(fileName, numsToGenerate)

if err != nil {

return

}

utilz.MeasureTime(start, "Generating array")

}

package sort

import (

"bufio"

"fmt"

"github.com/Nav1Cr0ss/algorithms/pkg/fs"

"os"

"sort"

"strconv"

)

const *maxMemory* = int64(100 \* 1024 \* 1024)

type ExternalSort struct {

fs \*fs.FS

}

func NewExternalSort() \*ExternalSort {

return &ExternalSort{

fs: fs.NewFS(),

}

}

func (s \*ExternalSort) Sort(inputFile, outputFile string) error {

chunkFiles, err := s.createSortedChunks(inputFile, *maxMemory*)

if err != nil {

return err

}

defer func(files []string) {

err = s.fs.DeleteFiles(files)

if err != nil {

}

}(chunkFiles)

err = s.fs.MergeChunks(chunkFiles, outputFile)

if err != nil {

return err

}

return nil

}

func (s \*ExternalSort) createSortedChunks(inputFile string, maxMemory int64) ([]string, error) {

file, err := os.Open(inputFile)

if err != nil {

return nil, err

}

defer file.Close()

chunkFiles := []string{}

scanner := bufio.NewScanner(file)

buffer := []int{}

memUsed := int64(0)

for scanner.Scan() {

line := scanner.Text()

num, err := strconv.Atoi(line)

if err != nil {

return nil, err

}

buffer = append(buffer, num)

memUsed += int64(len(line) + 1)

if memUsed >= maxMemory {

sort.Ints(buffer)

chunkFile := fmt.Sprintf("chunk%d.txt", len(chunkFiles))

err := s.fs.WriteChunk(chunkFile, buffer)

if err != nil {

return nil, err

}

chunkFiles = append(chunkFiles, chunkFile)

buffer = []int{}

memUsed = 0

}

}

if len(buffer) > 0 {

sort.Ints(buffer)

chunkFile := fmt.Sprintf("chunk%d.txt", len(chunkFiles))

err := s.fs.WriteChunk(chunkFile, buffer)

if err != nil {

return nil, err

}

chunkFiles = append(chunkFiles, chunkFile)

}

return chunkFiles, nil

}

package sort

import "github.com/Nav1Cr0ss/algorithms/pkg/fs"

type MemorySort struct {

fs \*fs.FS

}

func NewMemorySort() \*MemorySort {

return &MemorySort{

fs: fs.NewFS(),

}

}

func (s \*MemorySort) merge(left, right []int) []int {

result := make([]int, 0, len(left)+len(right))

l, r := 0, 0

for l < len(left) && r < len(right) {

if left[l] < right[r] {

result = append(result, left[l])

l++

} else {

result = append(result, right[r])

r++

}

}

result = append(result, left[l:]...)

result = append(result, right[r:]...)

return result

}

func (s \*MemorySort) MergeSort(arr []int) []int {

if len(arr) <= 1 {

return arr

}

mid := len(arr) / 2

left := s.MergeSort(arr[:mid])

right := s.MergeSort(arr[mid:])

return s.merge(left, right)

}

func (s \*MemorySort) Sort(inputFile, outputFile string) error {

nums, err := s.fs.ParseIntArrayFromFile(inputFile)

if err != nil {

return err

}

sortedNums := s.MergeSort(nums)

err = s.fs.WriteIntArrayToFile(outputFile, sortedNums)

if err != nil {

return err

}

return nil

}

package generate

//go:generate go run pkg/scripts/del\_file/main.go

//go:generate go run pkg/scripts/gen\_rand\_num\_file/main.go

Висновок

У ході виконання лабораторної роботи було досліджено та реалізовано два важливих алгоритми сортування: сортування в пам'яті та зовнішнього сортування.

Сортування в пам'яті є дієвим методом для сортування невеликих обсягів даних, коли весь набір даних поміщається у пам'ять комп'ютера.

Ми вивчили реалізацію алгоритму сортування злиттям **(Merge Sort)** та виконали практичну реалізацію цього алгоритму у мові програмування Go. Сортування в пам'яті є ефективним та швидким, але має обмеження на обсяг даних, які можна обробити.

Зовнішнє сортування **(External Sort)** є методом сортування для дуже великих обсягів даних, які не вміщуються у пам'ять. Ми дослідили алгоритм зовнішнього сортування, який вимагає поділу даних на менші частини (чанки), їх сортування та об'єднання в правильному порядку. За для досягнення максимальної швидкості, запис в банки проводив п паралельному форматі. Цей метод дозволяє сортувати навіть дуже великі набори даних, але вимагає більше обчислювальних та дискових ресурсів.

Критерії оцінювання

У випадку здачі лабораторної роботи до 08.10.2022 включно максимальний бал дорівнює – 5. Після 08.10.2022 максимальний бал дорівнює – 4,5.

Критерії оцінювання у відсотках від максимального балу:

* псевдокод алгоритму – 15%;
* програмна реалізація алгоритму – 20%;
* програмна реалізація модифікацій – 20%;
* робота з git – 40%;
* висновок – 5%.