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 №11**

«РЕШЕНИЕ КОМБИНАТОРНЫХ ЗАДАЧ МЕТОДОМ

ПЕРЕБОРА»

Выполнил:

студент группы 213 Файтельсон А.А.

Проверил:

ассистент кафедры ПОиАИС Овсянников А.В.

Курск, 2025

**Цель работы:** Знакомство с общей постановкой комбинаторных задач, освоение техники решения таких задач методом перебора и приемов анализа решения.

**Словесная постановка задачи:**

Задача о куче камней. Имеется N камней, известны их веса Pi (i=1...N), задано количество куч M. Требуется разложить камни на M куч так, чтобы минимизировать вес самой тяжелой кучи. (Вариант формулировки: даны N программ с длительностями Pi и M процессоров. Требуется распределить программы так, чтобы раньше закончить выполнение).

**Вариант задания:** 6

**Таблица результатов программы:**

|  |  |  |  |
| --- | --- | --- | --- |
| Номер теста | Кол-во камней | Кол-во куч | Время исполнения |
| 1 | 5 | 3 | 11 мс |
| 2 | 5 | 2 | 6 мс |
| 3 | 5 | 5 | 4 мс |
| 4 | 5 | 2 | 5 мс |

**Листинг программы**

package main

import (

"fmt"

"math/rand"

"sort"

"time"

)

// canDistribute проверяет, можно ли уложить все камни weights

// в M куч с "потолком" mid. Используем backtracking.

func canDistribute(weights []int, M int, mid int) bool {

// loads[j] будет хранить суммарный вес кучи j

loads := make([]int, M)

// рекурсивная функция, которая пытается

// разложить камни, начиная с индекса idx

var backtrack func(idx int) bool

backtrack = func(idx int) bool {

// Если все камни разложили — успех

if idx == len(weights) {

return true

}

w := weights[idx]

// Перебираем кучи, пытаясь положить туда камень weights[idx]

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

// если можно положить в кучу i

if loads[i]+w <= mid {

loads[i] += w

// Переходим к следующему камню

if backtrack(idx + 1) {

return true

}

// Откатываемся

loads[i] -= w

// Важный трюк оптимизации: если кучи i и i-1 имеют одинаковую

// загрузку, то нет смысла класть камень в следующие такие же кучи

// (иначе получим те же комбинации).

// То есть, если эта куча была 0 или равна предыдущей, не тратим

// время на перебор следующих пустых/аналогичных куч.

if loads[i] == 0 {

break

}

}

}

// Если не смогли положить камень ни в одну из куч, возвращаем false

return false

}

// Запускаем backtracking с 0-го камня

return backtrack(0)

}

func main() {

// Примеры тестов

tests := []struct {

weights []int

M int

}{

{[]int{1, 2, 3, 4, 5}, 3}, // Простой случай

{[]int{7, 2, 5, 10, 8}, 2}, // Сложный случай

{[]int{1, 1, 1, 1, 1}, 5}, // Все камни равны

{[]int{30, 20, 10, 50, 40}, 2}, // Разные веса

{generateLargeDataset(1000, 1000), 10}, // Большие данные: 1000 камней, 10 куч

{generateLargeDataset(10000, 10000), 100}, // Очень большие данные: 10000 камней, 100 куч

{generateLargeDataset(100000, 100000), 1000}, // Максимальные данные: 100000 камней, 1000 куч

}

for i, test := range tests {

fmt.Printf("Test %d: Weights length = %d, M = %d\n", i+1, len(test.weights), test.M)

sort.Ints(test.weights) // Сортируем веса для удобства

start := time.Now() // Начало замера времени

minimizeMaxWeight(test.weights, test.M)

elapsed := time.Since(start) // Конец замера времени

fmt.Printf("Execution time: %v\n\n", elapsed)

}

}

func minimizeMaxWeight(weights []int, M int) {

// Пример входных данных.

// N = 5 камней, M = 2 кучи

// weights = [2, 2, 4, 7, 10]

// Ожидаем в результате минимальный "максимальный вес кучи" = 12 (пример: [10,2] и [7,4,2])

// M := 2

// weights := []int{2, 2, 4, 7, 10}

// Сортируем по убыванию — это помогает бэктрекингу быстрее находить неудачные раскладки

sort.Slice(weights, func(i, j int) bool {

return weights[i] > weights[j]

})

// Левая граница: максимальный камень

left := weights[0]

// Правая граница: сумма всех камней

sum := 0

for \_, w := range weights {

sum += w

}

right := sum

// Бинарный поиск по ответу

for left < right {

mid := (left + right) / 2

if canDistribute(weights, M, mid) {

// Если уложились в mid, то нужно попробовать найти что-то меньше

right = mid

} else {

// Иначе надо брать больше

left = mid + 1

}

}

// После выхода из цикла left == right — это наш ответ

fmt.Println("Минимально возможная максимальная куча:", left)

}

// generateLargeDataset генерирует случайный массив весов длины n,

// где каждый вес находится в диапазоне [1, maxWeight].

func generateLargeDataset(n int, maxWeight int) []int {

rand.Seed(time.Now().UnixNano()) // Инициализация генератора случайных чисел

weights := make([]int, n)

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

weights[i] = rand.Intn(maxWeight) + 1 // Генерация веса от 1 до maxWeight

}

return weights

}