

**课 程 实 验 报 告**

**课程名称： 算法设计**

**实验名称： 01背包问题**

**院 系 ：计算机科学与技术**

**专业班级 ： 计卓1701**

**学 号 ： U201714487**

**姓 名 ： 蔡靖涛**

**指导教师 ： 何琨**

**2019 年 11 月 10 日**

1. **实验内容**

本次实验的内容是基于动态规划算法思想的01背包问题求解，我所实现的具体内容是由Kotlin语言编写的采用动态规划01背包问题求解算法。

1. **算法描述**

我的算法可以大致分为以下几步：

* + - 1. 对于n件物品（value, weight），背包总容量为W的问题，声明一个大小为n\*W的二维数组，K[i][w]表示在面对第i件物品，且背包剩余容量为w时所能获得的最大价值。
      2. 现在对于从0开始的i，K[0][w] = 0，K[i][0] = 0。然后从i=1开始循环直到i=n，在w<W的情况下：
      3. w<w[i]的情况，这时候背包容量不足以放下第i件物品，只能选择不拿，即w不变，K[i][w] = K[i-1][w]。
      4. w>=w[i] 的情况，这时背包容量可以放下第 i 件物品，我们就要考虑拿这件物品是否能获取更大的价值，即：

K[i][w] = max(

K[i-1][w-weight[i-1]] + value[i - 1]，

K[i - 1][w])

1. **程序设计**
   1. **数据结构定义（数据成员）**
      1. val K = Array(n + 1) { IntArray(W + 1) {0} }

初始化一个大小为(n+1)\*(W+1)的二维Int数组并初始化值为0。其中i=0行和w=0列都为0，表示无物品在背包中。

* 1. **核心功能实现（函数成员）**
     + 1. fun knapSack(

W: Int,

weight: Array<String>,

value: Array<String>,

n: Int): Int

计算函数knapSack函数接收：

一个Int型值参W，表示背包重量容量；

一个字符串数组weight表示所有物品的重量；

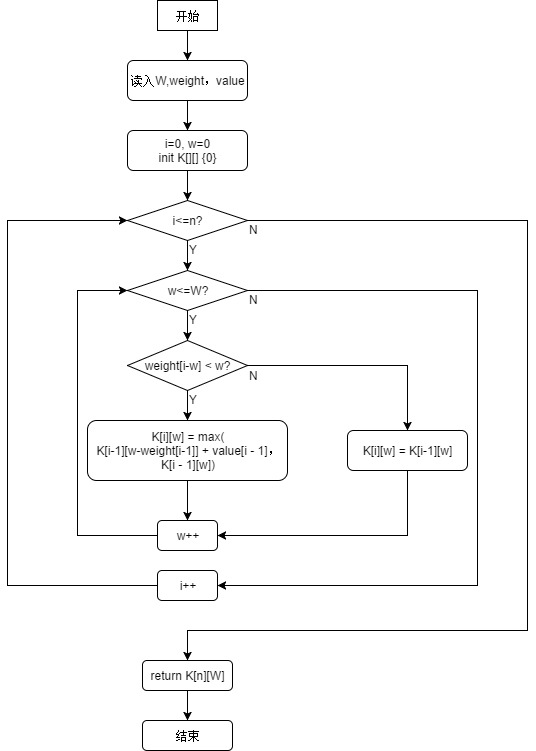
一个字符串数组value表示所有物品的价值；

一个Int型值参n表示物品总个数

函数产生一个Int型var变量w表示当前背包内重量，一个Int型二维数组K，K[i][w]表示在面对第i件物品，且背包内剩余容量为w时所能获得的最大价值。

函数逻辑就是算法的递推逻辑，当对于i=0，w=0开始，对每一个i和每一个w判断：

* + - * 1. 若i==0 || w==0, K[i][w] = 0
        2. 若weight[i-1]<=w，背包剩余容量内可以放下这件物品，K[i][w] = max(K[i-1][w-weight[i-1]] + value[i - 1]，K[i - 1][w])，加入的话则w-weight[i]，K+value[i]，不加入的话则保持为K[i-1][w]
        3. 否则放不下当前物品，K保持为K[i-1][w]。
        4. 返回K[n][W]
  1. **输入输出与测试模块**
     + 1. 程序读入一个含有三行数据的文本文件作为输入，第一行包含一个整数，为背包总容量W；第二行包含n个整数，依次为n个物品的重量，以空格符号隔开；第三行也包含n个整数，对应地为n个物品的价值（最后另外两行标明正确输出，但是并不读入）。
       2. 输出首先显示读入的W，weight数组以及value数组，然后输出背包可以携带的最大价值。



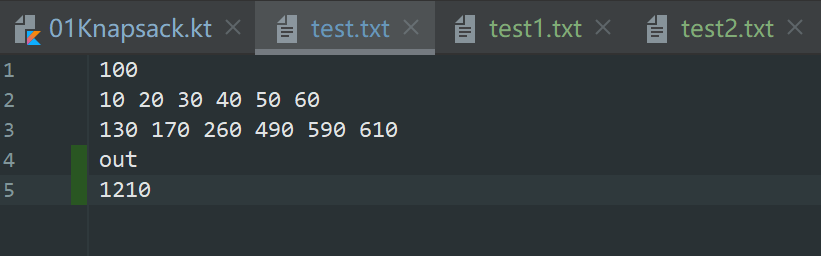
**图3-1 程序流程图**

1. **结果测试**
   1. **测试环境**

该程序在运行在Windows 10系统上的IntelliJ IDEA 19.2 Community版本编译通过并成功运行（Java12，Kotlin1.3）。

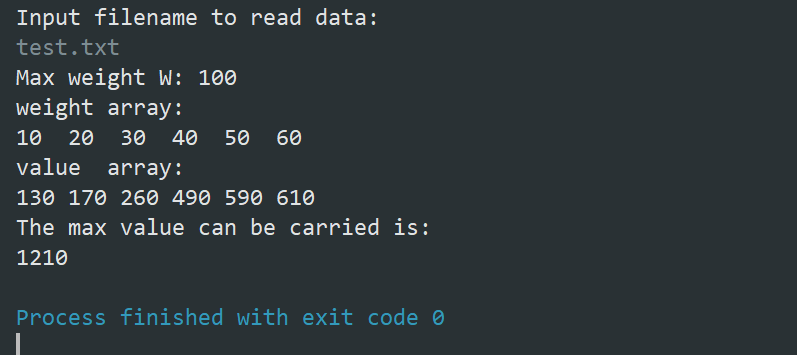
* 1. **测试样例与结果**
     1. **测试样例1**
        1. 输入：

具有五个物体，背包容量为100的0-1knapsack问题，问题文件如下：



**图4-1-1 sample1测试文件**

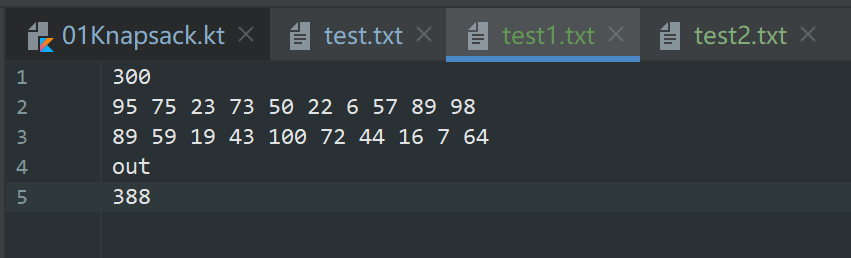
* + - 1. 测试结果：



**图4-1-2 sample1测试结果**

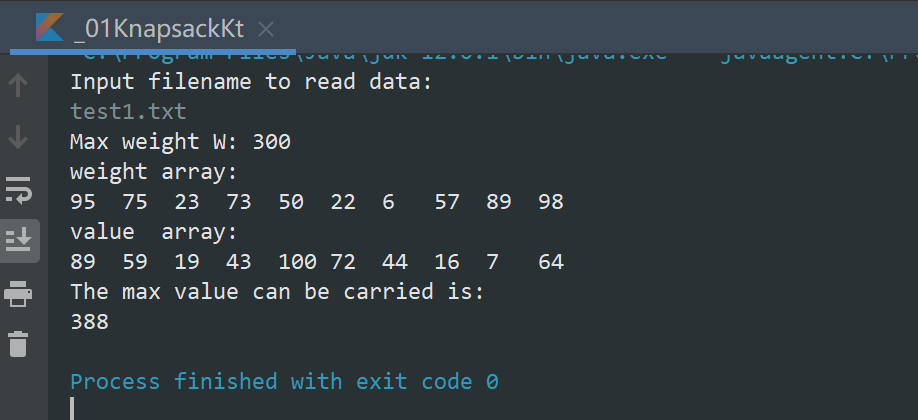
* + 1. **测试样例2**
       1. 输入：

具有十个物体，背包容量为300的0-1knapsack问题，问题文件如下：



**图4-2-1 sample2测试文件**

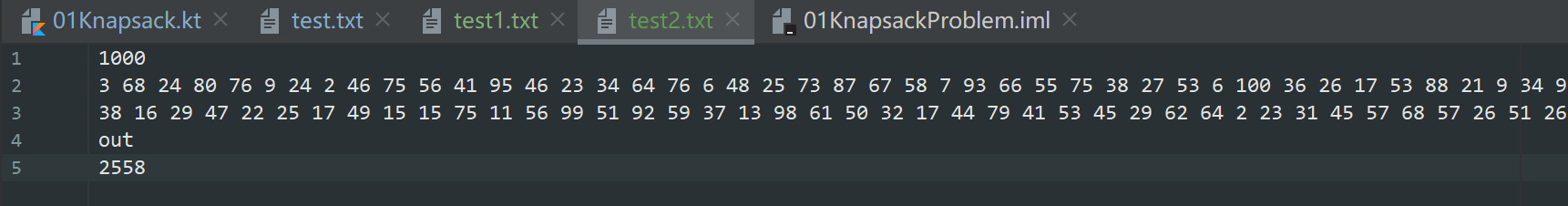
* + - 1. 测试结果：



**图4-2-2 sample2测试结果**

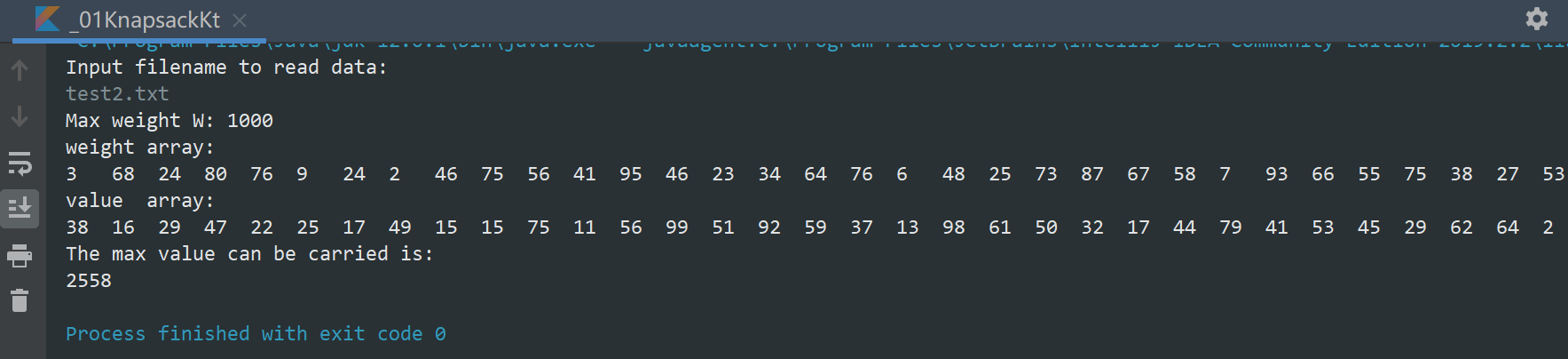
* + 1. **测试样例3**
       1. 输入：

具有一百个物体，背包容量为1000的0-1knapsack问题，问题文件如下：



**图4-3-1 sample3测试文件（不完整）**

* + - 1. 测试结果：



**图4-3-2 sample3测试结果**

1. **时间复杂度分析**

该算法很明显存在两个嵌套的循环，用以构建动态规划的价值表K[n][W]，所以时间复杂度为O(n\*W)，其中n为物品件数，W为背包总承重。

1. **源代码**

import java.io.File

// A utility function that returns maximum of two integers

fun max(a: Int, b: Int): Int {

return if (a > b) a else b

}

// Returns the maximum value that can be put in a knapsack of capacity W

fun knapSack(W: Int, weight: Array<String>, value: Array<String>, n: Int): Int {

var i: Int = 0

var w: Int

val K = Array(n + 1) { IntArray(W + 1) { 0 } }

// Build table K[][] in bottom up manner

while (i <= n) {

w = 0

while (w <= W) {

if (i == 0 || w == 0)

K[i][w] = 0

else if (weight[i - 1].toInt() <= w)

K[i][w] = max(value[i - 1].toInt() + K[i - 1][w - weight[i - 1].toInt()], K[i - 1][w])

else

K[i][w] = K[i - 1][w]

w++

}

i++

}

return K[n][W]

}

// Driver program to test above function

fun main(args: Array<String>) {

// val value = intArrayOf(60, 100, 120)

// val weight = intArrayOf(10, 20, 30)

// val W = 50

println("Input filename to read data:")

val filename = readLine()

val line = File(filename).readLines()

val W = line[0].toInt()

val weight = line[1].split(" ").toList().toTypedArray()

val value = line[2].split(" ").toList().toTypedArray()

var i = 0

val n = value.size

println("Max weight W: $W")

println("weight array:")

while (i < n) {

print("${weight[i]}\t")

i++

}

i = 0

println()

println("value array:")

while (i < n) {

print("${value[i]}\t")

i++

}

println()

println("The max value can be carried is:")

println(knapSack(W, weight, value, n))

}