Московский авиационный институт (национальный исследовательский университет)

Институт информационных технологий и прикладной математики

«Кафедра вычислительной математики и программирования»

Лабораторная работа по предмету "Дискретный анализ" №7

Студент: Шипилов К. Ю.

Преподаватель: Макаров Н. К.

Группа: М8О-303Б-22

Дата: 02.11.2024

Оценка:

Подпись:

Москва 2024 г.

Оглавление

Постановка задачи

У вас есть рюкзак, вместимостью m, а так же n предметов, у каждого из которых есть вес w_i и стоимость c_i. Необходимо выбрать такое подмножество I из них, чтобы:

- $\sum_{i \in I} w_i \leqslant m$
- $(\sum_{i \in I} c_i) * |I|$ является максимальной из всех возможных.

|I| – мощность множества I.

Формат ввода

В первой строке заданы $1 \leqslant n \leqslant 100$ и $1 \leqslant m \leqslant 5000$. В последующих n строках через пробел заданы параметры предеметов: w_i и c_i.

Формат вывода

В первой строке необходимо вывести одно число – максимальное значение $(\sum_{i \in I} c_i) * |I|$, а на второй – индексы предметов, входящих в ответ.

Алгоритм решения

Для решения задачи использовался классический алгоритм решения задачи о рюкзаке с добавлением в массив dp третьего измерения, которое соответствует количеству предметов в рюкзаке.

Из-за ограничений по памяти на i-ом шаге алгоритма хранится только значение dp[i — 1].

Многомерный массив dp_i с размерностью (m+1)×(n+1) инициализируется с помощью конструктора по умолчанию для класса Knapsack, который соответствует пустому рюкзаку.

При заполнении массива на i-ом необходимо пройтись по каждому элементу, используя внешний цикл по переменной w, которая соответствует текущему максимальному весу рюкзака, и внутренний цикл по переменной k, которая соответствует количеству предметов в рюкзаке. Если вес текущего предмета больше максимальной вместимости рюкзака, необходимо в текущую ячейку сохранить значение, которое хранилось в массиве из предыдущего шага по индексам, соответствующим w и k. Если вес текущего предмета меньше или равен максимальной вместительности рюкзака, то надо найти рюкзак, который хранился в массиве из предыдущего шага по индексам, соответствующим w и k, и рюкзак, с весом равным разности между w и весом текущего предмета, в котором количество предметов было k — 1 и добавить к нему новый предмет. Среди двух рюкзаков нужно выбрать максимум и сохранить в массиве по индексам w, k.

После прохождения n шагов, необходимо из последнего двумерного массива взять все рюкзаки с весом m и найти среди них максимум.

Для получения списка всех предметов, входящих в ответ в классе Knapsack хранится список всех предметов. При перегрузке оператора +, предметы в рюкзаке справа от оператора добавляются в конец копии списка из рюкзака слева от оператора. Так как при использовании этого оператора в программе к рюкзаку с предметами имеющими номера < i добавляется рюкзак с одним предметом под номером i, то i всегда будет в конце списка для рюкзаков с любым количеством элементов. Поэтому этот список можно не сортировать.

Сложность алгоритма по времени составляет $O(n^2m)$, так как приходится n раз заполнять матрицу (m+1)×(n+1).

Исходный код

main.cpp

```
#include <iostream>
#include <vector>

class Knapsack {
 private:
  u_int64_t cost;
  std::vector<u_int64_t> items;

 public:
  Knapsack() : cost(0), items() {}

  Knapsack(u_int64_t cost, u_int64_t item) : cost(cost),
items() {
    items.push_back(item);
  }

  Knapsack(u_int64_t cost, const std::vector<u_int64_t>& items)
      : cost(cost), items(items) {}

  u_int64_t getValue() const noexcept { return cost *
items.size(); }

  const std::vector<u_int64_t>& getItems() const noexcept
{ return items; }

  Knapsack operator+(const Knapsack& other) const {
```

```
    u_int64_t cost = this->cost + other.cost;
    std::vector<u_int64_t> items{this->items};
    for (auto item : other.items) {
      items.push_back(item);
    }
    return Knapsack(cost, items);
  }

  bool operator>(const Knapsack& other) const noexcept {
    return this->getValue() > other.getValue();
  }
};

std::vector<std::vector<Knapsack>> calculateItem(
    u_int64_t n, u_int64_t m, u_int64_t item, u_int64_t weight,
u_int64_t cost,
    const std::vector<std::vector<Knapsack>>& previousRow) {
  std::vector<std::vector<Knapsack>> row(
      m + 1, std::vector<Knapsack>(n + 1, Knapsack()));
  for (u_int64_t w = 1; w <= m; ++w) {
    for (u_int64_t k = 1; k <= n; ++k) {
      if (weight <= w) {
        Knapsack withoutCurrentItem = previousRow[w][k];
        Knapsack withCurrentItem =
            previousRow[w - weight][k - 1] + Knapsack(cost,
item);
        if (withCurrentItem.getItems().size() == k &&
            withCurrentItem > withoutCurrentItem) {
```

```
          row[w][k] = withCurrentItem;
        } else {
          row[w][k] = withoutCurrentItem;
        }
      } else {
        row[w][k] = previousRow[w][k];
      }
    }
  }
  return row;
}

Knapsack calculateKnapsack(u_int64_t n, u_int64_t m) {
  std::vector<std::vector<Knapsack>> dp_i(
      m + 1, std::vector<Knapsack>(n + 1, Knapsack()));

  for (u_int64_t i = 1; i <= n; ++i) {
    u_int64_t weight, cost;
    std::cin >> weight >> cost;
    dp_i = calculateItem(n, m, i, weight, cost, dp_i);
  }
  Knapsack bestKnapsack = dp_i[m][0];
  for (Knapsack knapsack : dp_i[m]) {
    if (knapsack > bestKnapsack) {
      bestKnapsack = knapsack;
    }
  }
  return bestKnapsack;
```

```
}

int main() {
  u_int64_t n, m;
  std::cin >> n >> m;
  Knapsack solution = calculateKnapsack(n, m);
  std::cout << solution.getValue() << "\n";
  for (u_int64_t item : solution.getItems()) {
    std::cout << item << " ";
  }
  std::cout << "\n";
}
```

Тесты

Test 1

Input	Output
3 6 2 1 5 4 4 2	6 1 3

Test 2

Input	Output
4 5 4 11 2 3	22 1 4

2 3 1 1	

Test 3

Input	Output
6 35 5 2 8 1 9 7 8 15 5 11 9 10	195 1 2 4 5 6

Вывод

Выполняя лабораторную работу я изучил методы динамического программирования и решил задачу о рюкзаке с дополнительным ограничением. Методы динамического программирования позволяют эффективно решать задачи, требующие выбора оптимального подмножества из набора элементов с ограничениями. Решение задачи о рюкзаке можно применять для решения других задач подобного типа, что делает данный метод универсальным.