# Министерство образования и науки Российской Федерации

# Федеральное государственное бюджетное образовательное учреждение высшего образования «Новосибирский государственный технический университет»



## Кафедра параллельных вычислительных технологий

## Домашнее задание № 5 по дисциплине «Методы проектирования и анализа алгоритмов»

**NP-полные задачи**

Вариант 12

|  |  |  |
| --- | --- | --- |
|  | Факультет: | ПМИ |
| Группа: | ПМ-53 |
| Студент: | Тябин Е.А. |
| Преподаватель: | Щукин Г.А. |

Новосибирск

2017

# Условие задачи

# Для указанной NP-полной задачи задачи изучить и запрограммировать два алгоритма решения: находящий точное решение (полный перебор, с экспоненциальным временем работы) и находящий приближенное решение (жадный алгоритм, алгоритм локального поиска, рандомизированный алгоритм и т.п., с не экспоненциальным временем работы); язык программирования по выбору. Протестировать точный алгоритм на юнит-тестах. Сравнить время работы алгоритмов и качество полученных решений на наборах входных данных различного размера (входные данные сгенерировать самостоятельно). Если для данных большого размера не хватает времени дождаться получения точного решения, привести приблизительную оценку возможного времени работы и указать, что решение не получено.

12. Неограниченная задача о рюкзаке (unbounded knapsack - каждый предмет может быть выбран неограниченное число раз)

# Алгоритм решения

В этой задаче также имеется несколько предметов, для каждого предмета заданы две характеристики: вес Wi>0 и стоимость («полезность») предмета Pi>0. Необходимо выбрать множество предметов суммарной максимальной стоимости, при этом суммарная масса выбранных предметов должна быть ограничена значением K.

Будем для каждой возможной массы k хранить информацию о способе набора этой массы, и наилучшее решение для данной массы, то есть наибольшую стоимость предметов, которые можно набрать в рюкзак данной массы.

Формально определим так: F(i,k)- максимальная стоимость предметов, которые можно уложить в рюкзак массы k, если можно использовать только первые i предметов.

Выведем рекуррентное соотношение для F(i,k) уменьшив значение i. Есть две возможности собрать рюкзак, используя первые i предметов - взять предмет с номером i или не брать.

Если не брать предмет с номером i, то в этом случае F(i,k)=F(i−1,k), так как рюкзак массы k будет собран только с использованием первых i−1 предмета. В этом случае F(i,k)=F(i−1,k).

Если же предмет номер i войдет в рюкзак (это можно сделать только при k≥Wi), то останется свободная вместимость рюкзака k−Wi, которую можно будет заполнить первыми i−1 предметом, максимальная стоимость рюкзака в этом случае будет F (i−1, k−Wi). Но поскольку предмет номер i был включен в рюкзак, то стоимость рюкзака увеличится на Pi. То есть в этом случае F(i,k)=F(i−1,k−Wi)+Pi.

Из двух возможных вариантов нужно выбрать вариант наибольшей стоимости, то есть   
F(i, k)=max(F(i−1, k),F(i−1,k−wi)+pi).

Для хранения значения функции F можно использовать одномерный список, а не двумерный. Тогда можно будет убрать проверку условия F(i-1) <F(i), так как к следующему номеру i масса может только прибавиться или остаться. При этом массы предметов хранятся в списке W, их стоимости - в списке P. Будем считать (для простоты записи программы), что предметы пронумерованы от 1 до N.

Перебор:

**Алг** standart(W, P, backpack, items){

1. Создаём список F, который будет иметь k+1 элементов где с 1-й элемент — это рюкзак нулевого объема, а k+1-й элемент рюкзак с данный нашей задачей объемом. Сначала везде нули.
2. Для отслеживания помещаемых предметов создаём двумерный массив p[N][k+1], где k – объем ранца, n – количество предметов.
3. Так как мы мы решили использовать одномерный список F, то:

Начиная с i=0, Повторять:  
 Начиная с j=W[i], Повторять:

Если F(i,k)<F(i−1,k−Wi)+Pi:  
 F(i,k)=F(i−1,k−Wi)+Pi  
 p[i][k]+=1  
 Пока j<K+1

Пока i<n;

1. Для восстановления ответа будем перебирать все предметы «с конца» от n до 1. В переменной k будет храниться текущая вместимость рюкзака. Рассматривая предмет номер i определим, как было получено значение p(i, k). Пока p (i, k)=1. Предмет i помещаем в рюкзак, при этом значение k уменьшается на Wi. Циклом «пока» мы допускаем неограниченное добавление предмета в рюкзак. Переходя на k-W[i] мы ищем снова p(I,k)=1. Так мы просматривая p с конца, определяем какие предметы были помещены в рюкзак.

Жадный алгоритм:

Сортируем предметы по убыванию стоимости единицы каждого, где PCi - относительная стоимость единицы предмета i, Wi - вес предмета i, Pi - стоимость предмета i. Всего N предметов. Пытаемся поместить в рюкзак все что помещается, и одновременно наиболее дорогое по параметру PC. Оценим сложность метода. Для сортировки нам потребуется logN плюс проход по N предметам в цикле. Итого O(NlogN) - временная сложность выполнения алгоритма

# Текст программы

**/\* Файл DZ.py \*/**  
**import** time  
  
**from** item **import** Item  
**from** Backpack **import** Backpack  
  
**def** standart(W, P, backpack, items):  
 K=backpack.avalible\_weight  
 n=len(W)  
 Ans=[]  
 p=[[0]\*(K+1)**for** i **in** range(n)]  
 F = [0] \* (K+1)  
 **for** i **in** range(n):  
 **for** k **in** range(W[i], K+1):  
 **if** F[k]<F[k-W[i]]+P[i]:  
 F[k] = F[k - W[i]] + P[i]  
 p[i][k]+=1  
 i=n-1  
 k=0  
 **for** v **in** range (1, K+1):  
 **if** F[k]<F[v]:  
 k=v  
 **while** i>=0:  
 **if** p[i][k]==1:  
 **while** p[i][k] == 1:  
 k=k-W[i]  
 backpack.insert(items[i])  
 i=i-1  
 **return** backpack.current\_weight, backpack.full\_price  
  
**def** greedy(l, backpack):  
 items = l  
 items.sort(key=**lambda** x: x.price\_cooficient)  
 **for** i **in** items:  
 **while True**:  
 **if not** backpack.insert(i):  
 **break  
 return** backpack.current\_weight, backpack.full\_price  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 list\_of\_items = [Item(**'1'**, 2, 1), Item(**'2'**, 2, 3), Item(**'3'**, 3, 2), Item(**'4'**, 1, 1)]  
 W=[list\_of\_items[i].weight **for** i **in** range(len(list\_of\_items))]  
 P=[list\_of\_items[i].price **for** i **in** range(len(list\_of\_items))]  
 *#list\_of\_items = [Item("item " + str(i), random.random() \* 100, random.random() \* 1000, i) for i in range(1000)]*  
 bp = Backpack(7)  
 print(standart(W, P, bp, list\_of\_items))  
 bp.clear()  
 print(greedy(list\_of\_items, bp))

**/\* Файл Backpack.py \*/**

**from** funcs **import** custom\_sum  
  
  
**class** Backpack:  
 **def** \_\_init\_\_(self, avalible\_weight, items=[]):  
 self.avalible\_weight = avalible\_weight  
 self.items = items  
 self.current\_weight = 0  
 self.full\_price = custom\_sum(items, **lambda** item: item.price)  
  
 **def** insert(self, item):  
 **if** item.weight + self.current\_weight <= self.avalible\_weight:  
 self.items.append(item)  
 self.current\_weight += item.weight  
 self.full\_price += item.price  
 **return True  
 else**:  
 **return None  
  
 def** clear(self):  
 self.items = []  
 self.current\_weight = 0  
 self.full\_price = 0

**/\* Файл Item.py \*/**

**class** Item:  
 **def** \_\_init\_\_(self, name, weight, price):  
 self.name = name  
 self.weight = weight  
 self.price = price  
 self.price\_cooficient = self.weight / self.price

**/\* Файл timer.py \*/**

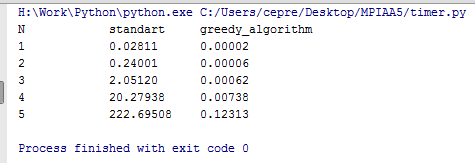
**import** timeit  
**import** DZ  
**import** random  
**from** item **import** Item  
**from** Backpack **import** Backpack  
**from** util **import** powers\_of  
  
**def** make\_header(func\_names):  
 **return " "**.join([**"{0:<12}"**.format(s) **for** s **in** [**"N"**] + func\_names])  
  
**def** make\_line(n, times):  
 **return " "**.join([**"{0:<12}"**.format(n)] + [**"{0:<12.5f}"**.format(t) **for** t **in** times])  
  
**def** time\_us(ns):  
 *"""Prints time table for given functions and inputs.  
 functions - dictionary of {func name: func(input)} - functions to time,  
 ns - list of n for which generate input,  
 generator - func(n) - input generation function,  
 repeats - number of times to call functions for each given input."""* print(make\_header([**'standart'**, **'greedy\_algorithm'**]))  
  
 **for** n **in** ns:  
 list\_of\_items = [Item(**"item "** + str(i), random.randint(1, 100), random.randint(1, 100)) **for** i **in** range(10\*\*n)]  
 W = [list\_of\_items[i].weight **for** i **in** range(10\*\*n)]  
 P = [list\_of\_items[i].price **for** i **in** range(10\*\*n)]  
 bp = Backpack(10000)  
 times = []  
 timer = timeit.Timer(**lambda**: DZ.standart(W, P, bp, list\_of\_items))  
 times.append(timer.timeit(1))  
 timer = timeit.Timer(**lambda**: DZ.greedy(list\_of\_items, bp))  
 times.append(timer.timeit(1))  
 print(make\_line(n, times))  
  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 time\_us(powers\_of(1, 0 , 4)

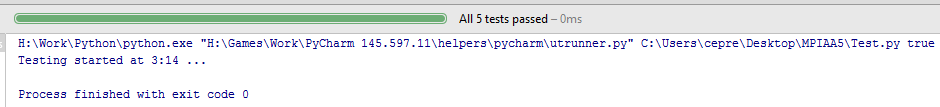
# Юнит-тесты

**/\* Файл Test.py \*/**

**from** Backpack **import** Backpack  
**from** item **import** Item  
**import** DZ  
**import** unittest  
  
**class** BackpackTests(unittest.TestCase):  
 **def** setUp(self):  
 self.backpack = Backpack(7)  
 self.backpack.clear()  
  
 **def** test\_empty(self):  
 self.assertEqual(self.backpack.current\_weight, 0)  
  
 **def** test\_insert(self):  
 self.backpack.insert(Item(**'1'**, 2, 4))  
 self.assertEqual(self.backpack.full\_price, 4)  
 self.assertEqual(self.backpack.current\_weight, 2)  
 self.assertEqual(self.backpack.items[0].name, **'1'**)  
  
 **def** test\_clear(self):  
 self.backpack.insert(Item(**'1'**, 2, 4))  
 self.backpack.clear()  
 self.assertEqual(self.backpack.full\_price, 0)  
 self.assertEqual(self.backpack.current\_weight, 0)  
 self.assertEqual(self.backpack.items, [])  
  
 **def** test\_greedy\_alg(self):  
 list\_of\_items = [Item(**'1'**, 2, 1), Item(**'2'**, 2, 3), Item(**'3'**, 3, 2), Item(**'4'**, 1, 1)]  
 self.assertEqual(DZ.greedy(list\_of\_items, self.backpack), (7, 10))  
  
 **def** test\_standart\_alg(self):  
 list\_of\_items = [Item(**'1'**, 2, 1), Item(**'T.E.A.'**, 2, 3), Item(**'3'**, 3, 2), Item(**'4'**, 1, 1)]  
 W = [list\_of\_items[i].weight **for** i **in** range(len(list\_of\_items))]  
 P = [list\_of\_items[i].price **for** i **in** range(len(list\_of\_items))]  
 self.assertEqual(DZ.standart(W, P, self.backpack, list\_of\_items), (7, 10))  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 unittest.main()

# Результаты замера времени работы алгоритма и тестов





# Выводы

Проделав работу, мы успешно построили алгоритмы для решения нашей задачи и реализовали его на языке программирования, после чего были сделаны юнит-тесты для проверки правильности работы алгоритма, а также сделаны замеры времени работы алгоритма. Было выяснено, что временная сложность перебора составляет O(N!) но в свою очередь перебор даёт точный результат. Временная сложность жадного алгоритма составляет O(NlogN) и является очень быстрым, но в свою очередь даёт не точный результат.