一． 问题定义

有 n 个物体，第 i 个物体价值为 p_i，重量为 w_i，其中 p_i 和 w_i 均为非负数。 背包容量为 m，m 为非负数。现需要考虑如何选择装入背包的物体，使装入背包的物体总价值 v 最大。

该问题形式化描述如下：

$$\max \sum_{i=1}^{n} p_i x_i \quad s.t. \quad \sum_{i=1}^{n} w_i x_i \le m$$

其中 $x_i \in \{0,1\}$ 表示是否将物体 i 装入背包中。

二． 0-1 背包问题算法

在这一章中，我们将分小节详细介绍解决 0-1 背包问题的算法，包括贪婪算法、动态规划、分支限界和遗传算法。对于每一个算法，我们将详细地分析其流程、复杂度和求解质量。需要特别声明的是，下面各算法的实验中，0-1 背包问题的规模是 1000，物体重量范围 1-100，价值范围 1-100，重量和价值独立无关。背包容量设置为物体总重量的二分之一。

2.1 贪婪算法

贪婪算法的流程如下：

（1） 计算每个物体的性价比并按照由大到小的顺序进行排序

```
15      # 计算每个物体的性价比并按照由大到小的顺序进行排序
16      ppr = {}
17      for i in range(1, n+1):
18          ppr[i] = p[i] / w[i]
19      sorted_ppr = sorted(ppr.items(), key=operator.itemgetter(1), reverse=True)
```

第 16 行计算性价比的时间复杂度是 O(n) ，第 17 行对物体按照性价比进行排序的时间复杂度是 O(n logn)。空间复杂度为 O(n)

（2） 每次尝试装入性价比最高的物品。如果不行，则跳过该物品，继续尝试下一个物品。

```python
    # 每次贪心地装入性价比最高的物体，直到装不下为止
    current_w, current_p, x = 0, 0, []
    for (k, ppr) in sorted_ppr:
        logging.debug("Object %d:  w: %d  p: %d  ppr: %.4f" %(k, w[k], p[k], ppr))
        if current_w + w[k] <= m:
            current_w += w[k]
            current_p += p[k]
            x.append(k)
```

该过程的时间复杂度是 O(n)

因此，该**贪婪算法的时间复杂度**是 O(n) + O(n logn) + O(n) = O(n logn)，空间复杂度为 O(n)。

贪婪算法的运行结果如下所示：

```
(venv) ➜ algorithm_project git:(master) python greedy_algorithm.py
INFO : 一共有1000个物品，背包重量为25433
INFO : 装入背包的总价值为: 41170
INFO : Using 0.0056s
```

2.2 动态规划算法

动态规划算法的流程如下：

（1） 设置动态规划初始条件

```python
    optp = [[0 for i in range(m+1)] for j in range(n+1)]
```

这里 optp[i][j]表示第 i 个物品装入载重量为 j 的背包中的最大价值。我们初始化 optp[0][*] 和 optp[*][0]，分别表示将第 0 个物品装入背包时的最大价值为 0，以及将物品装入载重量为 0 的背包的最大价值为 0。设置动态规划初始条件的时间复杂度为 O(nm)，空间复杂度为 O(mn)。

（2） 对于 optp[i][j]，如果我们选择不装入物品 i，那么问题就变为将第 i-1 个物品装入载重量为 j 的背包中的最大价值，即 optp[i-1][j]；如果我们选择装入物品 i，首先需要确保背

包的容量足够大，即 j>=w[i]。其次，这时候最大价值就变为将第 i-1 个物品装入载重量为 j-w[i]的背包的最大价值，加上物品 i 的价值，即 optp[i-1][j-w[i]]。这时候，将物品 i 装入背包的最大价值要大于不把物品 i 装入背包的最大价值，即 optp[i-1][j-w[i]]>optp[i-1][j]，我们才会装入物品 i。

```python
    for i in range(1, n+1):
        for j in range(1, m+1):
            optp[i][j] = optp[i-1][j]
            if j >= w[i] and optp[i-1][j-w[i]]+p[i] > optp[i-1][j]:
                optp[i][j] = optp[i-1][j-w[i]]+p[i]
```

该过程的时间复杂度是 O(nm)。

因此，该**动态规划算法的时间复杂度**是 O(nm) + O(nm) = O(nm)，空间复杂度为 O(mn)。

动态规划算法的运行结果如下所示：

```
(venv) ➜  algorithm_project git:(master) ✗ python dp_algorithm.py
INFO : 装入背包的总价值为: 41173
INFO : Using 14.4204s
```

2.3 分支限界算法

分支限界算法的流程如下：

（1） 将物品按照性价比进行排序

```python
    # 把物品按照性价比进行排序
    wp = [[i, w[i], p[i], p[i]/(w[i]*1.0)] for i in range(1, n+1)]
    sorted_wp = sorted(wp, key=lambda x : x[3], reverse=True)
```

排序的时间复杂度为 O(n logn)。

（2） 计算进行到第 j 步时的上界。

分支限界的每一步都是选择放入或者不放入某个物体。假设现在进行到第 j 步。如果背包被塞爆了，那么上界为 0，也就是说这个解是无效的。如果背包没有被塞满，我们的上界是这样计算的：往背包中依次放入剩下物品中性价比最高的物品，直到不能放入下一个为止。这

时候，我们放入下一个物品的一部分，这就是在当前状态下背包所能装入物品的最大价值了。

```python
def bound(curr_w, curr_p, j):
    if curr_w > m:
        return 0
    else:
        temp_w, temp_p = curr_w, curr_p
        while j < n and temp_w + sorted_wp[j][1] <= m:
            temp_w, temp_p, j = temp_w + sorted_wp[j][1], temp_p + sorted_wp[j][2], j + 1
        if j < n:
            temp_p += (m - temp_w) * sorted_wp[j][2] / (sorted_wp[j][1] * 1.0)
        return temp_p
```

计算上界的时间复杂度是 O(n)。

（3）分支限界每次取出堆中上界最高的节点，然后产生两个子节点，表示加入或者不加入下一个物品。

```python
43    h = [(0, 0, 0, -1, [])]  # 分别代表上界、重量、价值、深度和加入节点集合
44    while h:
45        f_b, f_w, f_p, f_j, f_s1  = heappop(h)
46
47        if f_j == n-1:
48            return f_p, f_s1
49
50        # 不加入 sorted_ppr 中的第 j+1 个节点
51        not_w, not_p, not_j = f_w, f_p, f_j+1
52        not_b, not_s1 = bound(not_w, not_p, not_j+1), f_s1
53        heappush(h, (-not_b, not_w, not_p, not_j, not_s1))
54
55        # 加入 sorted_ppr 中的第 j+1 个节点
56        in_w, in_p, in_j = f_w + sorted_wp[f_j+1][1], f_p + sorted_wp[f_j+1][2], f_j + 1
57        in_b, in_s1 = bound(in_w, in_p, in_j+1), f_s1 + [sorted_wp[f_j+1][0]]
58        heappush(h, (-in_b, in_w, in_p, in_j, in_s1))
```

第 44 行~58 行的 while 循环，循环体在最坏情况下，可能执行2^n次；

第 45 行、第 53 行和第 58 行的堆操作，时间复杂度为 O(logn)；

第 52 行、第 57 行计算上界，时间复杂度为 O(n)

因此，分支限界算法在最坏情况下的时间复杂度为 $O(n2^n)$，空间复杂度是 $O(n2^n)$

分支限界算法的运行结果如下所示：

```
(venv) ➜  algorithm_project git:(master) ✗ python bb_algorithm.py
INFO : 装入背包的总价值为: 41173
INFO : Using 0.2128s
```

2.3 遗传算法

遗传算法的流程如下：

（1） 产生初始解

```
def initial_pop():

    global best_fitn
    global best_indi
    temp_num = 0
    while True:
        temp_individual = [round(random.uniform(0, 1), 0) for i in range(n)]
        temp_weight = cal_weight(temp_individual)
        temp_value = cal_value(temp_individual)
        if temp_weight <= m:
            population.append(temp_individual)
            fitness.append(temp_value)

            if temp_value > best_fitn:
                best_indi = temp_individual
                best_fitn = temp_value
            temp_num += 1
            if temp_num >= pop_num:
                break
```

这里产生 pop_num 个初始解。在产生初始解的时候，还需要判断该初始解是否为合理解，即总重量不能超过背包的重量。判断是否为合理解的时间复杂度为 O(n)。因此，产生初始解的时间复杂度为 O(pop_num*n)，空间复杂度为 O(pop_num*n)；

（2） 使用轮盘赌算法选择个体。

```python
def select():

    global population
    global fitness
    global best_fitn
    global best_indi

    fitness_sum, upper_bound = 0, []
    for i in range(pop_num):
        fitness_sum += fitness[i]
        if fitness[i] > best_fitn:
            best_fitn = fitness[i]
            best_indi = population[i]
    upper_bound.append(fitness[0]/fitness_sum)
    for i in range(1, pop_num):
        upper_bound.append(upper_bound[i-1]+fitness[i]/fitness_sum)

    # 依据轮盘度算法产生新个体
    new_population, new_fitness = [], []
    new_population.append(best_indi)
    new_fitness.append(best_fitn)
    for i in range(1, pop_num):
        temp_num = random.uniform(0, 1)
        for j in range(pop_num):
            if temp_num < upper_bound[j]:
                new_population.append(population[j])
                new_fitness.append(fitness[j])
                break

    population, fitness = new_population[:], new_fitness[:]
```

计算轮盘赌的时间复杂度为 O(pop_num)，产生新个体的时间复杂度为 O(pop_num*pop_num*n)。因此，该步骤的时间复杂度为 O(pop_num*pop_num*n)

（3）对个体进行随机交叉

```
def crossover():
    for i in range(pop_num):
        if random.uniform(0, 1) <= crossover_prob:
            while True:
                j = random.randint(0, pop_num-1)
                point = random.randint(0, n-1)
                temp_indi1 = population[i][:point] + population[j][point:]
                temp_indi2 = population[j][:point] + population[i][point:]
                if cal_weight(temp_indi1) <= m and cal_weight(temp_indi2) <= m:
                    population[i], population[j] = temp_indi1[:], temp_indi2[:]
                    fitness[i], fitness[j] = cal_value(population[i]), cal_value(population[j])
                    break
```

对个体进行随机交叉产生的新个体，需要判断其是否为合理解，时间复杂度为 O(n)。因此该步骤的时间复杂度为 O(pop_num*n)

（4）对个体进行随机变异

```
def mutation():
    for i in range(pop_num):
        if random.uniform(0, 1) <= mutation_prob:
            point = random.randint(0, n-1)
            temp_indi = population[i][:]
            temp_indi[point] = 1 - temp_indi[point]
            if cal_weight(temp_indi) <= m:
                population[i] = temp_indi[:]
                fitness[i] = cal_value(temp_indi)
```

对个体进行随机变异产生新个体，需要判断其是否为合理解，时间复杂度为 O(n)。因此该步骤的时间复杂度为 O(pop_num*n)

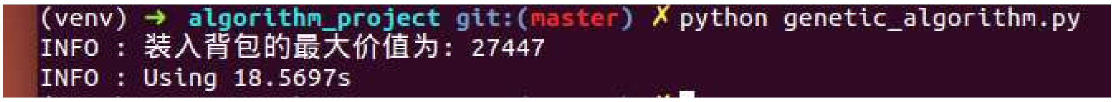
（5）循环运行 epoch 次，每次先使用轮盘赌算法选择新个体，然后对个体进行随机的交叉和变异。

```
    for i in range(epoch):
        select()
        crossover()
        mutation()
```

综上关于选择新个体、交叉和变异步骤的时间复杂度分析，我们可以得到该步骤的时间复杂度为 O(epoch*pop_num*pop_num*n)。

因此，**遗传算法的时间复杂度**是 O(epoch*pop_num*pop_num*n)

遗传算法的运行结果如下所示：



三． 0-1 背包问题算法的对比

从上面各算法的运行结果，对于 0-1 背包问题来说，我们得出以下结论：

- 贪婪算法是时间复杂度和空间复杂度最小的算法，同时也是逻辑最简单的算法。另外，在 0-1 背包问题上，贪婪算法算出来的最大价值并没有比动态规划算法或者分支限界算法差多少。因此，如果我们只是想寻找一个看起来还不错的解，同时又不想运行的时间太长，占用的内存空间太多的话，贪婪算法是一个不错的选择。
- 对于 0-1 背包问题来说，动态规划无论从时间复杂度还是空间复杂度上都是令人难以接受的，尽管其想法并不是很复杂。因此，如果我们的物品很少，同时又想简单快速地得到结果的话，可以考虑一下动态规划算法。
- 分支限界算法虽然在最快情况下，时间复杂度和空间复杂度为指数级。但是在我测试的多个结果中，都还能得到相对不错的结果。分支限界算法的优势是，其能得到准确的结果，同时一般时间复杂度和空间复杂度又不会很高。缺点就是实现起来会比动态规划和贪婪算法复杂一点点。因此，如果我们想得到准确解的话，推荐使用分支限界算法。
- 对于 0-1 背包问题来说，遗传算法是一个不能接受的算法。在我测试的所有样例中，遗传算法都没能在同等的时间内跑出相对不那么差的结果。究其原因，我认为 0-1 背包问题的解的取值只能取 0 或者 1，这使得遗传算法的交叉和编译过程沦为鸡肋。而这两个过程恰恰又是遗传算法的关键之处。