2022_AI導論_hw1

NE6101034　AI 所碩二　柳譯筑

P1. 如何實現 Brute Force

- Brute Force code
  - 我將所有的可能排列組合，將每種可能計算其 cost 再找出能產生最低成本的排列

```
def permutation(ele, l ,r):
    if l==r:
        g = ele.copy()
        cal_cost2(g)
    else:
        for i in range(l ,r+1):
            ele[l], ele[i] = ele[i], ele[l]
            permutation(ele, l+1, r)
            ele[l], ele[i] = ele[i] , ele[l]

def cal_cost2(chrom):
    global best_chromosome
    global best_cost
    cost = 0
    for i, pos in enumerate(chrom):
        cost += input[i][pos]
    if cost < best_cost:
        best_cost = cost
        best_chromosome = chrom
```

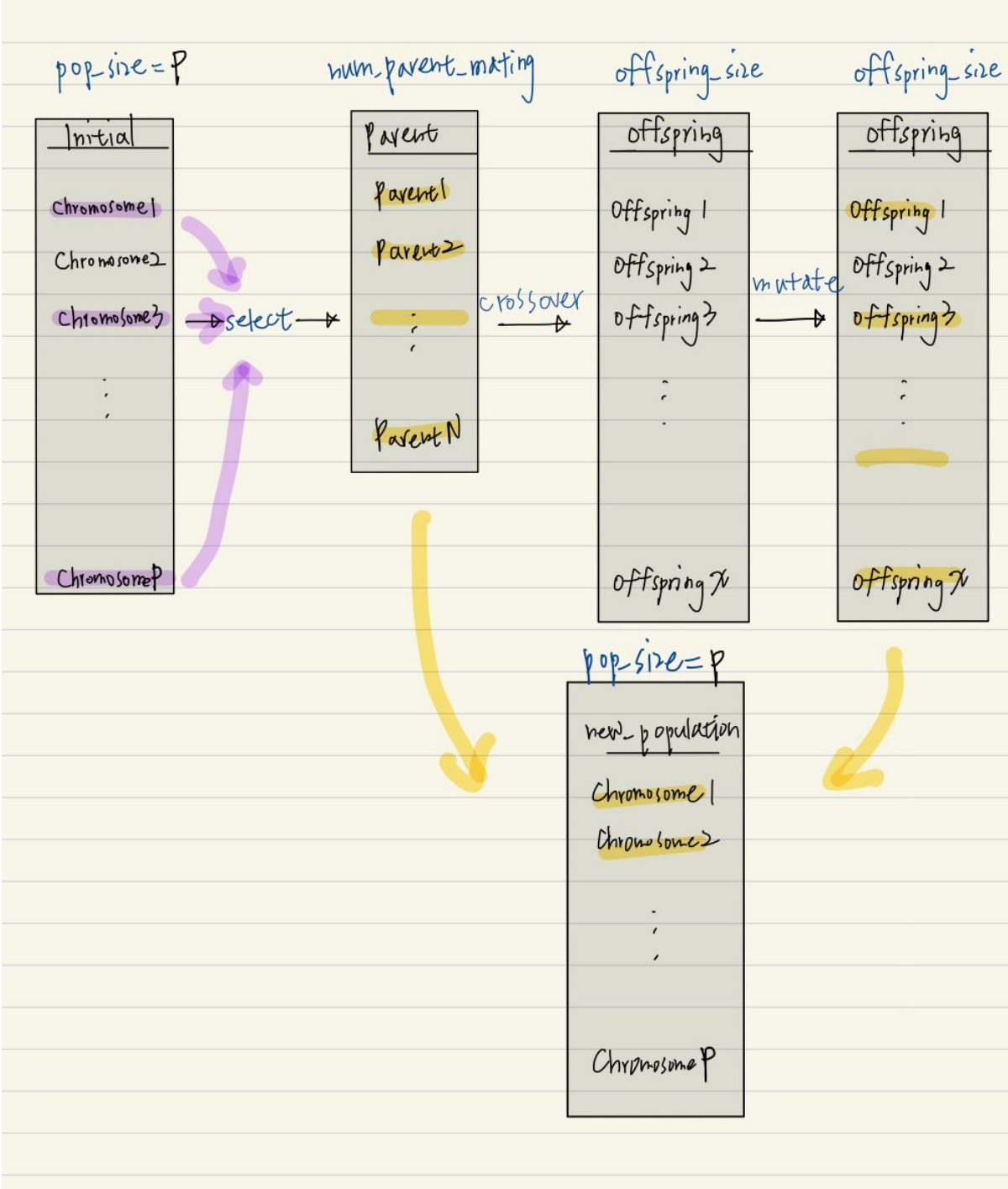
P2. 如何實現 Genetic Algorithm

Genetic Algorithm Pseudo code

```
# START
    # Generate the initial population
    # Compute fitness
    # REPEAT
    #     Selection
    #     Crossover
    #     Mutation
    #     Compute fitness
    # UNTIL num_generation
    # STOP
```

- **演算法解釋**:
  - initialize:

        - 一開始隨機排序 P 種分配方式（ P 種基因）
    - select:
        - 從 P 種分配方式中選出 N 種方式(N parents) 作為下一步的排列組合對象(mating)
    - crossover:
        - N 種分配方式中，倆倆對切開，把頭尾互相交換接上，offspring 為 parent 的兩倍
    - mutate:
        - 產生出來的 offspring 隨機的抽換某些排列 (基因突變)
    - new population:
        - 下一個遞迴使用一部分來自 parent 的排列組合（基因），一部分來自offspring 的排列組合（基因），選取方式則是用 fitness 當作順位選取



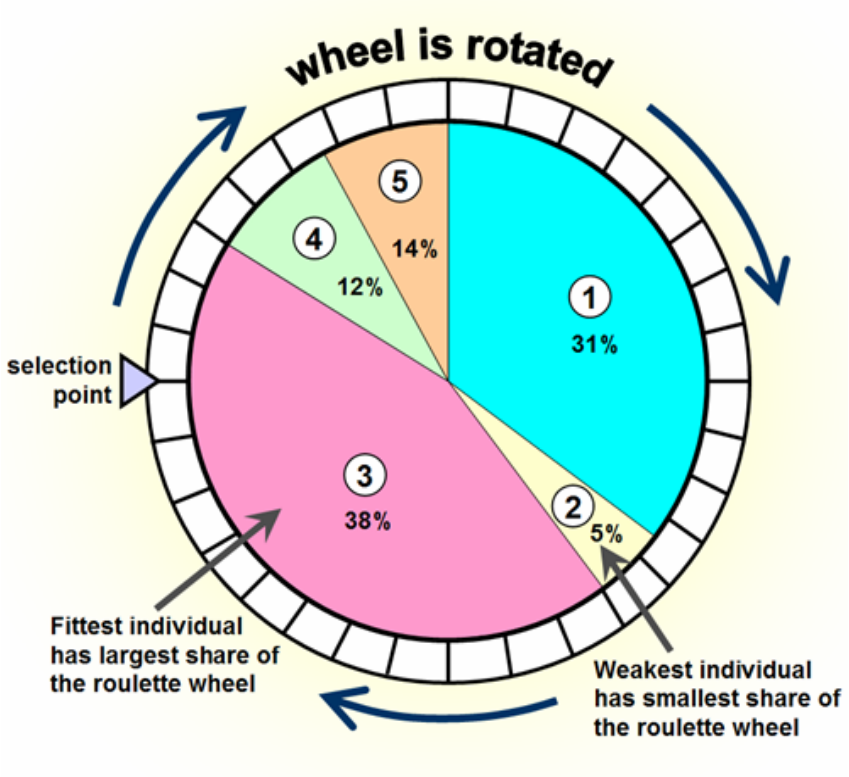
(A) Fitness function

- 由於這個問題的最佳解是工作時間和最低的分配方式，所以 fitness function不能用 maximum sum。我參考網路上的做法，先計算目前基因分配方法中總共的工時，接著去找所有基因之中時長最大值 Maximum，correct_fitness[i]= （Maximum - current_working_hour[i] ）+ pow(10, -5)
    - 最後的那個 10^-5 是為了防止接下來計算被選中的機率有分母為 0 的狀況

(B) Selection

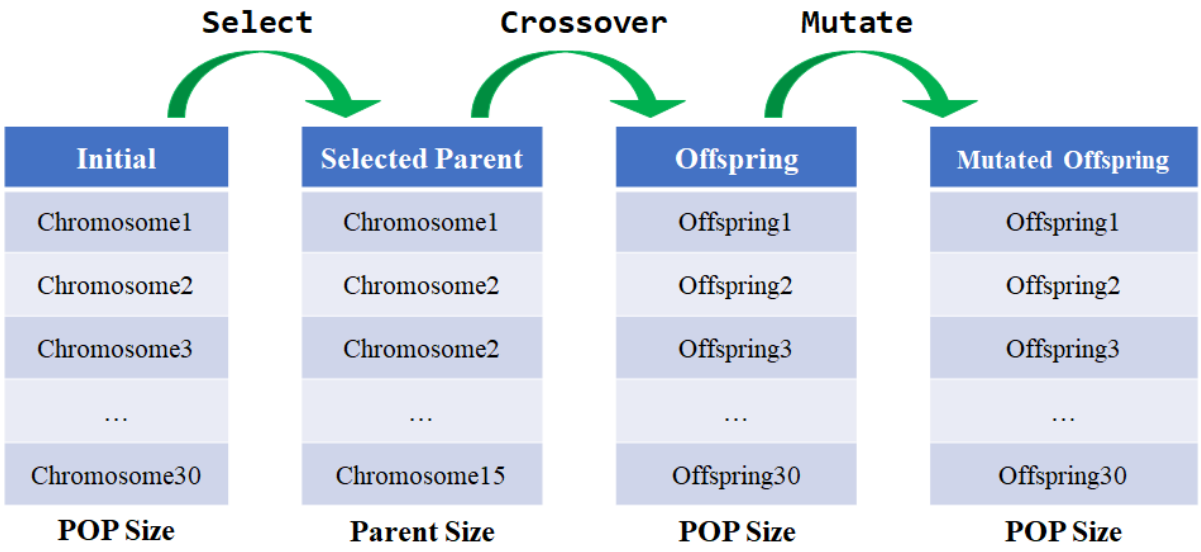
- **Roulette_wheel_selection**
  - chromosome 被選中的機率是 fitness / total_fitness
  - 根據這個 probability 選擇 N 個 parents 去做 mating

```
total_fitness = np.sum(fitness, axis=0)
chrom_probs = [fit/total_fitness for fit in fitness]
```



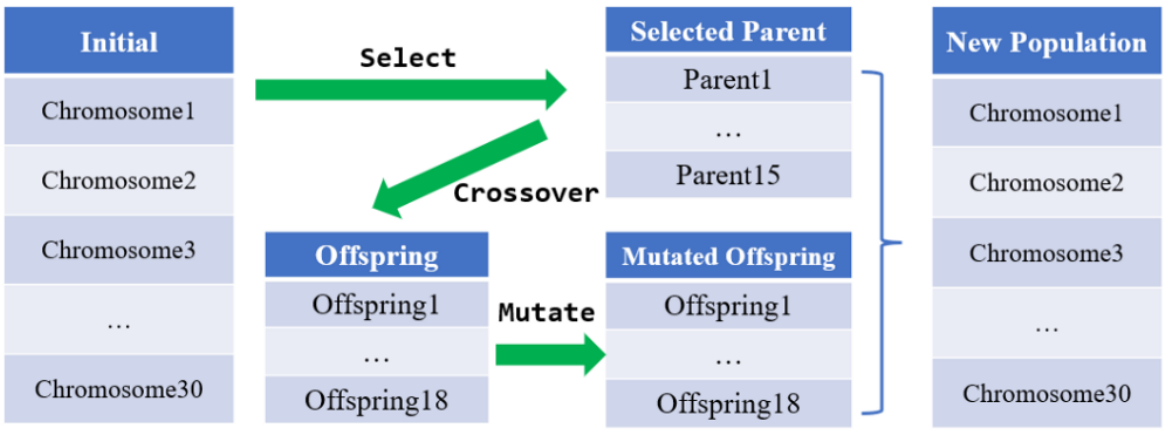
new population

- traditional 經典基因演算法流程
  - 首先根據設定的 population 進行 輪盤法 roulette wheel selection 隨機挑選 parent, 機率是根據 fitness 越大而越高
  - parent 兩兩交配產生 pop size 數量的後代 offspring
  - offspring 會基因突變 mutate
  - Mutate 過後的 offspring 成為下一代的 population



- Elitism selection 菁英挑選策略
  - 我實作的是菁英挑選策略，選擇完 fitness 最佳的 parent 後，從這群 parent 去做 crossover, mutation，產生下一代 generation 之後去計算 offspring 的 fitness，從這兩代中挑選出各自最優的幾條基因作為下一個 generation 的 population
  - 優點是可以保留好的染色體，然而一體兩面的缺點是，他容易陷入區域最佳解

菁英挑選策略演算法



(C) Crossover

- 由於編碼方式為整數編碼，所以要使用基於排序的 crossover 方式
- 我選擇了 **Order crossover** (proposed by Davis) , 與 D. Goldberg and R. Lingle 等人提出的 **Partially matched crossover(PMX)，**根據他人分析PMX擁有比大多數的crossover方法還要優異的表現，致使它成為使用頻率最高的方法
- **order crossover**
  - parent 兩兩交配，random 出一段要切開的基因位置
  - 直接交換 parent 這段位置的基因
  - 剩下的基因位置用 parent 2 還沒填進 child 過的基因，一一填入

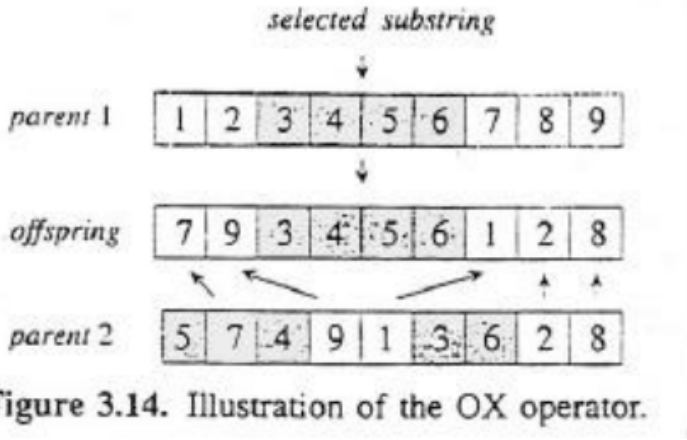
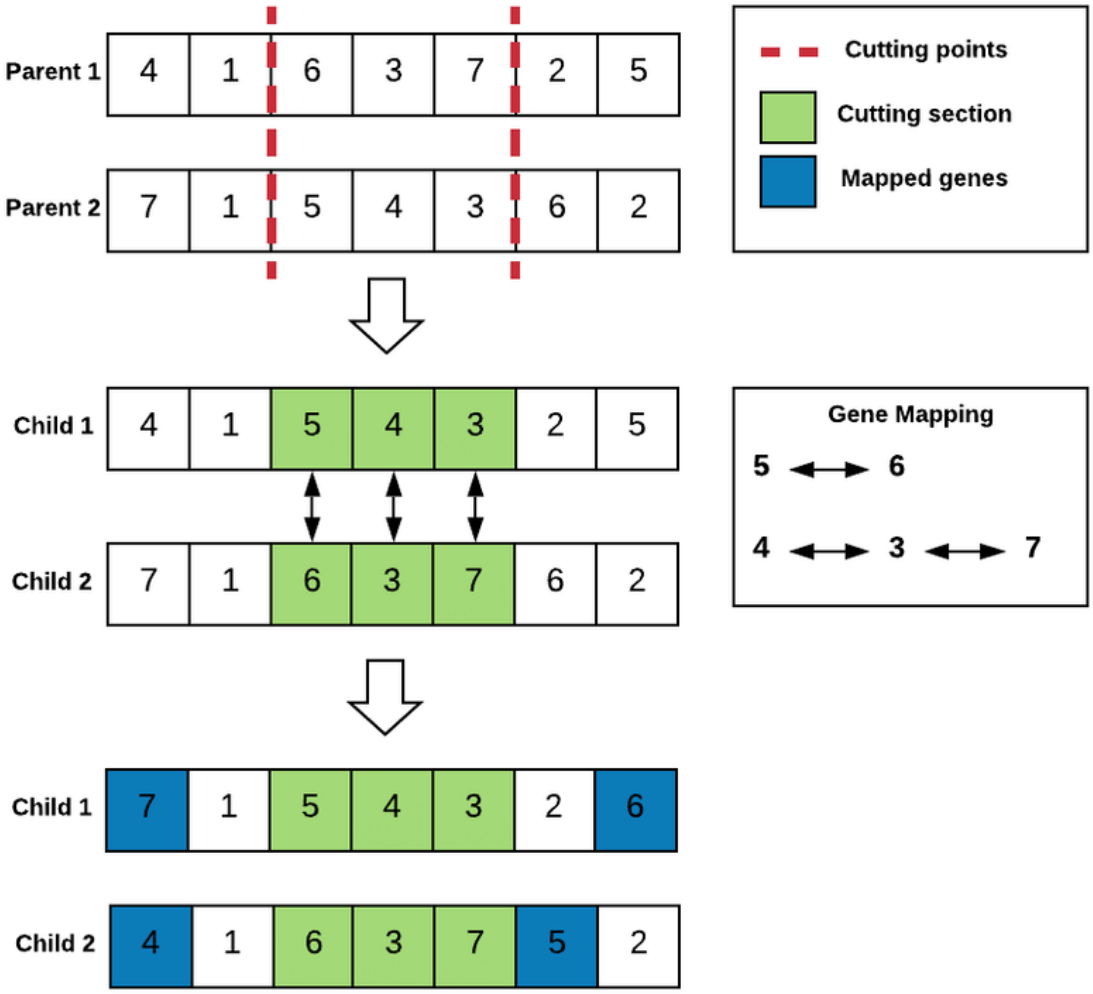


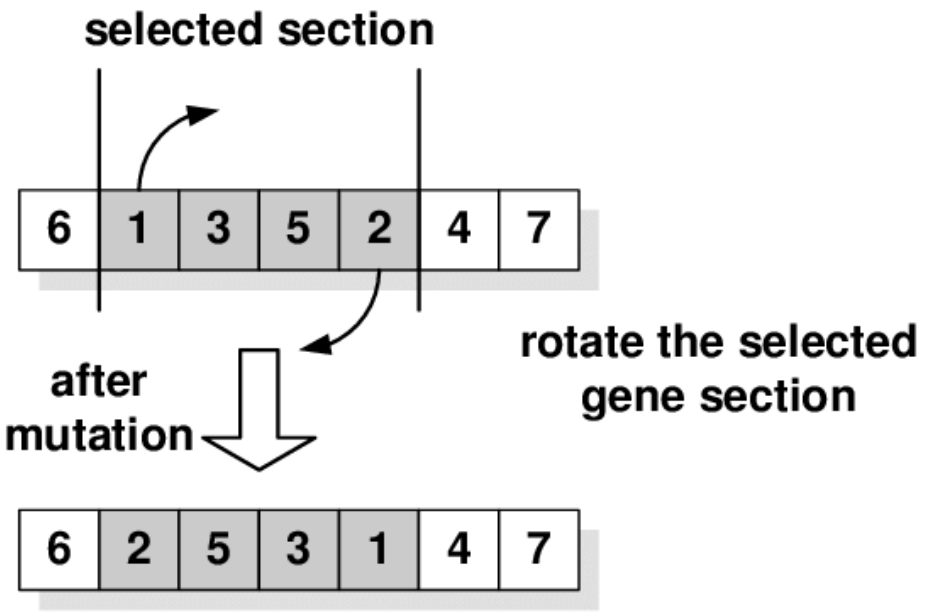
Figure 3.14. Illustration of the OX operator.

- **partial mapped crossover**
  - parent 兩兩交配，random 出一段要切開的基因位置
  - 直接交換 parent 這段位置的基因
  - 剩下的部分由於直接填入原本的基因會衝突，所以透過 map 去將直接交換的基因片段根據對應到的位置紀錄下來，例如下圖：5對應到6, 4 對應到 3 而 3 又對應到 7
  - 下圖所示，在 child 1 想填入 5 的時候就會衝突，這時候去看 map 到的表：5 ↔ 6，所以就直接找 6 (map[5])看看是否衝突，沒有衝突就跟 child 2 交換，因為 child 2 在填入 6 的時候也同樣會遇到衝突問題
  - 下圖所示，填入4的時候遇到衝突所以去找 map[4] = 3，而 3 又衝突到了，所以去找 map[3] = 7，7 沒有衝突所以就用 4 跟 child 2 交換 7

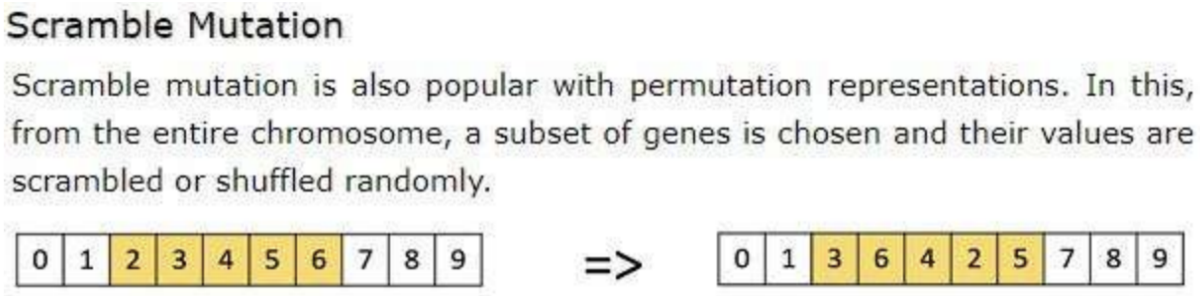


(D) Mutation

- 我選擇了兩種 mutation 實作
- **Inversion mutation**
  - 隨機選取一段基因，將他們排序顛倒



- **Scramble mutation**
  - 隨機取一段基因， 隨機的 shuffle 此段基因排序



P2 實作過程中觀察到什麼？

- Genetic Algorithm 主要可以針對兩個***變因*** 去做操作
  - 我選擇了以下幾個參數去做控制變因
    - Gene length (總共幾個 task 要完成)
    - population size (總共幾種 assignment)
    - num generation (總共做多少次的重新排列基因)
  - 以下是我使用的演算法

    - selection algorithm (Roulette_wheel_selection * Elitism selection)
    - crossover algorithm(Order crossover , Partial mapped crossover)
    - mutation algorithm (Inversion mutation, Scramble mutation)
- Genetic Algorithm 的***變數***
    - 時間複雜度
    - 結果是否掉入區域最佳解
- **實驗數據**

  1. [調整參數] 將每個不一樣排列組合的參數各做 100 次取平均所得到的最佳 cost 結果

Gene length	Generation_number	population size	average time	average of best cost
				assignment best **cost : 0.9086022396091811**
3	**3**	10	0.0009831690788269043	0.9086022396091807
3	**10**	10	0.00403418779373169	0.9086022396091807
3	10	**3**	0.0014972710609436036	0.9099881113493976
3	10	**5**	0.0019539523124694823	0.9086022396091807
				assignment best cost: 15
4	**10**	10	0.0037491774559020997	15
4	**20**	10	0.008858623504638672	15
4	**30**	10	0.012982118129730224	15
4	10	**3**	0.0018680024147033692	15.58
4	10	**10**	0.004684207439422608	15.58
4	10	**20**	0.008461217880249023	15
				assignment best **cost: 1.45020202**
7	**10**	10	0.007272663116455078	1.5743536610000017
7	**100**	10	0.07123545885086059	1.4551509783000032
7	**200**	10	0.13901139974594115	1.458944825200003
7	**500**	10	0.29672234058380126	1.4567591239000033
7	**800**	10	0.49085866928100585	1.450202020000003
7	100	**3**	0.021055774688720705	1.5450908826000016
7	100	**10**	0.07123545885086059	1.4551509783000032
7	100	**20**	0.1310188102722168	1.4567591239000033
7	100	**100**	0.6075157976150513	1.454204381400003

2. [調整 Crossover, Mutation 演算法]
    - Crossover
        - 舉例列出 order crossover 跟 partial mapped crossover 所產生的 offspring

```
Order_crossover offspring
len = 4
[[2 0 3 1]
[0 2 3 1]
[0 2 3 1]
[0 1 3 2]
[0 2 3 1]
[0 1 2 3]
[0 2 3 1]
[0 2 3 1]
[0 2 3 1]
[0 2 3 1]]

len = 10
[[ 2 11  0  9  8  6  5 12  7 13  4 10 14  3  1]
 [ 2 11  0  9  8  6  5  7 13  1  4 12 10 14  3]
 [ 2 11 12  8  6  5  7  0  9  3  1 10 14  4 13]
 [11  3  0  2  8  6  5  7  9 13  4 12  1 10 14]
 [11  4  6  0 13  2  5 10  1  9  3  8 12 14  7]
 [ 2 11  0  9  8  6  5 12  7 13  4  3  1 10 14]
 [11  8  6 12  0 13  2  5 10  1  9  7  4 14  3]
 [ 2 11  0  9  8  6  5 12  7 13  4 10  1 14  3]
 [ 3 11  0  2  8  6  5  7  9 13  4 12 10 14  1]
 [ 2 11  8  6  5  0  9 12  7 13  4 10  1 14  3]]

-
Partial_mapped_crossover offspring
len = 4
[[0 2 1 3]
 [0 2 1 3]
 [2 0 3 1]
 [0 3 2 1]
 [0 2 1 3]
 [0 2 1 3]
 [2 0 3 1]
 [0 3 2 1]
 [0 2 1 3]
 [0 2 1 3]]

len = 10
[ 8 11  1  5  6  4  0 12  7 13  3 10  9  2 14]
 [ 2  4 12 11 10  7  3  9  0 14 13  8  1  6  5]
 [ 2  4 11 12 10  7  3  9  0 14 13  8  1  6  5]
 [ 2  4 12 11 10  7  3  9 13 14  0  8  1  6  5]
 [ 8 11  0  4  9 13  6 10  2 14 12  7  1  3  5]
 [ 8 11  1  5  6  4  0 12  7 13  3 10  9  2 14]
 [ 2  4 12 11 10  7  3  9 13 14  0  8  1  6  5]
 [ 2  4 12 11 10  7  3  9  0 14 13  8  1  6  5]
 [ 8 11  0  4  9 13  6 10  2 14 12  7  1  3  5]
 [ 2  4 12 11 10  7  3  9  0 14 13  8  1  6  5]]
```

- mutation

- 舉例列出 inversion mutation 跟 scramble mutation 後的 offspring

```
len= 10
before inversion mutation
[[ 6  1 11  2  5  8  4 12 14  9 10  7  0 13  3]
 [ 4  0  9  1 10  8  7 11  2  5 12  3 13 14  6]
 [ 6  0  2 11  5  8  4 12 14  9 10  7  1 13  3]
 [ 6  1 11  2  5  8  4 12 14  9 10  7  0 13  3]
 [ 6  0  2 11  5  8  4 12 14  9 10  7  1 13  3]
 [ 6  0  2 11  5  8  4 12 14  9 10  7  1 13  3]
 [ 7 14 13 11  0  9  1  2  8  3 12  4  6 10  5]
 [ 6  1 11  2  5  8  4 12 14  9 10  7  0 13  3]
 [ 6  0 11  2  1  7 10  9 14 12  4  8  5 13  3]
 [ 6  1 11  2  5  8  4 12 14  9 10  7  0 13  3]]
after inversion mutation
[[ 6  1 11  2  5  8 13  0  7 10  9 14 12  4  3]
 [ 4  0  9  1 10  8 14 13  3 12  5  2 11  7  6]
 [ 6  0  2 11  5  8 13  1  7 10  9 14 12  4  3]
 [ 6  1 11  2  5  8 13  0  7 10  9 14 12  4  3]
 [ 6  0  2 11  5  8 13  1  7 10  9 14 12  4  3]
 [ 6  0  2 11  5  8 13  1  7 10  9 14 12  4  3]
 [ 7 14 13 11  0  9 10  6  4 12  3  8  2  1  5]
 [ 6  1 11  2  5  8 13  0  7 10  9 14 12  4  3]
 [ 6  0 11  2  1  7 13  5  8  4 12 14  9 10  3]
 [ 6  1 11  2  5  8 13  0  7 10  9 14 12  4  3]]

len = 10
before scramble mutation
[[11 10 14  0  6  4 13 12  5  8  7  3  1  9  2]
[ 9 11 13  6  0  2  8  4  1 14 10  7 12  3  5]
[ 9 11 13  6  0  2  8  4  1 14 10  7 12  3  5]
[ 4 11  1  5  7  2 13  0 12  6 10 14  9  3  8]
[10  8 12 13  1 11  7  0  6  2  9  5 14  3  4]
[11 10 14  0  6  4 13 12  5  8  7  3  1  9  2]
[ 9 11 13  6  0  2  8  4  1 14 10  7 12  3  5]
[11 10 14  0  6  4 13 12  5  8  7  3  1  9  2]
[ 9 11 13  6  0  2  8  4  1 14 10  7 12  3  5]
[11 10 14  0  6  4 13 12  5  8  7  3  1  9  2]]
after scramble mutation
[[11 10 14  0  6  4 13 12  5  9  3  7  1  8  2]
 [ 9  2  6  0 11 13  8  4  1 14 10  7 12  3  5]
 [ 9 11 13  6  0  2  8  4  1 14 10  7 12  3  5]
 [ 4 11  1  5  7  2 13  0 12  6 10  9 14  3  8]
 [10  1  6 12  8  9 11 13  2  0  7  5 14  3  4]
 [11 10 14  0  6  4 13 12  5  8  7  3  1  9  2]
 [ 9 11 13  6  0  2  8  4  1 10 14 12  7  3  5]
 [11 10 14  0  6 13  4 12  5  8  7  3  1  9  2]
 [ 2  8  0  6 11 13  9  4  1 14 10  7 12  3  5]
 [11 10 14  0  6  4 13 12  5  7  8  1  3  9  2]]
```

以下是我所觀察到的結果：

- 參數：
  - 透過實驗數據可以看出，基因演算法的時間複雜度正比於 O(generation * (population + O(crossover) + O(mutation))

    - pop size 越大，可以容納的可能性越多，最後得到的結果會比較接近全域最佳解，但是缺點是時間會以倍數線性成長
    - num generation 越高也可以提高最佳解為全域的可能，但是效益彰顯的程度比較沒有直接升高 pop size 來的那麼大，因為菁英挑選演算法在經歷過很多個 generation 最後可能會只保留優秀的基因導致最後的population variance 很小
- 演算法：
    - 可以看出 **crossover** 過後產生的 offspring 還是很受一開始 initialize 的狀況影響，不過由於實驗使用的基因不是非常大的數據，因此還看不出不同的演算法之間會帶來效益上多大的差距
    - 在比較短的基因中，mutation 除非設定會進行很大的突變，否則往往不太會變動，卡在區域最佳解

P3. 比較兩個方法的優缺點，實作時的心得

- **時間分析**
    - Gene algorithm 用 pop size = 10, num generation = 10 的參數下去實驗

	Brute Force	Gene algorithm
Gene length = 3	0.0000178098678588	0.0017978239059448242
Gene length = 4	0.000064659118	0.00215792179107666
Gene length = 7	0.019713878631591797	0.0031629180908203124
Gene length = 8	0.1510333776473999	0.00349334716796875
Gene length = 10	15.59584949016571	0.004161179065704346
Gene length = 15	-	0.005881037712097168
Gene length = 100	-	0.03524187564849854

    - **Brute Force**
        - 剛開始先用陣列儲存所有可能性再計算數量很小就 memory error 了，因此要一邊 permutation 一邊計算cost
        - 15! 所需要計算的複雜度就到了 **1.3076744e+12 * 15**, 計算量過於龐大費時
    - 在時間上很明顯看出當基因序列很長的時候，可以凸顯基因演算法的優點
- **最佳解分析**
    - 進行 100 次 iteration 平均結果，會根據不同基因長度給予不同參數值
    - 我設定 Gene length = 100的 input 規則為
        - 編號 < 50 → cost = 編號

編號 == 50 → cost = [200, 200, …, 200]

編號 >50 → cost = 100 - 編號

最佳解為 1225+ 1225 - 49 + 200 = 2601

	Brute Force (correct answer)	Gene algorithm
Gene length = 3	28	28.0
Gene length = 4	15	15.0
Gene length = 8	13	14.12
Gene length = 10	5	6.3
Gene length = 10	30	31.04
Gene length = 100 (iteration =3, pop size = 100, num generation = 100)	2601	3934.58

	Brute Force	Genetic Algorithm
Pros	1. 將所有可能性列出並計算過，為全域最佳解 2. 幾乎無需思考，不需要理解 domain knowledge直接對問題列出所有可能性硬解，不容易出錯	1. 當基因序列很長的時候，可以凸顯基因演算法的優點 2. GA ∝ O(generation * (O(population)+O(mutation)+O(crossover))，根據不同狀況與需求自行設定參數 3. 與不同問題領域無關，只需針對不同問題進行 fitness 設計 4. 具有隨機性，靈活性高
Cons	1. BF = O(n!) 計算量龐大 2. 效率低下	1. 演算法對初始種群的選擇有一定的依賴性 2. 不一定是全局最優解，可能會落到區域最佳解，這時候就需要基因演算法的其他變形去改善這個問題 3. 需要進行編碼與解碼 4. Crossover, mutation 等演算法都有許多參數需要手動調整，調整沒有一定的答案往往需要透過經驗去設定 5. 沒有變形過的遺傳演算法容易較早收斂，在優良個體可以保留與保持族群的多樣性之間達到平衡是比較困難的事
適合的狀況	基因序列較小的 task 需要完全準確的最佳解時	基因序列較長的 task 可以將問題描述為一個合理的 fitness function(很多問題無法求導，這時fitness function就展現了他的優點)

Reference

- 以Python實作基因演算法(Genetic Algorithm , GA)並解決工作指派問題(Job Assignment Problem, JAP) | by Carrot Cheng | Carrot Cheng的數據分析 | Medium
- chap-8-9 (uab.cat)