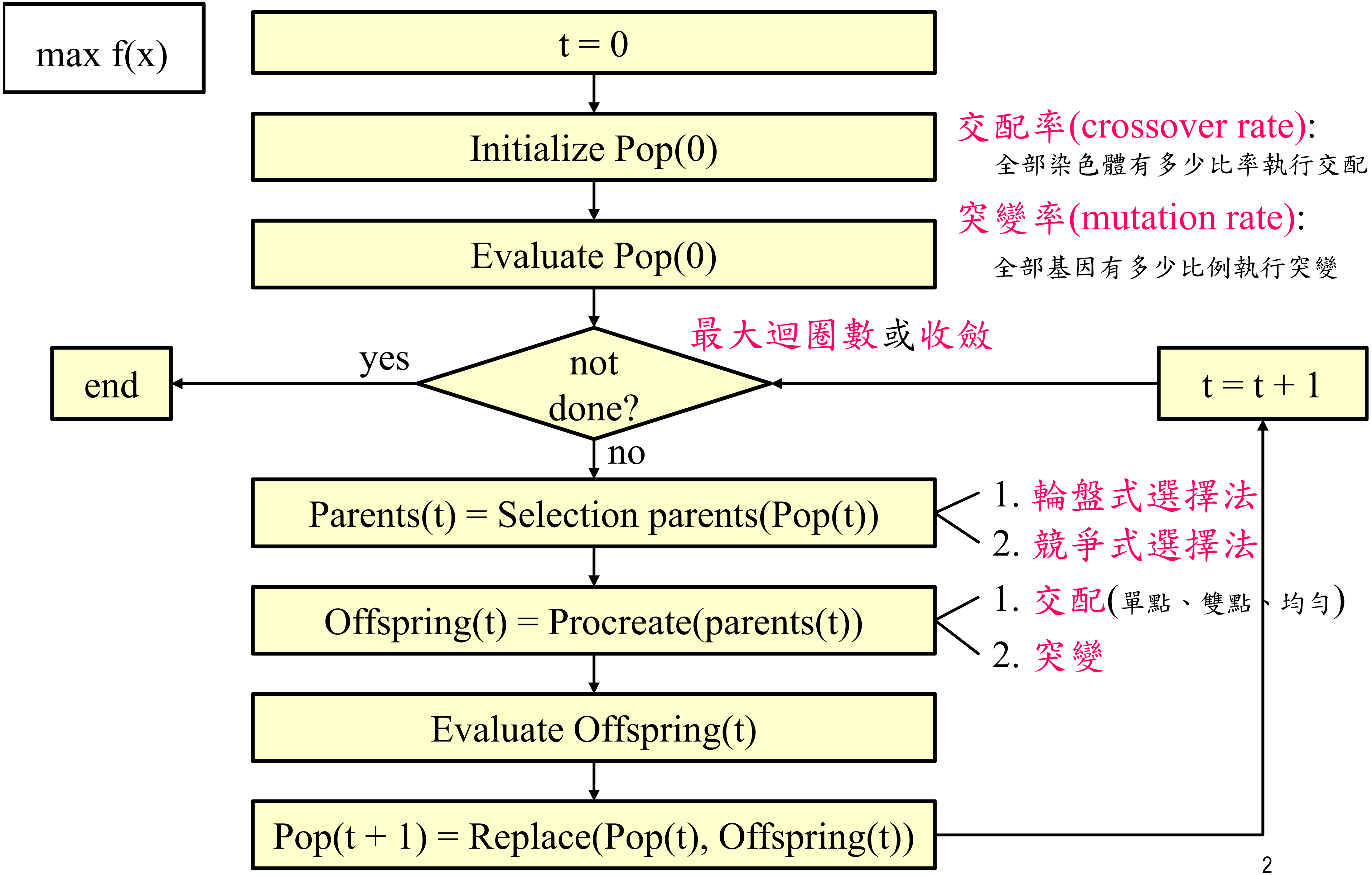
GA使用排列解的編碼 與Traveling Salesman Problem (TSP)

基因演算法流程圖



max f(x)

t = 0

Initialize Pop(0)

交配率(crossover rate):
全部染色體有多少比率執行交配

Evaluate Pop(0)

突變率(mutation rate):
全部基因有多少比例執行突變

not done?

最大迴圈數或收斂

yes

end

no

t = t + 1

Parents(t) = Selection parents(Pop(t))

1. 輪盤式選擇法
2. 競爭式選擇法

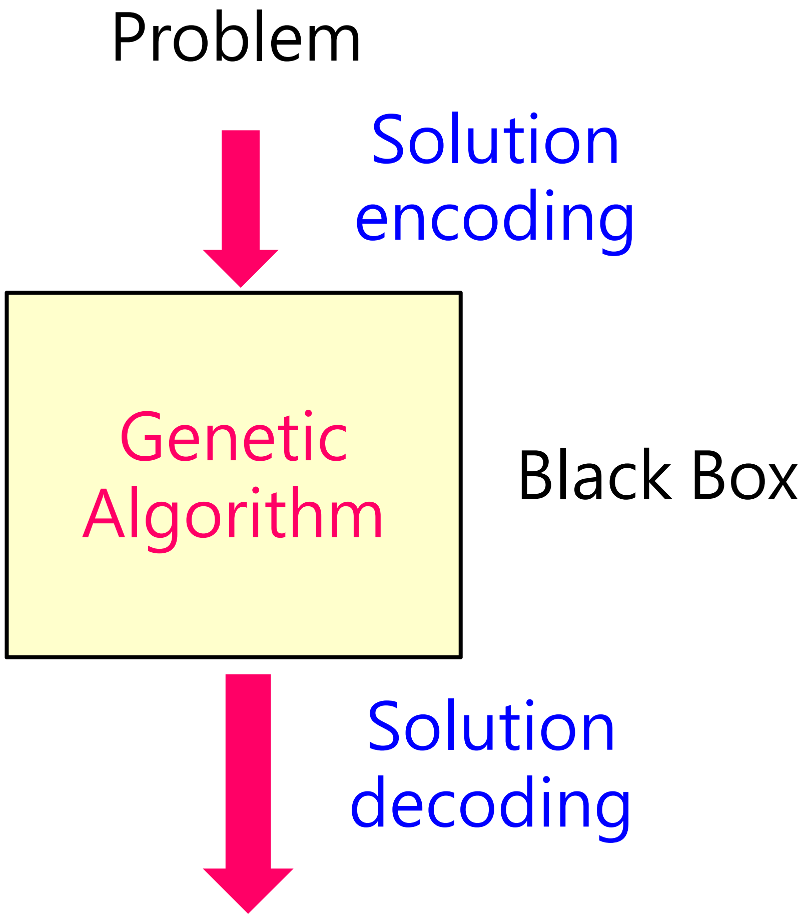
Offspring(t) = Procreate(parents(t))

1. 交配(單點、雙點、均勻)
2. 突變

Evaluate Offspring(t)

Pop(t + 1) = Replace(Pop(t), Offspring(t))

Framework of using the GA



Problem

Solution encoding

Genetic Algorithm

Black Box

Solution decoding

- Optimal solutions or approximation solutions
- Could solve large-scale problems

Solution encoding/decoding

- Encoding continuous decision variables

3.6	7.2	4.9	1.3	2.9

- Encoding discrete decision variables

1	0	0	1	1

3	7	4	3	2

- Encoding permutation solutions

3	1	5	4	2

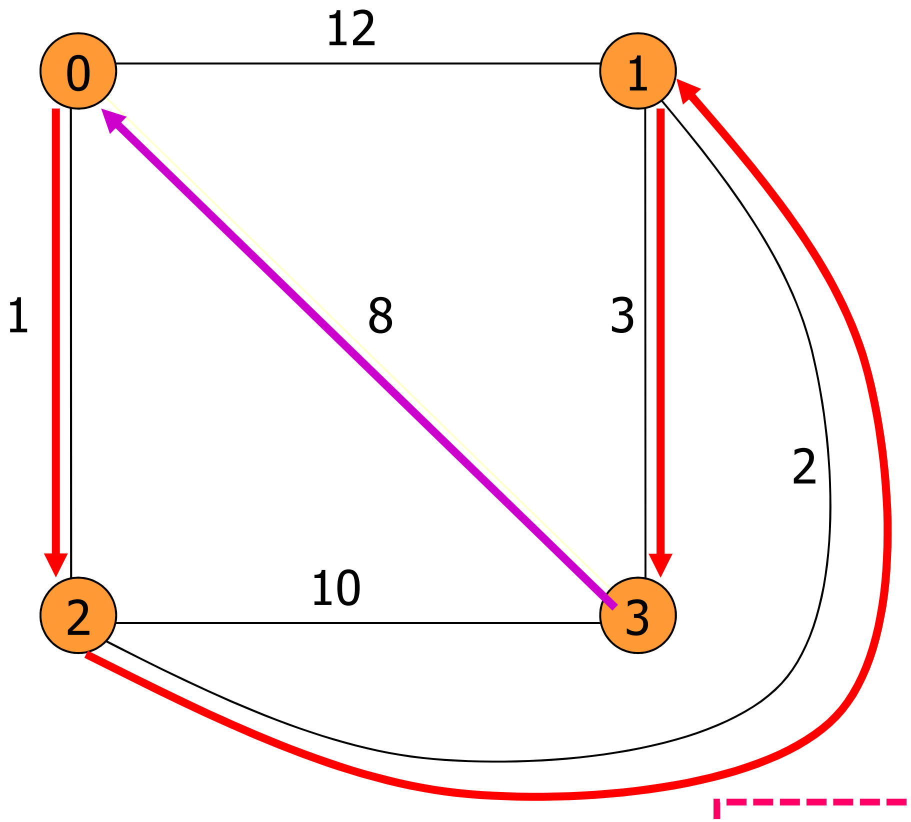
- Mixed encoding

3	1	5	4	2

1	0	0	1	1

Traveling Salesman Problem (TSP):

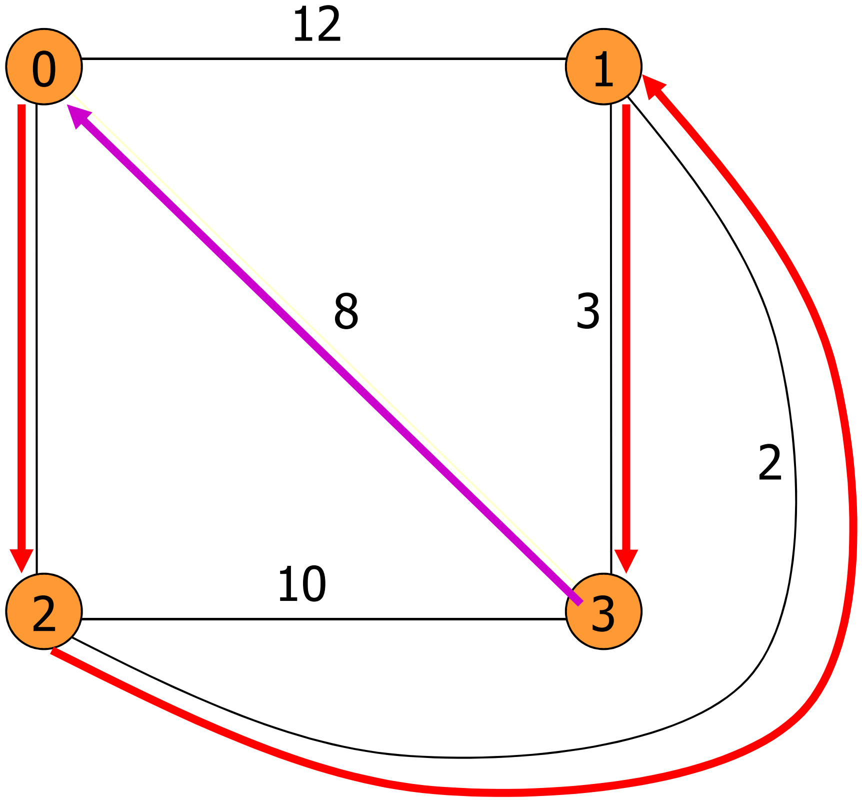
- Find the shortest tour that passes each city, and begins and ends at the same city (0)



TSP is NP-complete

Encoding a solution for TSP

- A permutation of all city IDs
  - e.g., (0, 2, 1, 3)

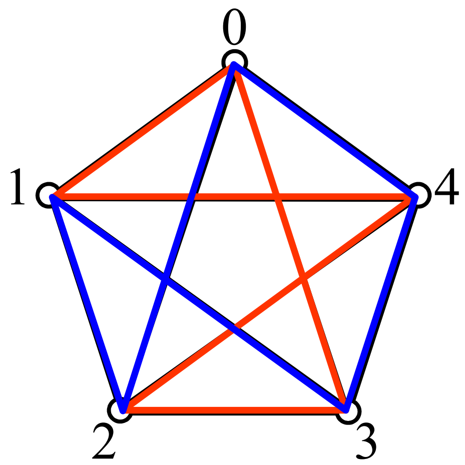


- But, how to conduct the crossover and mutation operators on permutation?

Uniform crossover in the TSP

- Single-point crossover has a high chance to generate illegal permutation solutions

- Uniform crossover can often be modified to avoid this problem
  - E.g. in TSP with simple path coding:
    - ✓ Where mask is 1, copy cities from one parent
    - ✓ Where mask is 0, choose the remaining cities in the order of the other parent



mask

0 1 0 1 1 　　 0 1 0 1 1

母 0 1 4 | 2 3 　　 父 0 2 1 | 3 4

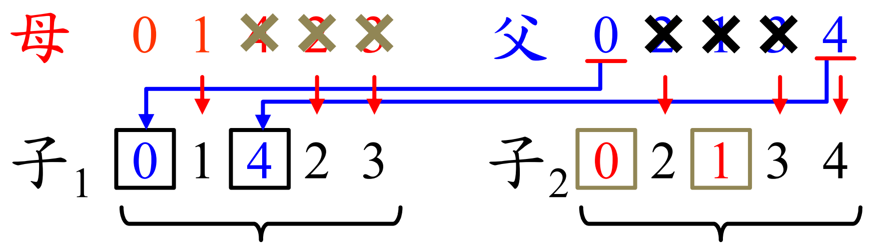
① 單點交配

子$_1$ 0 1 4 3 4 　　 子$_2$ 0 2 1 2 3

illegal！

② 均勻交配

mask 0 1 0 1 1 　　 0 1 0 1 1



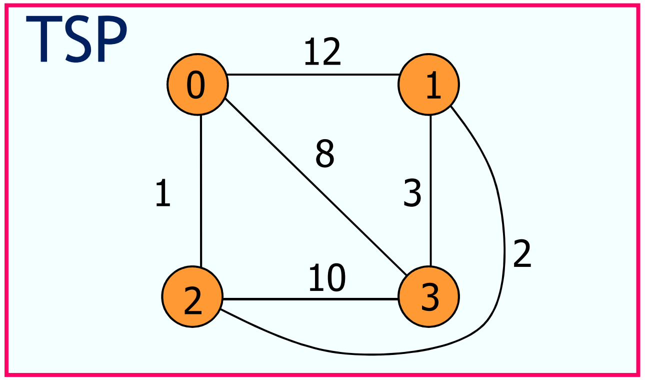
母 0 1 ~~4~~ ~~2~~ ~~3~~ 　　 父 0 ~~2~~ ~~1~~ ~~3~~ 4

子$_1$ 0 1 4 2 3 　　 子$_2$ 0 2 1 3 4 　　 legal

不重複 　　 不重複

使用排列解的GA求解

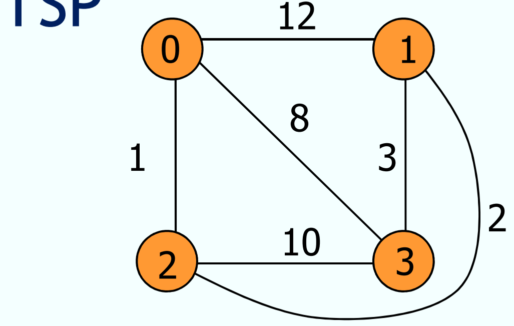
1. Coding a chromosome
   - 為{0, 1, 2, 3}的一個排列，例如 (0, 3, 1, 2)，即為城市走訪順序
   - 初始化(X,Y)：隨機產生一個{0, 1, 2, 3]的排列
   - 修復不可行解：不用處理
2. Fitness function
   - 根據編碼走訪的城市順序，可計算出此解的成本
     → Fitness = -成本
3. Selection
   - 用競爭式選擇法或輪盤式選擇法選出
4. Reproduction
   - 交配：用TSP作法的均勻交配產生合法子代
   - 突變：任選某一染色體中的任意二基因互換
5. Replacement
   - `Pop(t+1) = {Pop(t)∪{kids}}-{worsts}}`

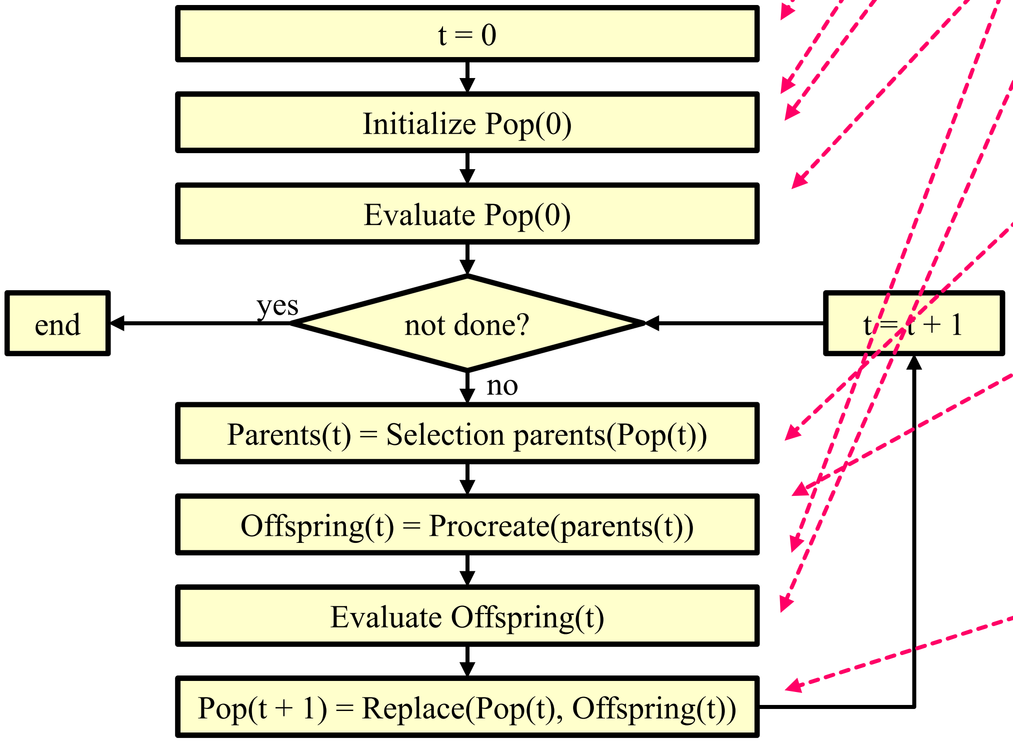


max f(x)

基因演算法流

TSP





t = 0
Initialize Pop(0)
Evaluate Pop(0)
not done?
yes
end
no
t = t + 1
Parents(t) = Selection parents(Pop(t))
Offspring(t) = Procreate(parents(t))
Evaluate Offspring(t)
Pop(t + 1) = Replace(Pop(t), Offspring(t))

1. 編碼：為{0,1,2,3}的一個排列，例如 (0, 3, 1, 2)

初始化：隨機產生{0, 1,2,3]排列

修復不可行解：不用處理

2. 適應度：根據編碼走訪的城市順序，可計算出此解的成本 → Fitness = -成本

3. 選擇：用競爭式選擇法或輪盤式選擇法選出

4. 繁衍：

- 交配：用均勻交配生合法子代
- 突變：任選某一染色體任意二基因互換

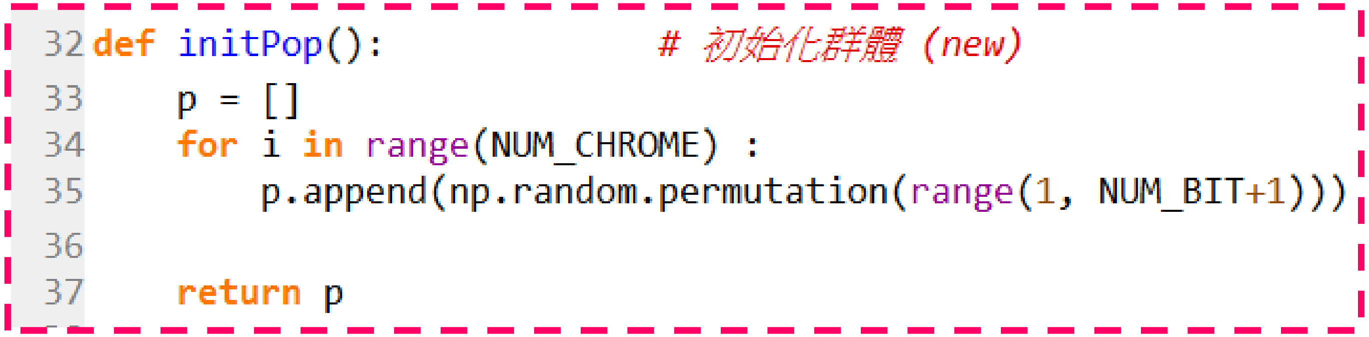
5. 取代：

$$Pop(t+1) = \{Pop(t) \cup \{kids\}\} - \{worsts\}$$

Python code for solution representation

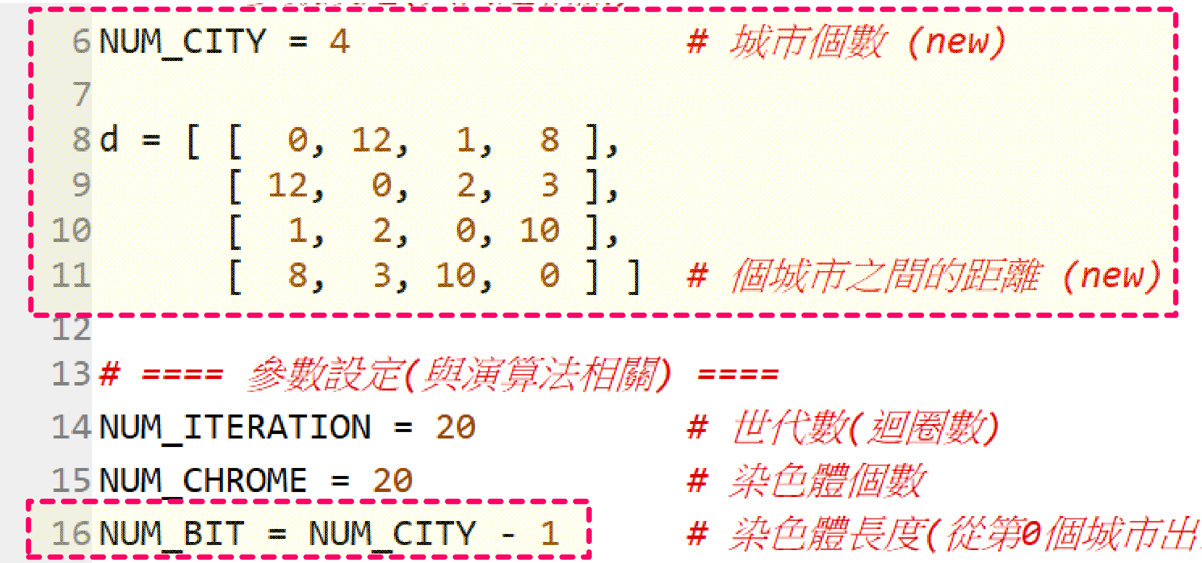
- 編碼與初始化
  - 隨機產生{0, 1, 2, 3]排列
  - 因為起頭和結尾都是city 0，
    所以只考慮其他3城市，編碼為(X[0], X[1], X[2])
  - pop = $\begin{bmatrix} X_0[0] & X_0[1] & X_0[2] \\ X_1[0] & X_1[1] & X_1[2] \\ \dots & \dots & \dots \\ X_n[0] & X_n[1] & X_n[2] \end{bmatrix}$,

    當中$X_i[j]$表示第i個染色體的第j個基因，
    $[X_i[0], X_i[1], X_i[2]]$為1, 2, 3的隨機排列

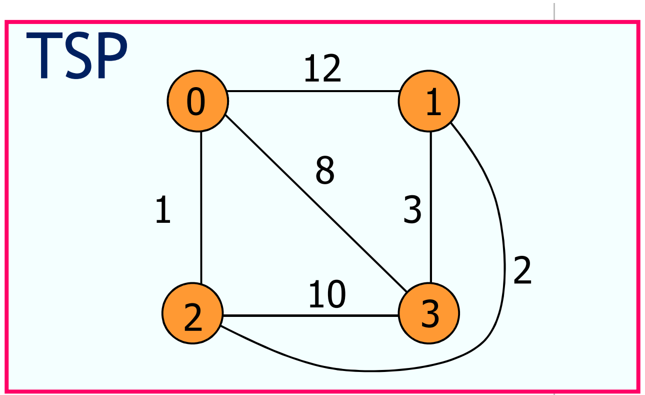


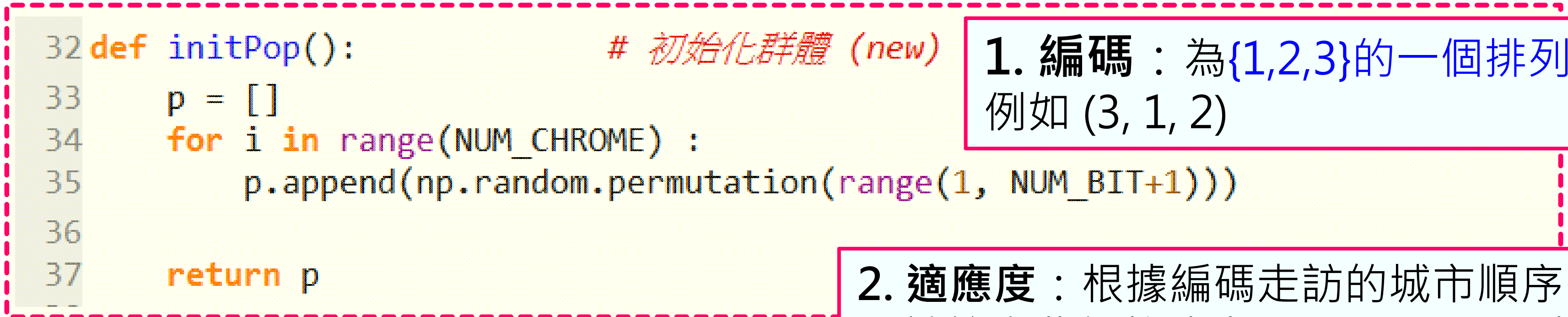
```
def initPop():                  # 初始化群體 (new)
    p = []
    for i in range(NUM_CHROME) :
        p.append(np.random.permutation(range(1, NUM_BIT+1)))

    return p
```

NUM_CHROME即為上述公式的n, NUM_BIT = 3



```
NUM_CITY = 4                     # 城市個數 (new)

d = [ [  0, 12,  1,  8 ],
      [ 12,  0,  2,  3 ],
      [  1,  2,  0, 10 ],
      [  8,  3, 10,  0 ] ]  # 個城市之間的距離 (new)

# ====  參數設定(與演算法相關)  ====
NUM_ITERATION = 20               # 世代數(迴圈數)
NUM_CHROME = 20                  # 染色體個數
NUM_BIT = NUM_CITY - 1           # 染色體長度(從第0個城市出發，最終回到第0個城市，所以city 0不
```

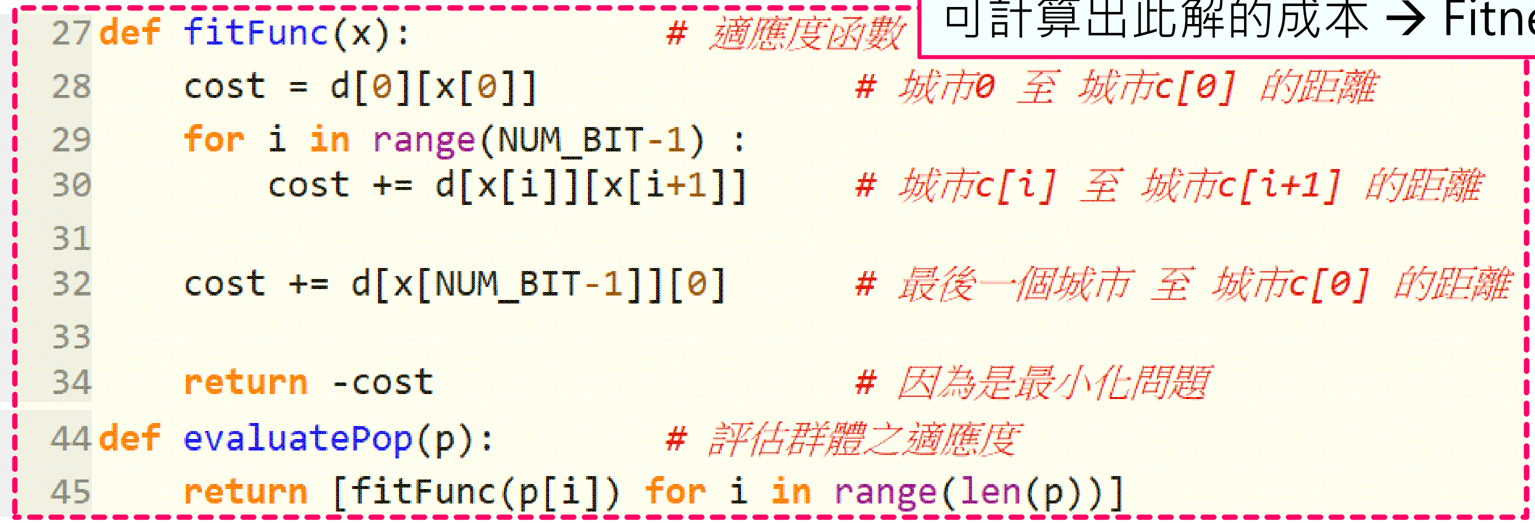




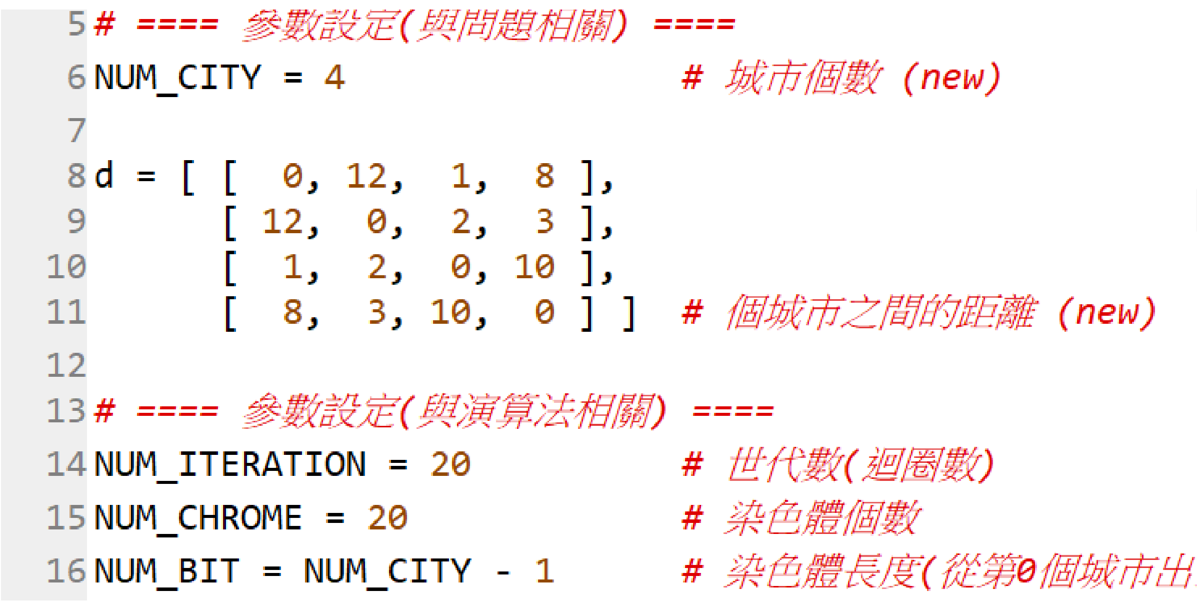
```
def initPop():              # 初始化群體 (new)
    p = []
    for i in range(NUM_CHROME) :
        p.append(np.random.permutation(range(1, NUM_BIT+1)))

    return p
```

1. 編碼：為{1,2,3}的一個排列，例如 (3, 1, 2)

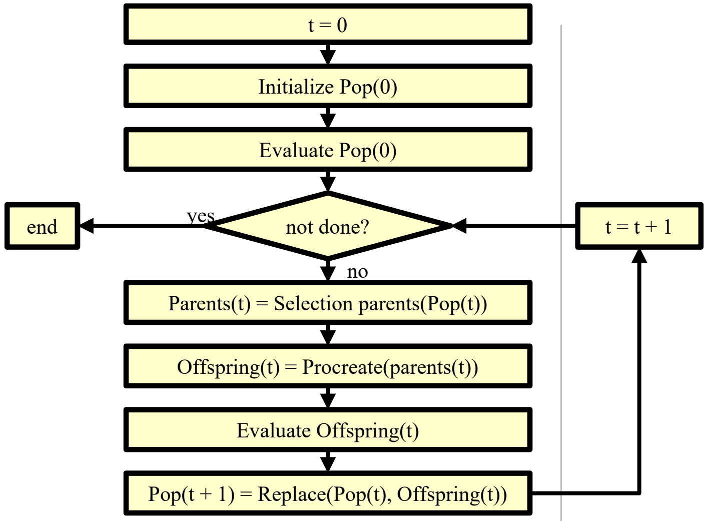
2. 適應度：根據編碼走訪的城市順序，可計算出此解的成本 → Fitness = -成本

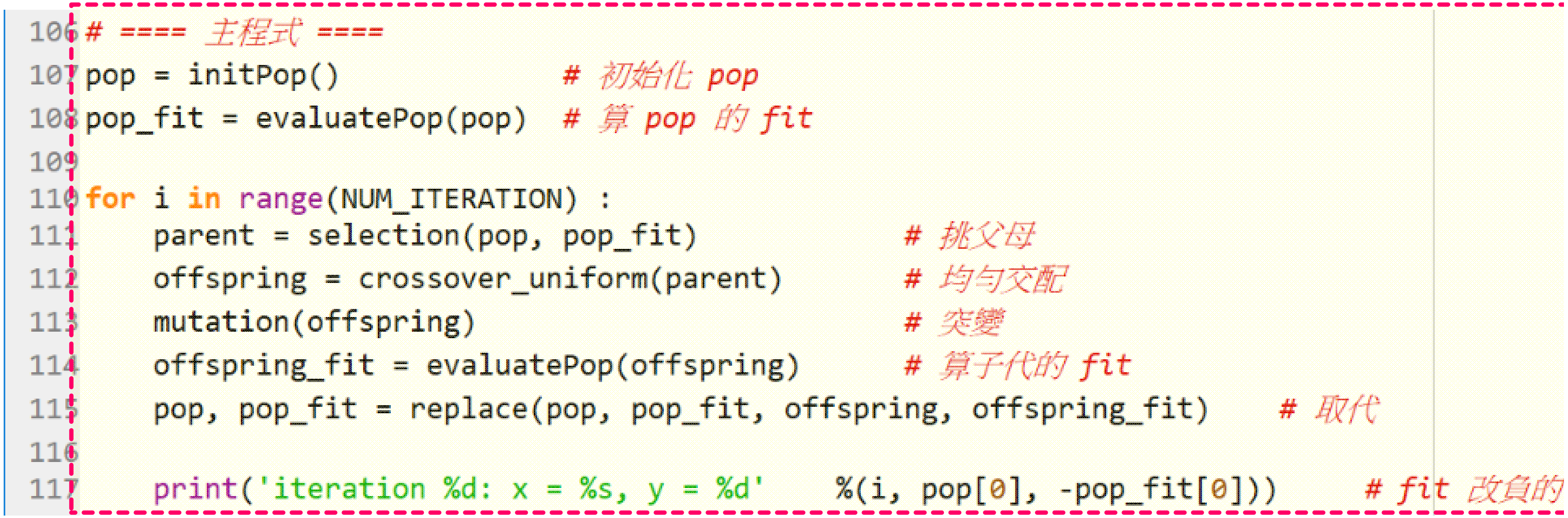


```
def fitFunc(x):             # 適應度函數
    cost = d[0][x[0]]                  # 城市0 至 城市c[0] 的距離
    for i in range(NUM_BIT-1) :
        cost += d[x[i]][x[i+1]]        # 城市c[i] 至 城市c[i+1] 的距離

    cost += d[x[NUM_BIT-1]][0]         # 最後一個城市 至 城市c[0] 的距離

    return -cost                       # 因為是最小化問題

def evaluatePop(p):         # 評估群體之適應度
    return [fitFunc(p[i]) for i in range(len(p))]
```

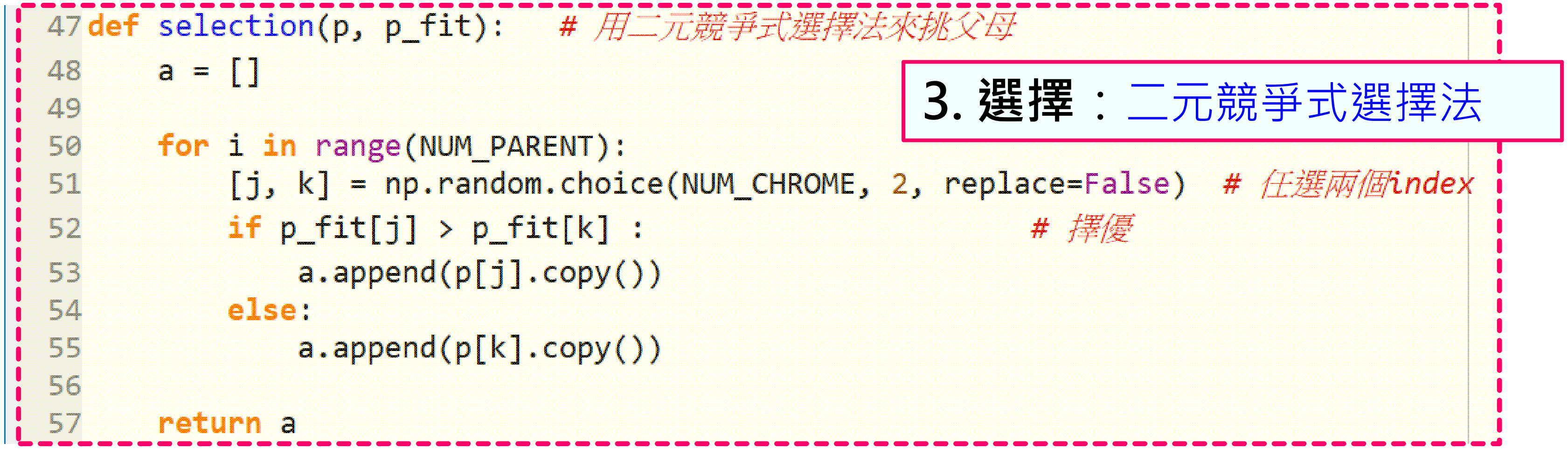


```
# ==== 參數設定(與問題相關) ====
NUM_CITY = 4                    # 城市個數 (new)

d = [ [  0, 12,  1,  8 ],
      [ 12,  0,  2,  3 ],
      [  1,  2,  0, 10 ],
      [  8,  3, 10,  0 ] ]  # 個城市之間的距離 (new)

# ==== 參數設定(與演算法相關) ====
NUM_ITERATION = 20              # 世代數(迴圈數)
NUM_CHROME = 20                 # 染色體個數
NUM_BIT = NUM_CITY - 1          # 染色體長度(從第0個城市出發，最終回到第0個城市，所以city 0不考
```


t = 0
Initialize Pop(0)
Evaluate Pop(0)
not done?
yes
end
no
Parents(t) = Selection parents(Pop(t))
Offspring(t) = Procreate(parents(t))
Evaluate Offspring(t)
Pop(t + 1) = Replace(Pop(t), Offspring(t))
t = t + 1

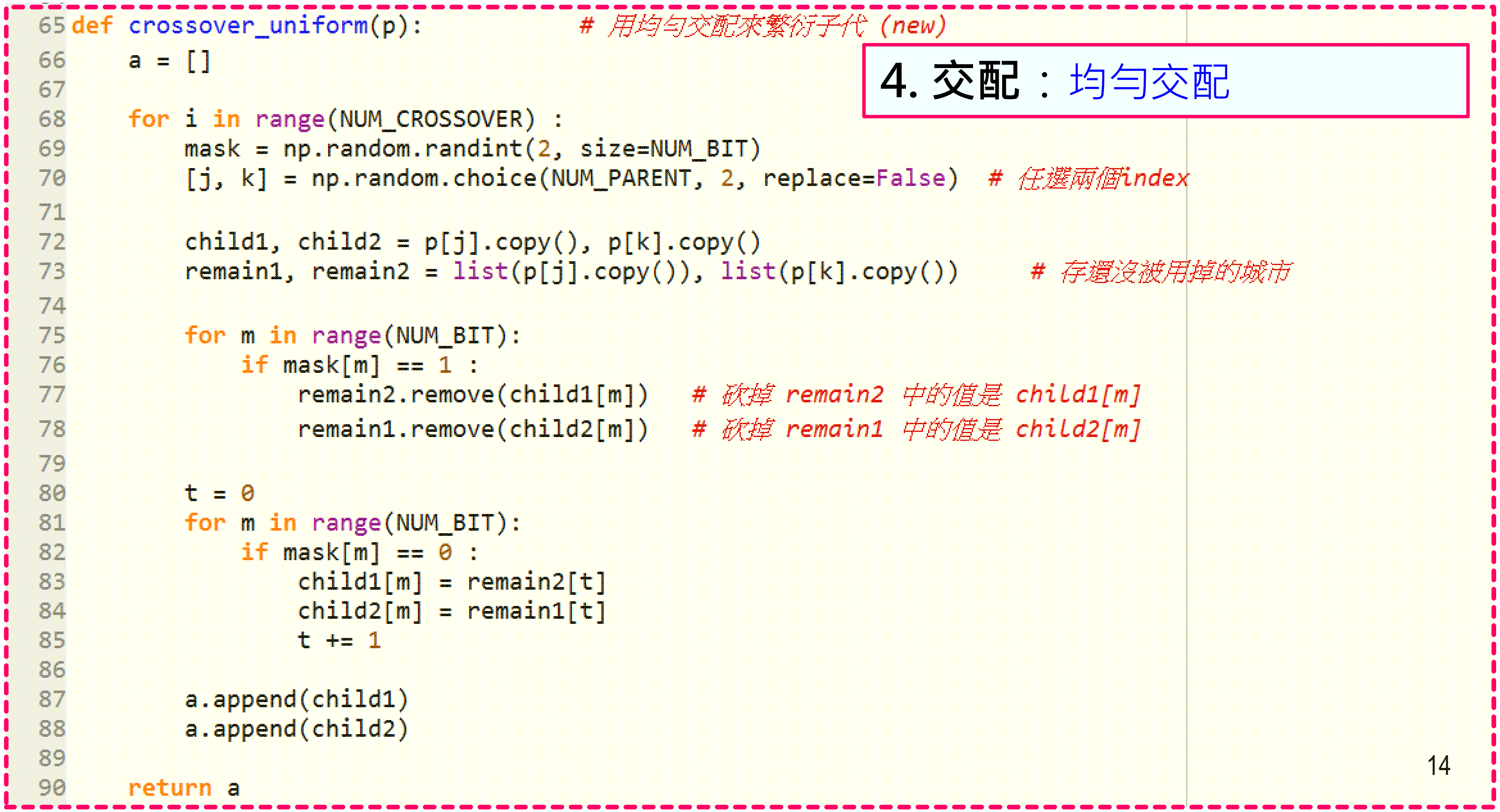


```
# ==== 主程式 ====
pop = initPop()             # 初始化 pop
pop_fit = evaluatePop(pop)  # 算 pop 的 fit

for i in range(NUM_ITERATION) :
    parent = selection(pop, pop_fit)            # 挑父母
    offspring = crossover_uniform(parent)       # 均勻交配
    mutation(offspring)                         # 突變
    offspring_fit = evaluatePop(offspring)      # 算子代的 fit
    pop, pop_fit = replace(pop, pop_fit, offspring, offspring_fit)    # 取代

    print('iteration %d: x = %s, y = %d'    %(i, pop[0], -pop_fit[0]))      # fit 改負的
```



3. 選擇：二元競爭式選擇法

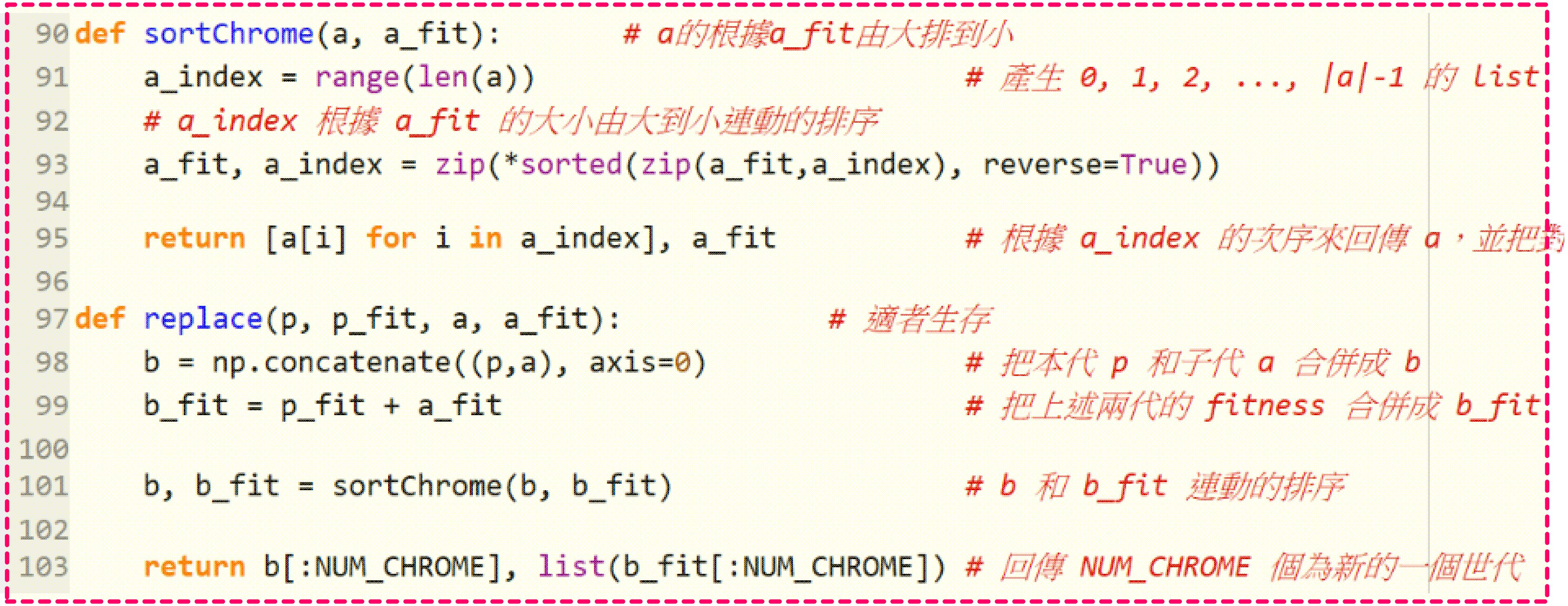
```python
def selection(p, p_fit):    # 用二元競爭式選擇法來挑父母
    a = []

    for i in range(NUM_PARENT):
        [j, k] = np.random.choice(NUM_CHROME, 2, replace=False)  # 任選兩個index
        if p_fit[j] > p_fit[k] :                              # 擇優
            a.append(p[j].copy())
        else:
            a.append(p[k].copy())

    return a
```



4. 交配：均匀交配

```python
def crossover_uniform(p):           # 用均匀交配來繁衍子代 (new)
    a = []

    for i in range(NUM_CROSSOVER) :
        mask = np.random.randint(2, size=NUM_BIT)
        [j, k] = np.random.choice(NUM_PARENT, 2, replace=False)  # 任選兩個index

        child1, child2 = p[j].copy(), p[k].copy()
        remain1, remain2 = list(p[j].copy()), list(p[k].copy())     # 存還沒被用掉的城市

        for m in range(NUM_BIT):
            if mask[m] == 1 :
                remain2.remove(child1[m])   # 砍掉 remain2 中的值是 child1[m]
                remain1.remove(child2[m])   # 砍掉 remain1 中的值是 child2[m]

        t = 0
        for m in range(NUM_BIT):
            if mask[m] == 0 :
                child1[m] = remain2[t]
                child2[m] = remain1[t]
                t += 1

        a.append(child1)
        a.append(child2)

    return a
```

6. 取代：Pop(t+1) = {Pop(t) - {worsts}} ∪ {kids}



```python
def sortChrome(a, a_fit):        # a的根據a_fit由大排到小
    a_index = range(len(a))                         # 產生 0, 1, 2, ..., |a|-1 的 list
    # a_index 根據 a_fit 的大小由大到小連動的排序
    a_fit, a_index = zip(*sorted(zip(a_fit,a_index), reverse=True))

    return [a[i] for i in a_index], a_fit           # 根據 a_index 的次序來回傳 a，並把

def replace(p, p_fit, a, a_fit):            # 適者生存
    b = np.concatenate((p,a), axis=0)               # 把本代 p 和子代 a 合併成 b
    b_fit = p_fit + a_fit                           # 把上述兩代的 fitness 合併成 b_fit

    b, b_fit = sortChrome(b, b_fit)                 # b 和 b_fit 連動的排序

    return b[:NUM_CHROME], list(b_fit[:NUM_CHROME]) # 回傳 NUM_CHROME 個為新的一個世代
```

與之前版本一樣