Exercise 4

Pseudo code of Exercise 3：

```
func = {"add", "sub", "mul", "div", "pow", "sqrt", "log", "exp", "max", "ifleq", "data", "diff", "avg"};
terminate = {"var", "const"};

def genetic():
    print("genetic");
    population = initialization(100, 6);
    p_operate = population[:];

    f = [fitness(dumps(i), 8, 500, "housing.txt") for i in population];
    sort_fitness(p_operate, f, 0.8);

    # select parents in population
    for g in range(1, 10):
        print("generation:", g);
        for i in range (0, 20):
            child = crossover(p_operate);
            while tree_depth(child) > 10 or child in population:
                child = crossover(p_operate);


            population.append(child);
            print(len(population));

        f = [];
        for i in population:
            f = f + [fitness(dumps(i), 8, 100, "housing.txt")];

        print("fitness", "ge", f);
        sort_fitness(population, f, 0.8);
        population = population[0:100];
        f = f[0:100];

    sort_fitness(population, f, 1);
    return population[0:20];

def initialization(size, maxdepth):
    ele = [full(maxdepth)];
```

```
    for i in range(1, size):
        individual = full(maxdepth);
        while initialization in ele:
            individual = full(maxdepth);
        ele = ele + [individual];
    return [full(maxdepth) for i in range (0, size)];

def full(maxdepth):
    if maxdepth == 0:
        leaf = random.choice(tuple(terminate));
        if leaf == "var":
            return [Symbol("data"), random.randint(0, 8)];
        else:
            return random.randint(1, 5);
    else:
        node = random.choice(tuple(func));
        if node == "add":
            return [Symbol("add"), full(maxdepth - 1), full(maxdepth - 1)];
        if node == "sub":
            return [Symbol("sub"), full(maxdepth - 1), full(maxdepth - 1)];
        if node == "mul":
            return [Symbol("mul"), full(maxdepth - 1), full(maxdepth - 1)];
        if node == "div":
            return [Symbol("div"), full(maxdepth - 1), full(maxdepth - 1)];
        if node == "pow":
            return [Symbol("pow"), full(maxdepth - 1), full(maxdepth - 1)];
        if node == "avg":
            return [Symbol("avg"), full(maxdepth - 1), full(maxdepth - 1)];
        if node == "max":
            return [Symbol("max"), full(maxdepth - 1), full(maxdepth - 1)];
        if node == "diff":
            return [Symbol("diff"), full(maxdepth - 1), full(maxdepth - 1)];
        if node == "sqrt":
            return [Symbol("sqrt"), full(maxdepth - 1)];
        if node == "log":
            return [Symbol("log"), full(maxdepth - 1)];
        if node == "exp":
            return [Symbol("exp"), full(maxdepth - 1)];
        if node == "data":
            return [Symbol("data"),    full(maxdepth - 1)];
        if node == "ifleq":
            return [Symbol("ifleq"), full(maxdepth - 1), full(maxdepth - 1), full(maxdepth - 
1), full(maxdepth - 1)];
```

```
def get_parents(population):
    p = 0.4 ;
    rand = random.uniform(0, 1);
    for i in range(0, len(population)):
        if rand <   p * math.pow((1 - p), i):
            return population[i];
    return population[len(population) - 1];

def crossover_point(tree, n, index):
    if len(tree) == 2 and isinstance(tree[1], int):
        index = index + [0];
        return tree, index;
    if len(tree) == 3 and (isinstance(tree[1], int) or isinstance(tree[2], int)):
        index = index + [0];
        return tree, index;
    if len(tree) == 5 and (
            isinstance(tree[1], int) or isinstance(tree[2], int) or isinstance(tree[3], int) or
isinstance(tree[4], int)):
        index = index + [0];
        return tree, index;
    else:
        if random.uniform(0, 1) < 1/n:
            index = index + [0];
            return tree, index;
        else:
            tree_len = len(tree);

            if tree_len == 2:
                index = index + [1];
                return crossover_point(tree[1], n, index);
            else:
                rand = random.uniform(0, 1);
                for i in range(1, tree_len - 1):
                    if rand < i * 1/tree_len:
                        index = index + [i];
                        return crossover_point(tree[i], n, index);
                index = index + [tree_len - 1];
                return crossover_point(tree[tree_len - 1], n, index);

def replace(tree, subtree, r):
    temp = tree;
    if len(r) == 1: # the case [0]
        return subtree;
    if len(r) == 2:
```

```
        temp[r[0]] = subtree;
        return tree;
    else:
        for i in r:
            temp = temp[i];
    return tree;

def crossover(population):
    population_dun = population[:];
    parent1 = get_parents(population_dun);
    population_dun.remove(parent1);
    parent2 = get_parents(population_dun);

    offspring = parent1[:];
    number_parent1 = 20;
    number_parent2 = 20;
    (stree1, route1) = crossover_point(parent1, number_parent1, []);
    (stree2, route2) = crossover_point(parent2, number_parent2, []);

    while abs(tree_depth(stree1) - tree_depth(stree2)) > 2 or stree2 == stree1:

        (stree1, route1) = crossover_point(parent1, number_parent1, []);
        (stree2, route2) = crossover_point(parent2, number_parent2, []);

    offspring = replace(offspring, stree2, route1);
    return offspring;

def tree_depth(tree):
    if not tree:
        return 0;
    if isinstance(tree, list):
        return 1 + max(tree_depth(item) for item in tree);
    else:
        return 0;


def fitness(expr, n, m, data):
    path = data;
    # read files here: replace later
    x, y = main.readdata(path, n);

    mse = [];
    for i in range(0, m):
        m = main.niso_lab3(1, n, " ".join(x[i]), expr);
```

```
            try:
                m = math.pow((float(y[i]) - float(m)), 2);
            except OverflowError:
                m = 0.001;
            mse = mse + [m];

        f = sum(mse)/m;
        return f;


def sort_fitness(population, fitness, p):
        if len(population) != len(fitness):
            print("invalid length");

        for w in range(0, len(fitness) - 2):
            for l in range(w, len(fitness) - 2):
                if fitness[w] > fitness[l] and random.uniform(0,1) < p:
                    temp = population[w];
                    population[w] = population[l];
                    population[l] = temp;

                    temp = fitness[w];
                    fitness[w] = fitness[l];
                    fitness[l] = temp;


def perform():
        gen_result = genetic();
        l_w = [];
        for i in gen_result:
            l_w = l_w + [fitness(dumps(i), 8, 100, "housing.txt")];
        sort_fitness(gen_result, l_w, 1);
        print(l_w[0:3]);
        for i in range(0, 3):
            print(gen_result);
```

Exercise 5

In this exercise, the main determine parameters are:

1. the size of population
2. the possibility in fitness sorting function: (When the fitness[a] > fitness[b], there is some possibility to exchange two elements;

3. the number of parents that enrolled in the crossover process

To select the optimum parameter (in limit time), for each parameter selection, I run for the same repetitions and selected the parameters which gave relatively better results and didn’t spend too much time.