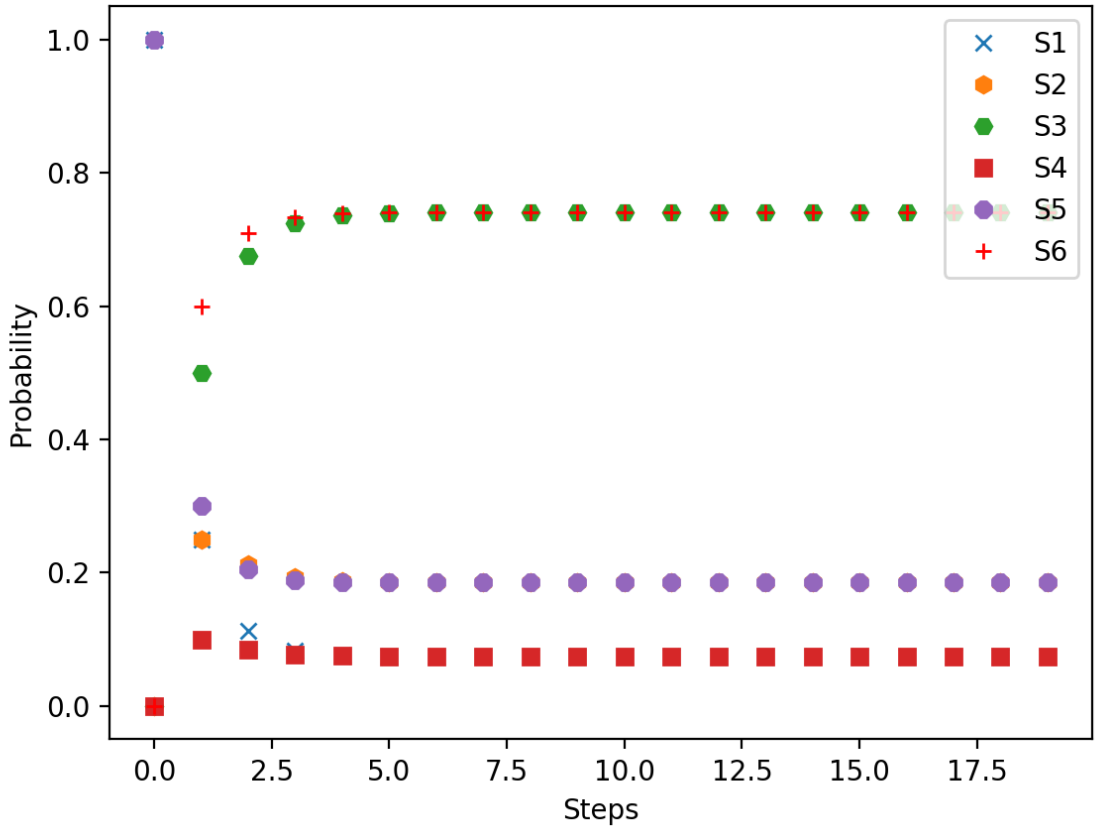
Question one

It is obviously a Markov Chain:

```
import numpy as np
from matplotlib import pyplot

P = np.matrix([
    [0.25,0.25,0.50],
    [0.10,0.30,0.60],
    [0.05,0.15,0.80]
              ])

v = np.matrix([
    [1,0,0],
    [0,1,0],
    [0,0,1]
              ])

# Get the data
plot_data = []
for step in range(20):
    result = v * P**step
    plot_data.append(np.array(result).flatten())

# Convert the data format
plot_data = np.array(plot_data)

# Create the plot
pyplot.figure(1)
pyplot.xlabel('Steps')
pyplot.ylabel('Probability')
lines = []
for i, shape in zip(range(6), ['x', 'h', 'H', 's', '8', 'r+']):
    line, = pyplot.plot(plot_data[:, i], shape, label="S%i" % (i+1))
    lines.append(line)
pyplot.legend(handles=lines, loc=1)
pyplot.show()
```

And we can get the plot:



Question two

> $r1 = 0.993 \times (1 - 0.005 \times 0.005) \times (1 - 0.002 \times (1 - 0.99 * 0.999)) \times 0.98$
> $\times (1 - (1 - 0.99 \times 0.99) \times (1 - 0.99 \times 0.99)) r1 = 0.993 \times 0.999975 \times 0.99997802 \times 0.98 \times 0.99960399 = 0.9727$
>
> $r2 = 0.998 \times (1 - 0.005 \times 0.005) \times (1 - 0.01 \times 0.001 \times 0.002) \times 0.98$
> $\times (1 - 0.00000001) r2 = 0.998 \times 0.999975 \times 0.99999998 \times 0.98 \times 0.99999999 = 0.978$

The reliability of the second design is higher than the reliability of the first design. Also, forlanding and rockets the second alternative has more parallel options, so it is less likely to fail.The assumption encoded in the design is that the mission is successful whenever power,communication, landing, storage and rockets work properly (at least one of the correspondingcomponents when more are connected in parallel, more complex relation for landing in firstdesign). The less reasonable assumption is that the design has no impact on the parameters ofthe actual mission: the minimum number of rockets required for propulsion probably has animpact on the actual force that can be generated.

Note that the second design has better power supply. Even if we replace it with one of thesame quality (reliability 0.993) the second design is still better, but the difference is reduced: $r2 = 0.9731$.

Question three

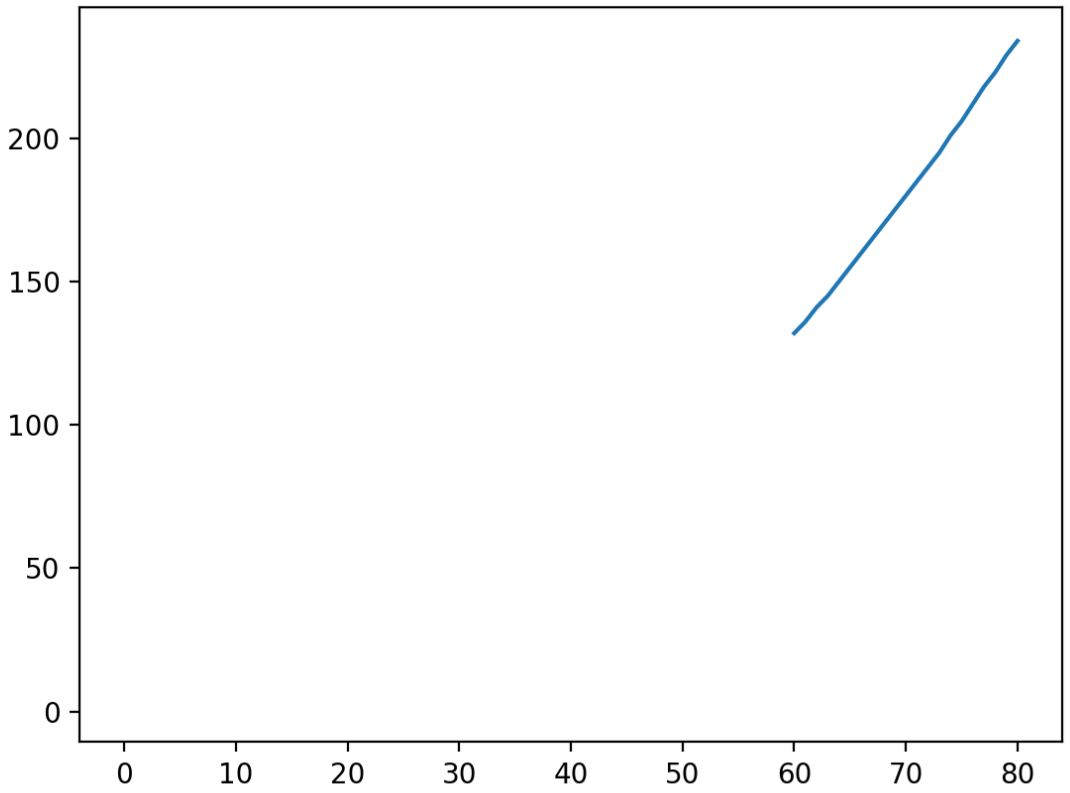
Consider the data in the table in order to predict weight as a function of height. Let x and y represent the height and weight respectively.

Height(in) x	**weight(lb)** y
60	132
61	136
62	141
63	145
64	150
65	155
66	160
67	165
68	170
69	175
70	180
71	185
72	190
73	195
74	201
75	206
76	212
77	218
78	223
79	229
80	234

The objective is to find the linear model for best fit $y = ax + b$

The objective is to find the values of aa and b uses the formulas

```
import matplotlib.pyplot as plt
import numpy as np
X = [60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80]
Y = [132,136,141,145,150,155,160,165,170,175,180,185,190,195,201,206,212,218,223,229,234]
z1 = np.polyfit(X, Y, 1)  #一次多项式拟合，相当于线性拟合
p1 = np.poly1d(z1)
print(z1)
print(p1)
plt.plot(X,Y,1)
plt.show()
```



Question Four

We can get these information from the question:

$$Maximize: f(x_1, x_2) = 2000x_1 + 3000x_2$$

$$S.t. \begin{cases} 50x_1 + 50x_2 \leq 2400 \\ 60x_1 + 40x_2 \leq 2500 \end{cases}$$

we can turn it into the standard form:

$$Minimize: f(x_1, x_2) = -2000x_1 - 3000x_2$$

$$S.t. \begin{cases} 50x_1 + 50x_2 + x_3 = 2400 \\ 60x_1 + 40x_2 + x_4 = 2500 \end{cases}$$

Then we can use the Simplex Method , This is the python code:

```
import numpy as np

class Simplex(object):
    def __init__(self, obj, max_mode=False):  # default is solve min LP, if want to solve max
lp,should * -1
        self.mat, self.max_mode = np.array([[0] + obj]) * (-1 if max_mode else 1), max_mode

    def add_constraint(self, a, b):
        self.mat = np.vstack([self.mat, [b] + a])

    def _simplex(self, mat, B, m, n):
        while mat[0, 1:].min() < 0:
            col = np.where(mat[0, 1:] < 0)[0][0] + 1  # use Bland's method to avoid degeneracy.
use mat[0].argmin() ok?
            row = np.array([mat[i][0] / mat[i][col] if mat[i][col] > 0 else 0x7fffffff for i in
                            range(1, mat.shape[0])]).argmin() + 1  # find the theta index
            if mat[row][col] <= 0: return None  # the theta is ∞, the problem is unbounded
            self._pivot(mat, B, row, col)
        return mat[0][0] * (1 if self.max_mode else -1), {B[i]: mat[i, 0] for i in range(1, m)
if B[i] < n}

    def _pivot(self, mat, B, row, col):
        mat[row] /= mat[row][col]
        ids = np.arange(mat.shape[0]) != row
        mat[ids] -= mat[row] * mat[ids, col:col + 1]  # for each i!= row do: mat[i]= mat[i] -
mat[row] * mat[i][col]
        B[row] = col

    def solve(self):
        m, n = self.mat.shape  # m - 1 is the number slack variables we should add
        temp, B = np.vstack([np.zeros((1, m - 1)), np.eye(m - 1)]), list(range(n - 1, n + m -
1))  # add diagonal array
        mat = self.mat = np.hstack([self.mat, temp])  # combine them!
        if mat[1:, 0].min() < 0:  # is the initial basic solution feasible?
            row = mat[1:, 0].argmin() + 1  # find the index of min b
            temp, mat[0] = np.copy(mat[0]), 0  # set first row value to zero, and store the
previous value
            mat = np.hstack([mat, np.array([1] + [-1] * (m - 1)).reshape((-1, 1))])
            self._pivot(mat, B, row, mat.shape[1] - 1)
            if self._simplex(mat, B, m, n)[0] != 0: return None  # the problem has no answer

            if mat.shape[1] - 1 in B:  # if the x0 in B, we should pivot it.
                self._pivot(mat, B, B.index(mat.shape[1] - 1), np.where(mat[0, 1:] != 0)[0][0]
+ 1)
            self.mat = np.vstack([temp, mat[1:, :-1]])  # recover the first line
            for i, x in enumerate(B[1:]):
                self.mat[0] -= self.mat[0, x] * self.mat[i + 1]
        return self._simplex(self.mat, B, m, n)


# I defined a function
# for this question:
```

```
>>> t = Simplex([-2000,-3000])
>>> t.add_constraint([5, 5], 240)
>>> t.add_constraint([6,4], 250)
>>> print(t.solve())
(-144000.0, {2: 48.0})
```

The maximum is 144000