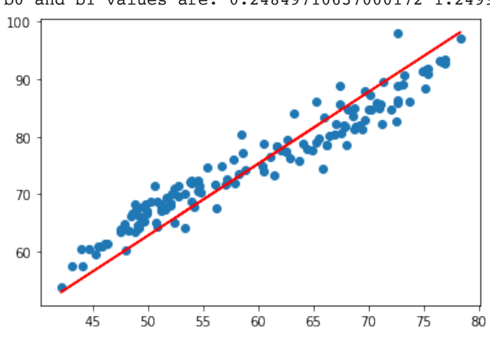
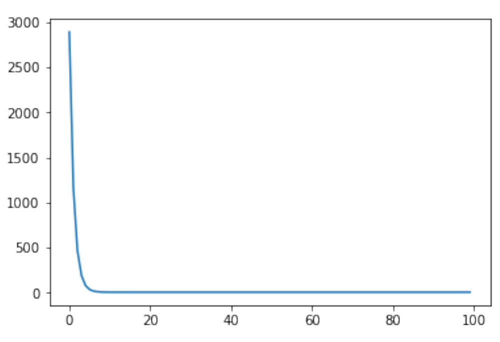
2(a)

In [1]:
```python
import numpy as np
import sympy as sym
import matplotlib.pyplot as plt
npyfile = np.load('data-2.npy')
x = npyfile[:,0].reshape(140,1)
y = npyfile[:,1].reshape(140,1)
print(len(x),len(y))
iterations = 5000
plt.scatter(x,y)
b0 = 0
b1 = 0
d_b0 = 0
d_b1 = 0
X = npyfile[:,0].reshape(140,1)
# X = np.concatenate([np.ones((X.shape[0], 1)), X], axis=1)
m = len(y)
alpha = 0.0001
def cost_function(b0,b1,X,y,iterations=5000):
    J = (1/(2*m))*sum((-b0-(b1*X)+y)**2)
    return J

def gradient(b0,b1,X,y):
    d_b0 = -(1/m)*sum(y-b0-(b1*X))
    d_b1 = -(1/m)*sum(X*(y-b0-(b1*X)))

    return d_b0, d_b1

def gradient_descent(b0,b1,X,y,alpha,iterations=5000):
    cost = []
    for i in range(iterations):
        [d_b0, d_b1] = gradient(b0,b1,X,y)
        cost.append(cost_function(b0,b1,X,y)[0])
        b0 = b0-(alpha*d_b0)
        b1 = b1-(alpha*d_b1)
    return b0[0],b1[0],cost
[b0, b1,cost] = gradient_descent(b0,b1,X,y,alpha)
print('b0 and b1 values are:',b0,b1)


Y = b0+b1*X
plt.plot(X,Y,'r')
plt.show()
# print(cost)
f = np.linspace(0,iterations,5000)
plt.plot(f[:100],cost[:100])
```

```
140 140
b0 and b1 values are: 0.24849710637000172 1.249979368235588
```



Out[1]: `[<matplotlib.lines.Line2D at 0x7fd7c8ae6280>]`

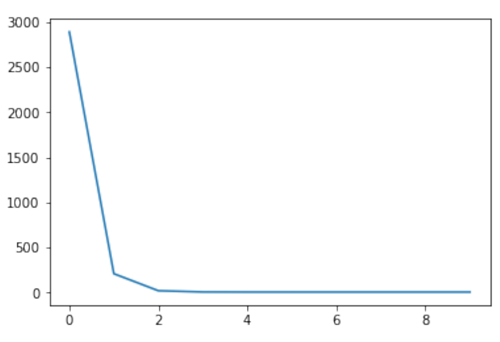


2(b)

In [2]:
```python
alpha = 0.0002
b0 = 0
b1 = 0
iterations = 5000
f = np.linspace(0,iterations,5000)
[b0, b1,cost] = gradient_descent(b0,b1,X,y,alpha)
print('b0 and b1 values are:',b0,b1)

plt.plot(f[:10],cost[:10])
```

```
b0 and b1 values are: 0.4736728136091704 1.246310709297146
```

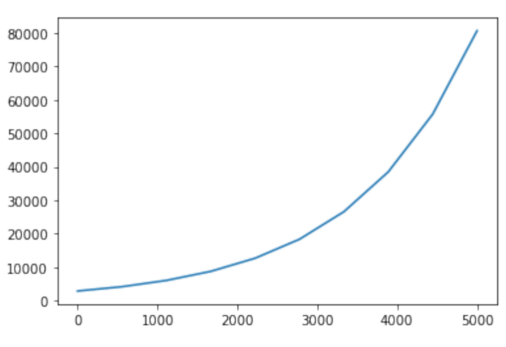
Out[2]: `[<matplotlib.lines.Line2D at 0x7fd7f8a452e0>]`



In [3]:
```python
alpha = 0.0006
b0 = 0
b1 = 0
iterations = 5000
f = np.linspace(0,iterations,10)
[b0, b1,cost] = gradient_descent(b0,b1,X,y,alpha)
print('b0 and b1 values are:',b0,b1)

plt.plot(f,cost[:10])
```

```
<ipython-input-1-ea25a796d343>:19: RuntimeWarning: overflow encountered in add
  J = (1/(2*m))*sum((-b0-(b1*X)+y)**2)
<ipython-input-1-ea25a796d343>:19: RuntimeWarning: overflow encountered in square
  J = (1/(2*m))*sum((-b0-(b1*X)+y)**2)
<ipython-input-1-ea25a796d343>:24: RuntimeWarning: overflow encountered in add
  d_b1 = -(1/m)*sum(X*(y-b0-(b1*X)))
<ipython-input-1-ea25a796d343>:34: RuntimeWarning: invalid value encountered in subtract
  b1 = b1-(alpha*d_b1)
b0 and b1 values are: nan nan
```

Out[3]: `[<matplotlib.lines.Line2D at 0x7fd7a023b280>]`



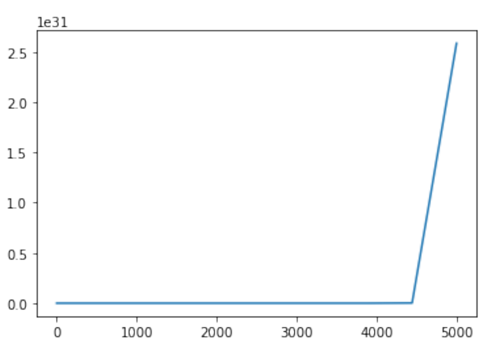
alpha = 0.01 b0 = 0 b1 = 0 iterations = 5000 f = np.linspace(0,iterations,10) [b0, b1,cost] = gradient_descent(b0,b1,X,y,alpha) print('b0 and b1 values are:',b0,b1)

plt.plot(f,cost[:10])

In [4]:
```python
alpha = 0.01
b0 = 0
b1 = 0
iterations = 5000
f = np.linspace(0,iterations,10)
[b0, b1,cost] = gradient_descent(b0,b1,X,y,alpha)
print('b0 and b1 values are:',b0,b1)

plt.plot(f,cost[:10])
```

```
<ipython-input-1-ea25a796d343>:19: RuntimeWarning: overflow encountered in square
  J = (1/(2*m))*sum((-b0-(b1*X)+y)**2)
<ipython-input-1-ea25a796d343>:19: RuntimeWarning: overflow encountered in add
  J = (1/(2*m))*sum((-b0-(b1*X)+y)**2)
<ipython-input-1-ea25a796d343>:24: RuntimeWarning: overflow encountered in add
  d_b1 = -(1/m)*sum(X*(y-b0-(b1*X)))
<ipython-input-1-ea25a796d343>:34: RuntimeWarning: invalid value encountered in subtract
  b1 = b1-(alpha*d_b1)
b0 and b1 values are: nan nan
```

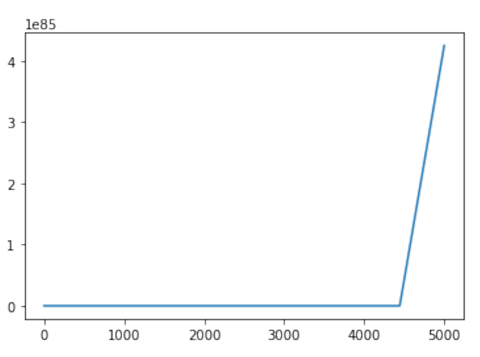
Out[4]: `[<matplotlib.lines.Line2D at 0x7fd7f8a49be0>]`



In [5]:
```python
alpha = 10
b0 = 0
b1 = 0
iterations = 5000
f = np.linspace(0,iterations,10)
[b0, b1,cost] = gradient_descent(b0,b1,X,y,alpha)
print('b0 and b1 values are:',b0,b1)

plt.plot(f,cost[:10])
```

```
<ipython-input-1-ea25a796d343>:19: RuntimeWarning: overflow encountered in square
  J = (1/(2*m))*sum((-b0-(b1*X)+y)**2)
<ipython-input-1-ea25a796d343>:23: RuntimeWarning: overflow encountered in add
  d_b0 = -(1/m)*sum(y-b0-(b1*X))
<ipython-input-1-ea25a796d343>:24: RuntimeWarning: overflow encountered in multiply
  d_b1 = -(1/m)*sum(X*(y-b0-(b1*X)))
<ipython-input-1-ea25a796d343>:33: RuntimeWarning: invalid value encountered in subtract
  b0 = b0-(alpha*d_b0)
<ipython-input-1-ea25a796d343>:34: RuntimeWarning: invalid value encountered in subtract
  b1 = b1-(alpha*d_b1)
b0 and b1 values are: nan nan
```

Out[5]: `[<matplotlib.lines.Line2D at 0x7fd7d8a4c490>]`



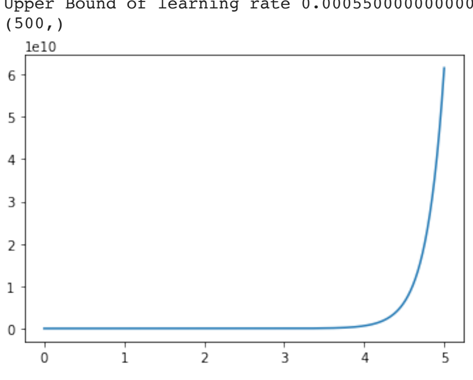
2(c)

The best learning rate is 0.0001 since it converges and converges quicker than the other learning rates

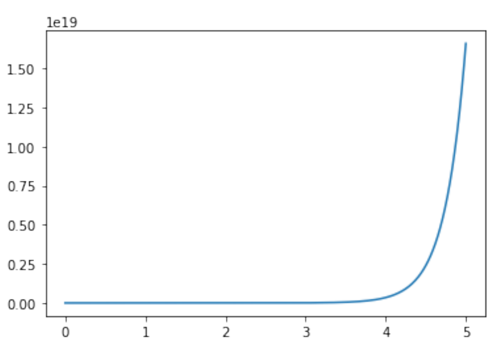
2(e)

In [6]:
```python
b0 = 0
b1 = 0
cost_1 = []
iterations = 5000
f = np.linspace(0,5,500)
print(np.shape(f))
alpha = 0.000001
for i in range(iterations):

    alpha+=0.000001
    [b0, b1,cost] = gradient_descent(b0,b1,X,y,alpha,iterations = 500)
    cost_1 = cost
    if cost[-1]>=6000:
        break
alpha = alpha-0.000001
print('Upper Bound of learning rate',alpha)
print(np.shape(cost))
plt.plot(f,cost)
plt.show()
[b0, b1,cost] = gradient_descent(b0,b1,X,y,alpha,iterations = 500)
plt.plot(f,cost)
```

```
(500,)
Upper Bound of learning rate 0.0005500000000000069
(500,)
```



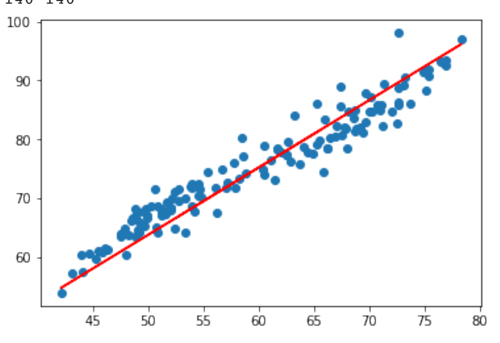
Out[6]: `[<matplotlib.lines.Line2D at 0x7fd799573280>]`



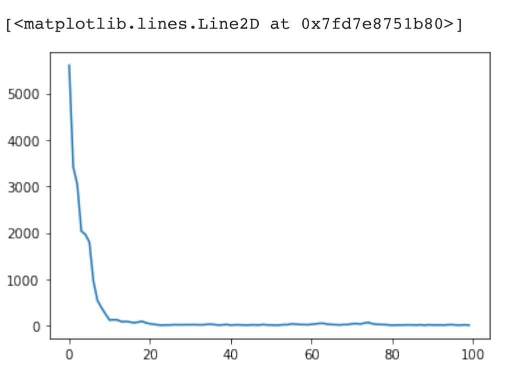
2(f)

In [7]:
```python
npyfile = np.load('data-2.npy')
x = npyfile[:,0].reshape(140,1)
y = npyfile[:,1].reshape(140,1)

a = np.arange(len(X))
np.random.shuffle(a)
np.random.seed(20)
print(len(x),len(y))
iterations = 5000
plt.scatter(x,y)
b0 = 0
b1 = 0
d_b0 = 0
d_b1 = 0
X = np.array(npyfile[:,0].reshape(140,1))
# X = np.concatenate([np.ones((X.shape[0], 1)), X], axis=1)
m = 20
alpha = 0.0005
def cost_function(b0,b1,X,y,iterations=5000):
    J = (1/(m))*sum((-b0-(b1*X)+y)**2)
#     print(J)
    return J

def gradient(b0,b1,X,y):
    d_b0 = -(1/m)*sum(y-b0-(b1*X))
    d_b1 = -(1/m)*sum(X*(y-b0-(b1*X)))
#     breakpoint()

    return d_b0, d_b1

def gradient_descent(b0,b1,X,y,alpha,iterations=5000):
    cost = []
    for i in range(iterations):
        np.random.shuffle(a)
        for j in range(0,140,20):
#             print(a)
            indices = a[j:j+m]
#             print(indices)
#             print(x_1)
            [d_b0, d_b1] = gradient(b0,b1,X[indices],y[indices])
    #         print(np.shape(x_1))
            cost.append(cost_function(b0,b1,X[indices],y[indices]))
#             print(cost)
            b0 = b0-(alpha*d_b0)
            b1 = b1-(alpha*d_b1)
    #         breakpoint()

    return b0,b1,cost
[b0, b1,cost] = gradient_descent(b0,b1,X,y,alpha)
# print('b0 and b1 values are:',b0,b1)


Y = b0+b1*X
plt.plot(X,Y,'r')
plt.show()
# print(cost)
f = np.linspace(0,iterations,5000)
plt.plot(f[:100],cost[:100])
```

```
140 140
```



Out[7]: `[<matplotlib.lines.Line2D at 0x7fd7e8751b80>]`



In [ ]:

2. (d) $\dfrac{\partial J(b)}{\partial b_0} \Rightarrow -\dfrac{1}{m} \sum_{i=1}^{m} (y^{(i)} - b_0 - b_1 x^{(i)}) = 0$

$$\sum_{i=1}^{m} (y^{(i)} - b_0 - b_1 x^{(i)}) = 0$$

$$\left(\sum_{i=1}^{m} y^{(i)} - b_0 - \sum_{i=1}^{m} b_1 x^{(i)}\right)$$

$$m b_0 = \sum_{i=1}^{m} y^{(i)} - \sum_{i=1}^{m} b_1 x^{(i)}$$

$$\boxed{b_0 = \frac{\left[\sum_{i=1}^{m} y^{(i)} - \sum_{i=1}^{m} b_1 x^{(i)}\right]}{m}}$$

$$\frac{\partial J(b)}{\partial b_1} = -\frac{1}{m} \sum_{i=1}^{m} \left[(y^{(i)} - b_0 - b_1 x^{(i)}) x^{(i)}\right] = 0$$

$$\Rightarrow \sum_{i=1}^{m} x^{(i)} y^{(i)} - \sum_{i=1}^{m} x^{(i)} b_0 - \sum_{i=1}^{m} b_1 x^{2i} = 0$$

$$\Rightarrow \sum_{i=1}^{m} b_1 x^{2i} = \sum_{i=1}^{m} x^{(i)} y^{(i)} - \sum_{i=1}^{m} x^{(i)} b_0$$

$$\Rightarrow \boxed{b_1 = \frac{\sum_{i=1}^{m} x^{(i)} (y^{(i)} - b_0)}{\sum_{i=1}^{m} (x^{(i)})^2}}$$

Gradient descent is an ~~iterative~~ optimization algorithm used to minimize a function by iteratively moving in the direction of the steepest descent, which is found by taking the negative of the gradient. It is used in machine learning to update parameters of a model, e.g. → coefficients in linear regression and weights in neural networks.

The 3 types of gradient descent are:

1) Batch: The weights of the network are updated after the entire dataset (batch) is processed.

2) Stochastic: The weights of the network are updated for each sample at every step and the weights change very quickly

3) Mini-batch: Tradeoff between batch and stochastic gradient descent. Uses a subset of the dataset to update weight at each step.