1. Calculate option price using Binomial Tree

Consider an asset whose price is denoted S_i for $i = \{0, \dots, N\}$. Suppose that S_i can take 2 values and that $u > d$:

$$S_i = \begin{cases} uS_{i-1} \\ dS_{i-1} \end{cases} \tag{1}$$

We can deduce p^u, p^d the risk-neutral probability that asset value moves up and down respectively at each period using the following formula where $r \geq 0$ is the annual risk-free rate and T is the time until maturity date (in year(s)):

$$\begin{cases} p^u = \dfrac{R-d}{u-d} \\ p^u = \dfrac{u-R}{u-d} \end{cases} \text{ where } R = 1 + r\frac{T}{N} \tag{2}$$

Suppose the following parameters:

$$S_0 = 100, K = 98, T = 1, N = 4, r = 0.04, d = 0.9525, u = 1.0425$$

We can calculate S_i^j, the asset price at period i given it moved up $j = \{0, \dots, i\}$ times, at each node and the risk-neutral probability as following.

$$S_i^j = S_0 u^j d^{i-j} \tag{3}$$

In [51]:

```
import numpy as np
import matplotlib.pyplot as plt

T = 1
N = 4
r = 0.04
d = 0.9525
u = 1.0425
K = 98
S0 = 100

def proba(T,N,d,u,r):
    R = 1+r*T/N
    p_u = (R-d)/(u-d)
    p_d = (u-R)/(u-d)
    return R,p_u, p_d

# Asset price evolution
print("Asset price evolution\n")

def price_evolution(S0,N,u,d):
    S = np.zeros([N+1,N+1])
    S[0,0] = S0
    for i in range(N+1):
        for j in range(i+1):
            S[i,j] = S0*u**j*d**(i-j)
    return S

S = price_evolution(S0,N,u,d)

for i in range(N+1):
    print("i=",i,", Si=", S[i,:i+1])
# Risk-neutral probability
print("\nRisk-neutral probability\n")

R, p_u, p_d = proba(T,N,d,u,r)
print("Probability of moving up:", p_u)
print("Probability of moving down:", p_d)
```

```
Asset price evolution

i= 0 , Si= [100.]
i= 1 , Si= [ 95.25 104.25]
i= 2 , Si= [ 90.725625  99.298125 108.680625]
i= 3 , Si= [ 86.41615781  94.58146406 103.51829531 113.29955156]
i= 4 , Si= [ 82.31139032  90.08884452  98.60117629 107.91782286 118.114782
5 ]

Risk-neutral probability

Probability of moving up: 0.6388888888888891
Probability of moving down: 0.36111111111111094
```

Consider an European Put option on this underlying whose current price is denoted V_0. We want to calculate V_0 using the Binomial Tree method. The pay-off of an EU Put is defined as:

$$\text{Pay-off} = max(K - S_T, 0) \text{ where } K \text{ is the strike price} \tag{4}$$

Thus, V_i^j which is the option price at period i and node j can be calculated as

$$V_i^j = \frac{1}{R}\left(p_u V_{i+1}^{j+1} + p_d V_{i+1}^j\right) \tag{5}$$

In [52]:

```
def pay_off(S,K):
    return max(K-S, 0)

def EUPut(S0,K,T,N,d,u,r):
    S = price_evolution(S0,N,u,d)
    R, p_u, p_d = proba(T,N,d,u,r)
    V = np.zeros([N+1,N+1])

    for j in range(N+1):
        V[N,j] = pay_off(S[N,j],K)

    for i in range(N-1,-1,-1):
        for j in range(i+1):
            V[i,j] = (p_u*V[i+1,j+1] + p_d*V[i+1,j])/R
    return V

V = EUPut(S0,K,T,N,d,u,r)
for i in range(N,-1,-1):
    print("i=",i,", Vi=", V[i,:i+1])
# Price of EU Put:
print("\nPrice of EU Put: ", V[0,0])
```

```
i= 4 , Vi= [15.68860968  7.91115548  0.          0.          0.        ]
i= 3 , Vi= [10.61354516  2.82852094  0.          0.        ]
i= 2 , Vi= [5.58394028 1.01129736 0.        ]
i= 1 , Vi= [2.63616785 0.36157497]
i= 0 , Vi= [1.1712433]

Price of EU Put:  1.1712432962532606
```

In case of an American Put, we have the right to exercise the option during its lifetime so V_i^j is the greater between the value of an EU Put and the direct pay-off at (i, j).

$$V_i^j = max\left(\frac{1}{R}\left(p_u V_{i+1}^{j+1} + p_d V_{i+1}^j\right), K - S_i^j\right) \tag{6}$$

We can see that the price of an American Put is a pricier than that of an European Put.

In [53]:

```
def AmPut(S0,K,T,N,d,u,r):
    S = price_evolution(S0,N,u,d)
    R, p_u, p_d = proba(T,N,d,u,r)
    V = np.zeros([N+1,N+1])

    for j in range(N+1):
        V[N,j] = pay_off(S[N,j],K)

    for i in range(N-1,-1,-1):
        for j in range(i+1):
            V[i,j] = max((p_u*V[i+1,j+1] + p_d*V[i+1,j])/R, pay_off(S[i,j], K))
    return V

V = AmPut(S0,K,T,N,d,u,r)
for i in range(N,-1,-1):
    print("i=", i, V[i,:i+1])
# Price of Am Put:
print("\nPrice of American Put: ", V[0,0])
```

```
i= 4 [15.68860968  7.91115548  0.          0.          0.        ]
i= 3 [11.58384219  3.41853594  0.          0.        ]
i= 2 [7.274375   1.22224882 0.        ]
i= 1 [3.37399884 0.43699765]
i= 0 [1.48275388]

Price of American Put:  1.4827538776745017
```

It is possible to compare the accuracy of Binomial Tree model with Black-Scholes model given that [1]:

$$u = e^{\sigma\sqrt{T/N}} \qquad (7)$$
$$d = e^{-\sigma\sqrt{T/N}}$$

Assume the following parameters:

$$S_0 = 100, K = 98, T = 1, r = 0.04, \sigma = 0.2$$

From Black-Scholes model, we know that the price of an European Put is

$$V_0 = Ke^{-rT}N(-d2) - SN(-d1) \qquad (8)$$

We can test the accuracy with different discretising number of periods N from N={5,10,...,500}.

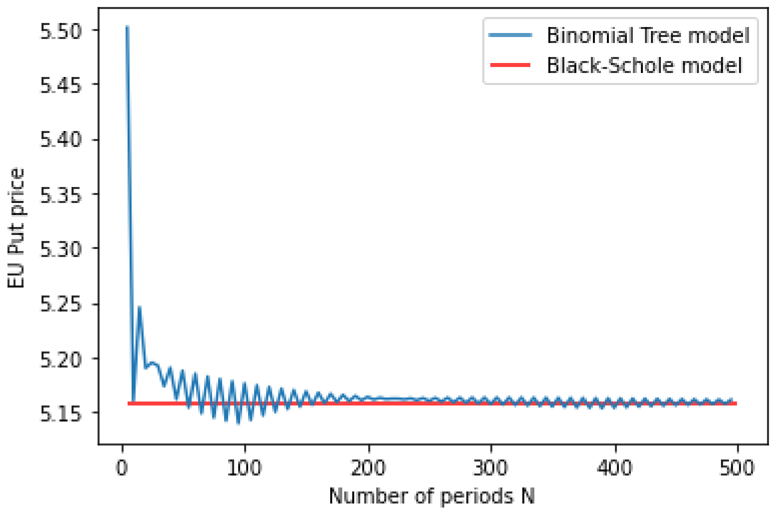
In [54]:

```
T = 1
r = 0.04
sigma = 0.2
K = 98
S0 = 100

def EUPut_BS(S,K,r,sigma,T):
    from scipy.stats import norm
    d1 = (np.log(S/K)+(r+sigma**2/2)*T)/(sigma*T**.5)
    d2 = d1 - sigma*T**.5
    return K*np.exp(-r*T)*norm.cdf(-d2) - S*norm.cdf(-d1)

print("\nPrice of European Put (Blach-Schole model): ", EUPut_BS(S0,K,r,sigma,T))

EUPut_Binomial = []
for N in range(5,500,5):
    d = np.exp(-sigma*(T/N)**.5)
    u = np.exp(sigma*(T/N)**.5)
    EUPut_Binomial.append(EUPut(S0,K,T,N,d,u,r)[0,0])

plt.plot(range(5,500,5), EUPut_Binomial)
plt.hlines(EUPut_BS(S0,K,r,sigma,T),5,500, colors='red')
plt.legend(['Binomial Tree model', 'Black-Schole model'])
plt.xlabel('Number of periods N')
plt.ylabel('EU Put price')
```

```
Price of European Put (Blach-Schole model):  5.157975067429362
```

Out[54]:

```
Text(0, 0.5, 'EU Put price')
```



We can see the that the Binomial Tree model converges to Black-Schole model as the number of periods N increases.

2. VaR and CVaR calculation using optimisation approach

Consider a Markowiz portfolio x of n assets such that $\sum_{i=1}^{n} x_i = 1, \sum_{i=1}^{n} \mu_i x_i \geq R, x_i \geq 0$ for $i = \{1, \ldots, n\}$ whose loss function is defined as

$$f(x, y_s) = (b - y_s)^T x \tag{7}$$

where y_s is the vector of asset price at scenario $s = \{1, \ldots, S\}$ and b is the current asset price vector.

In this section, I assume the Black-Schole models on asset price so that the price of asset i at T is given by:

$$S_T^i = S_0^i exp\left(\left(\mu_i - \frac{\sigma_i^2}{2}\right)T + \sigma_i \sqrt{T} W\right) \text{ where } W \sim \mathcal{N}(0, 1) \tag{8}$$

The following parameters are used to generate $S = 2000$ scenarios

$$\mu = (0.11, 0.12, 0.15), \sigma = (0.1, 0.15, 0.2)$$

First, we can calculate VaRα as the α-quantile of the loss function and CVaR as the average of all the loss that exceeds VaR. Suppose a portfolio of equal weights, VaR and CVaR are 0.2366 and 0.518 respectively with expected return roughly 12.67%.

In [28]:

```
mu = [0.11, 0.12, 0.15]
vol = [0.1, 0.15, 0.2]

b = [10,10,10]
n = 3
S = 2000
alpha = 0.95
y = np.zeros([S,n])
u = np.zeros([S,n])
for s in range(S):
    for i in range(n):
        y[s,i] = b[i]*np.exp((mu[i]-vol[i]**2/2)*T + vol[i]*T**0.5*np.random.normal())
    u[s,:] = b-y[s,:]

x = 1/n*np.ones(n)

loss = np.dot(u, x)
VaR = np.quantile(loss, alpha)
CVaR = np.mean(loss[loss>VaR])
ret = np.dot(mu,x)

print("VaR: ", VaR)
print("CVaR: ", CVaR)
print("Portfolio return: ", ret)
```

```
VaR:  0.19280848889176705
CVaR:  0.5309573718564671
Portfolio return:  0.12666666666666665
```

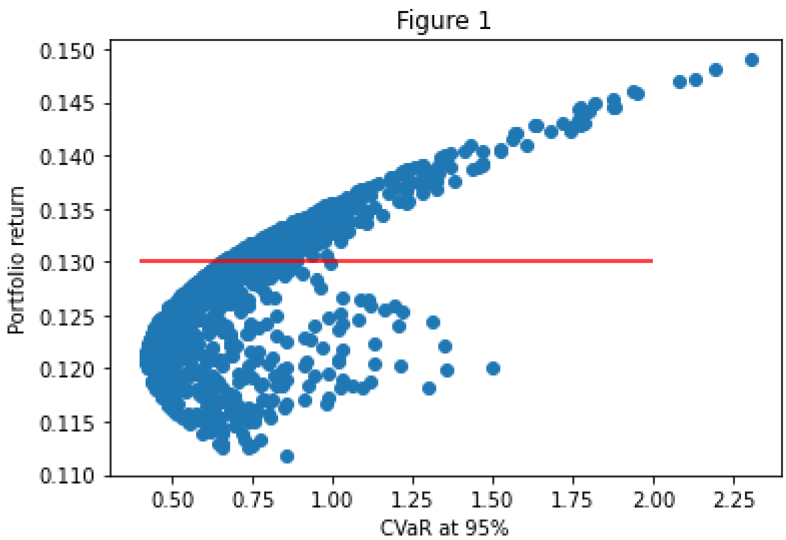
Using different portfolio structure x, we can deduce different values for VaR and CVaR and Figure 1 shows the relationship between $\text{CVaR}_{0.95}$ and portfolio return. At $R = 13\%$ we can see that optimal $\text{CVaR}_{0.95}$ should be somewhere around 0.65-0.7 which is represented in Figure 1 as the intersection between the red line and the frontier of the blue region.

In [47]:

```
VaR = []
CVaR = []
ret = []
for k in range(1000):
    x = np.random.random(size = (n,1))
    x = x/np.sum(x)
    loss = np.dot(u, x)

    VaR.append(np.quantile(loss, alpha))
    CVaR.append(np.mean(loss[loss>VaR[k]]))
    ret.append(np.dot(mu,x))

plt.scatter(CVaR,ret)
plt.hlines(.13,.4,2, colors='red')
plt.title("Figure 1")
plt.xlabel("CVaR at 95%")
plt.ylabel("Portfolio return")
```

Out[47]:

```
Text(0, 0.5, 'Portfolio return')
```



We seek the portfolio so as to minimise CVaR at a given probability α and suppose the scenarios have equal probability $1/S$ to occur:

$$\begin{cases} \min \gamma + \dfrac{1}{(1-\alpha)S} \sum_{s=1}^{S} z_s \\ z_s \geq 0 \text{ for } s = \{1, \dots, S\} \\ z_s \geq (b - y_s)^T x - \gamma \\ \sum_{i=1}^{n} x_i = 1 \text{ for } i = \{1, \dots, n\} \\ \sum_{i=1}^{n} \mu_i x_i \geq R \\ 0 \leq x_i \leq 1 \end{cases} \tag{9}$$

Because $f(x, y_s)$ is linear in x, we can solve (8) for CVaR and VaR using by simplex method. The function linprog of scipy.optimize is a function that employs simplex method. The function support to solve a general linear optimisation problem of the form:

$$\begin{cases} \min c^T w \\ A_{eq} w = b_{eq} \\ A_{ub} w \leq b_{ub} \\ w \geq 0 \end{cases} \tag{10}$$

For our problem, $w = (x, \gamma, z)$ is the vector of controlling variables.

$$w = (x, \gamma, z)$$

$$c = (0 \times R^n, 1, v \times R^S) \text{ where } v = \frac{1}{S(1-\alpha)}$$

$$A_{eq} = (1 \times R^n, 0)$$

$$b_{eq} = 1$$

$$A_{ub} = \begin{pmatrix} -\mu & 0 & 0 \\ U & -1 & -I \end{pmatrix} \text{ where } I \text{ is the identity matrix}$$

$$\text{and } U = \begin{pmatrix} b - y_1 \\ b - y_2 \\ \dots \\ b - y_S \end{pmatrix}$$

$$b_{ub} = \begin{pmatrix} -R \\ 0 \end{pmatrix}$$

Consider the same portfolio as above and suppose we want the minimum return on the portfolio return is $R = 13\%$.

In [41]:

```
R = 0.13
v = 1/S/(1-alpha)

c = np.zeros(S+n+1)
c[n] = 1
c[n+1:] = v

Aeq = np.zeros([1,S+n+1])
Aeq[0,:n] = 1

Aub = np.zeros([S+1,S+n+1])
Aub[0,:n] = [-mui for mui in mu]
for s in range(S):
    Aub[s+1, :n] = u[s,:]
    Aub[s+1, n] = -1
    Aub[s+1, n+s+1] = -1

bub = np.zeros(S+1)
bub[0] = -R

from scipy.optimize import linprog as lp

x = lp(c = c, A_ub = Aub, b_ub = bub, A_eq = Aeq, b_eq = np.array([1]))
print(x)
```

```
     con: array([-2.95905522e-11])
     fun: 0.6526978053081114
 message: 'Optimization terminated successfully.'
     nit: 15
   slack: array([1.77036164e-12, 1.35103209e+00, 3.75460286e-02, ...,
       1.98482917e+00, 3.85723509e-10, 8.48434938e-01])
  status: 0
 success: True
       x: array([2.95946268e-01, 2.72071643e-01, 4.31982089e-01, ...,
       3.70283584e-10, 2.53118973e-01, 3.54339421e-10])
```

In [42]:

```
print("Portfolio structure:", x.x[:n])
print("VaR: ", x.x[n])
print("CVaR: ", x.fun)
```

```
Portfolio structure: [0.29594627 0.27207164 0.43198209]
VaR:  0.3024902954123438
CVaR:  0.6526978053081114
```

The result agrees with the premilinary analysis in Figure 1.

Now suppose that we want to maximise the return of this portfolio such that its CVaR does not exceed a threshold h_α, the problem boils down to:

$$\begin{cases} \max \sum_{i=1}^{n} \mu_i x_i \text{ for } i = \{1, \ldots, n\} \\ \gamma + \dfrac{1}{(1-\alpha)S} \sum_{s=1}^{S} z_s \leq h_\alpha \\ z_s \geq 0 \text{ for } s = \{1, \ldots, S\} \\ z_s \geq (b - y_s)^T x - \gamma \\ \sum_{i=1}^{n} x_i = 1 \\ 0 \leq x_i \leq 1 \end{cases} \tag{11}$$

We can verify the result of previous by supposing $h_{0.95} = 0.65$, two parts must yield the same result so we can expect the optimal portfolio return should be somewhere around 13%.

In [45]:

```
h = 0.65

c = np.zeros(S+n+1)
c[:n] = [-mui for mui in mu]

Aeq = np.zeros([1,S+n+1])
Aeq[0,:n] = 1

Aub = np.zeros([S+1,S+n+1])
Aub[0,n] = 1
Aub[0,n+1:] = v
for s in range(S):
    Aub[s+1, :n] = u[s,:]
    Aub[s+1, n] = -1
    Aub[s+1, n+s+1] = -1

bub = np.zeros(S+1)
bub[0] = h

from scipy.optimize import linprog as lp

x = lp(c = c, A_ub = Aub, b_ub = bub, A_eq = Aeq, b_eq = np.array([1]))
print(x)
```

```
     con: array([-1.30856215e-10])
     fun: -0.12994218753555395
 message: 'Optimization terminated successfully.'
     nit: 45
   slack: array([1.92222427e-09, 1.34661603e+00, 3.27336101e-02, ...,
       1.97913141e+00, 2.50640140e-07, 8.47003280e-01])
  status: 0
 success: True
       x: array([2.97783503e-01, 2.71549079e-01, 4.30667418e-01, ...,
       2.50456043e-07, 2.52743115e-01, 2.50461704e-07])
```

In [46]:

```
print("Portfolio structure:", x.x[:n])
print("VaR: ", x.x[n])
print("CvaR: ", np.dot(Aub[0], x.x))
print("Portfolio return: ", -x.fun)
```

```
Portfolio structure: [0.2977835  0.27154908 0.43066742]
VaR:  0.3007296174296259
CvaR:  0.6499999980777755
Portfolio return:  0.12994218753555395
```

The result also agress with previous part.

Reference

[1] https://analystprep.com/study-notes/frm/part-1/valuation-and-risk-management/binomial-trees/ (https://analystprep.com/study-notes/frm/part-1/valuation-and-risk-management/binomial-trees/)