Problem 1.25 a)

Let's define E_i as the event in which the random algorithm for min-cut finds a minimum cut in its *i*th execution. We know from the textbook that $P(E_i) = \frac{2}{n(n-1)}$. Conversely, the probability of not finding the minimum cut is $P(\bar{E}_i) = 1 - P(E_i) = 1 - \frac{2}{n(n-1)}$.

Since every execution of the random algorithm for min-cut is independent from each other, we know that:

$$P(E_i \cap E_j) = P(E_i)P(E_j) \quad \text{and} \quad P(\bar{E}_i \cap \bar{E}_j) = P(\bar{E}_i)P(\bar{E}_j)$$

Defining Y_m as event in which the algorithm finds the minimum cut after *m* attempts. We define its probability as any case except the one in which all attempts failed:

$$P(Y_m) = 1 - P(\bar{E}_1 \cap \bar{E}_2 \ldots \cap \bar{E}_m) = 1 - P(\bar{E}_1)P(\bar{E}_2)\ldots P(\bar{E}_m) = 1 - \prod_{i=1}^{m} P(\bar{E}_i)$$

Which, in general for any *m* and *n*, is (and can be bound to):

$$= 1 - \left(1 - \frac{2}{n(n-1)}\right)^m \geq 1 - e^{\frac{2m}{n(n-1)}}$$

In the case of $m = 2$, this would be equal to:

$$= 1 - \left(1 - \frac{2}{n(n-1)}\right)\left(1 - \frac{2}{n(n-1)}\right) = \frac{\mathbf{4}}{\boldsymbol{n(n-1)}} - \frac{\mathbf{4}}{\boldsymbol{n^2(n-1)^2}}$$

Since we are running the algorithm twice, the number of edge contractions will double as well, that is:

$$\#\,EdgeContractions = \mathbf{2(n-2) = 2n-4}$$

Problem 1.25 b)

In this approach, we first reduce the original *n* vertex graph G by executing n-*k* iterations of our randomized algorithm. This will result in a *k* vertex graph G' with two possibilities:

- **Event A:** No edges of C have been contracted, therefore it is still possible to find C in G'. (note: C is the set of edges conforming a minimum cut)
- **Event B:** One or more edges of C have been contracted, therefore there no further processing will be able to return a minimum cut from G'.

The probability of A is that of having executed the algorithm *k* contractions:

$$P(A) = P(F_{n-k}) = \prod_{i=1}^{n-k}\left(\frac{n-i-1}{n-i+1}\right) = \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right)\cdots\left(\frac{k-1}{k+1}\right) = \frac{k(k-1)}{n(n-1)}$$

The probability of B is any other possible outcome:

$$P(B) = 1 - P(A)$$

If B occurs, then all further executions of the algorithm will be to no avail. Therefore, assuming that A occurs, we proceed to apply the randomized algorithm to G' *l* times. We will call Q the event in which at least one of those attempts succeed in finding the minimum cut in G'.

In this case, the amount of vertexes of G' will be *k*, and each of the *l* attempts will iterate *k-2* times. By the results obtained in problem 25a, we determine that the probability Q assuming A is:

$$P(Q|A) = 1 - \left(1 - \frac{2}{k(k-1)}\right)^l$$

However, we need to consider Q under the possibility that A occurs or not, therefore the possibility of finding the minimum cut of G by this method is bound by:

$$P(Q) = P((Q|A)\cap A) =$$

$$P(Q|A)P(A) = \frac{k(k-1)}{n(n-1)} - \frac{k(k-1)}{n(n-1)}\left(1 - \frac{2}{k(k-1)}\right)^l \geq \boldsymbol{\frac{k(k-1)}{n(n-1)}\left(1-e^{\frac{-2l}{k(k-1)}}\right)}$$

The total amount of edge contractions will be:

$$\#\,EdgeContractions = \boldsymbol{n-k + (k-2)l}$$

Problem 1.25 c)

Having run the original algorithm twice would have demanded the following amount of edge contractions:

$$\#\,MaxEdgeContractions = 2(n-2) = 2n-4$$

If we are restricted to use this amount of contractions for our *k*, *l* approach, then:

$$n-k + (k-2)l = 2n-4$$

$$(k-2)l = n-4+k$$

$$l = \left\lfloor\frac{n-4+k}{k-2}\right\rfloor \quad \text{for any } k > 3 \qquad \text{(Equation 1)}$$

We can then conclude that $l = f(n, k)$. (That is: given n and k, l can be calculated exactly).

To increase our probability of finding min cut, we would try to maximize P(Q). Recalling:

$$P(Q) = \frac{k(k-1)}{n(n-1)} - \frac{k(k-1)}{n(n-1)}\left(1 - \frac{2}{k(k-1)}\right)^{l} \qquad \text{(Equation 2)}$$

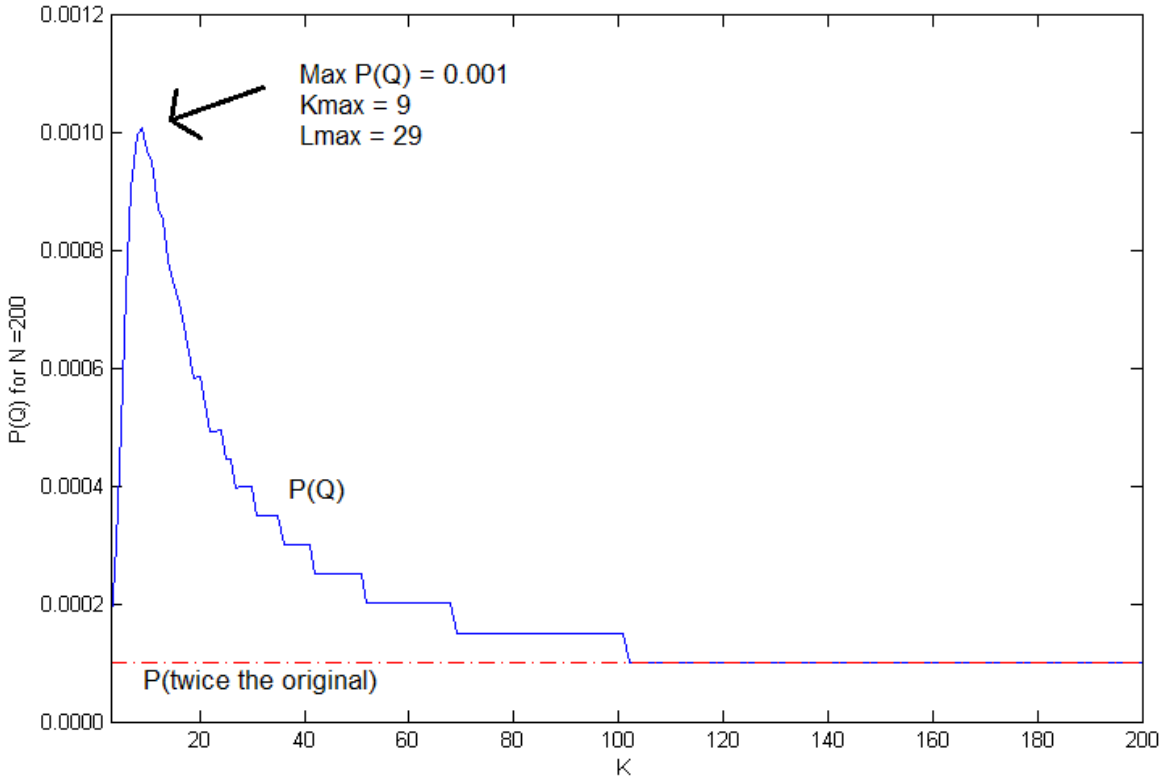
The best way to compute the maximizing k as a function of n is to differentiate the equation below and equate it to zero. This procedure could be quite tricky. A more forgiving approach is to perform a numerical evaluation to try to define a lower and upper boundary to the maximizing function.

In order to find the maximizing k numerically I defined the following simple algorithm with complexity O(n):

```
Have n as input
Define Pmax = 0
Define Kmax, Lmax

for k = 3 to n
 l = f(n,k) as per Equation 1
 P = P(Q) given n, k, l, as per Equation 2
 if P > Pmax then
    Pmax = P
    Kmax = k, Lmax = l
 end if
end for

return Kmax, Lmax
```

This algorithm will iterate all possible values of k given a certain n, then obtain l (which corresponds to the maximum amount of contractions given the restriction), and calculate P(Q) with those values. The maximum P(Q) will determine the maximizing values of k and l for the given n.

In the following figure we can see the distribution of probability of varying k given a certain n = 200.



From the illustration, a handful of observations can be made:

1 – The evolution of P(Q) shows a square-like format because of the floor operation on calculating *l*, since *l* can take only integer values.

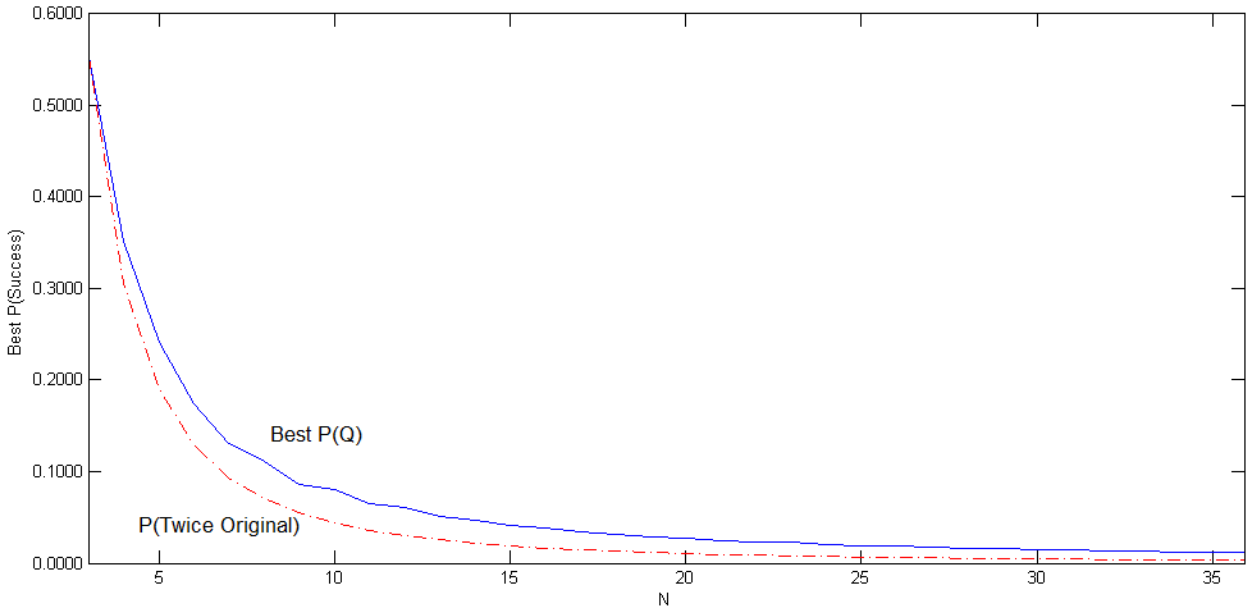
2 – The worst probability of the new approach is exactly the probability of success for the original algorithm ran twice for all $k \geq n/2$ This is because with *k* more than the half of *n*, this means that only $l = 2$ repetitions can be made, which emulates exactly the behavior of running the original algorithm twice.

3 – On the other hand, for $k \leq n/2$ the probability of success is 10 times more than that of the original algorithm ran twice.

The MATLAB code for running this experiment is defined below:

```
n = 200; % Define N
PQ = zeros(n-2,1); % Initialize P(Q)
POrig = zeros(n-2,1); % Initialize P of running the original algo twice
Pmax = 0; % Variable to keep the maximum P(Q)
format short g
for i = 3:n
    k = i; % Assign a value of k
    l = floor((n-4+k)/(k-2)); % Calculate l as a f(k,n)
    PA = (k*(k-1)) / (n*(n-1)); % Calculate P(A)
    PQA = 1 - (1 - 2/(k*(k-1)))^l; % Calculate P(Q|A)
    PQ(i-2) = PA * PQA; % Calculate P(Q) = P(Q|A)P(A) since they are independent
    POrig(i-2) = 1 - (1 - 2 / (n*(n-1)))^2; % Calculate P(Original)
    if P(i-2) > Pmax;
        Pmax = P(i-2)
        Kmax = k
        Lmax = l
    end
end

%Plotting stuff
plot(3:n, PQ, 3:n, POrig, '-.r'  )
xlabel('K');
ylabel(strcat('P(Q) for N = ', num2str(n)));
yt = get(gca,'YTick');
set(gca,'YTickLabel', sprintf('%.4f|',yt))
xlim('manual')
xlim([3 n]);
```

It would also be interesting how this approach compares to twice the original approach in terms of probability of success, given different values of N. At every N, the maximizing combination of *k* and *l* is selected for P(Q), as per the algorithm previously defined. The following plot shows the comparative performance:

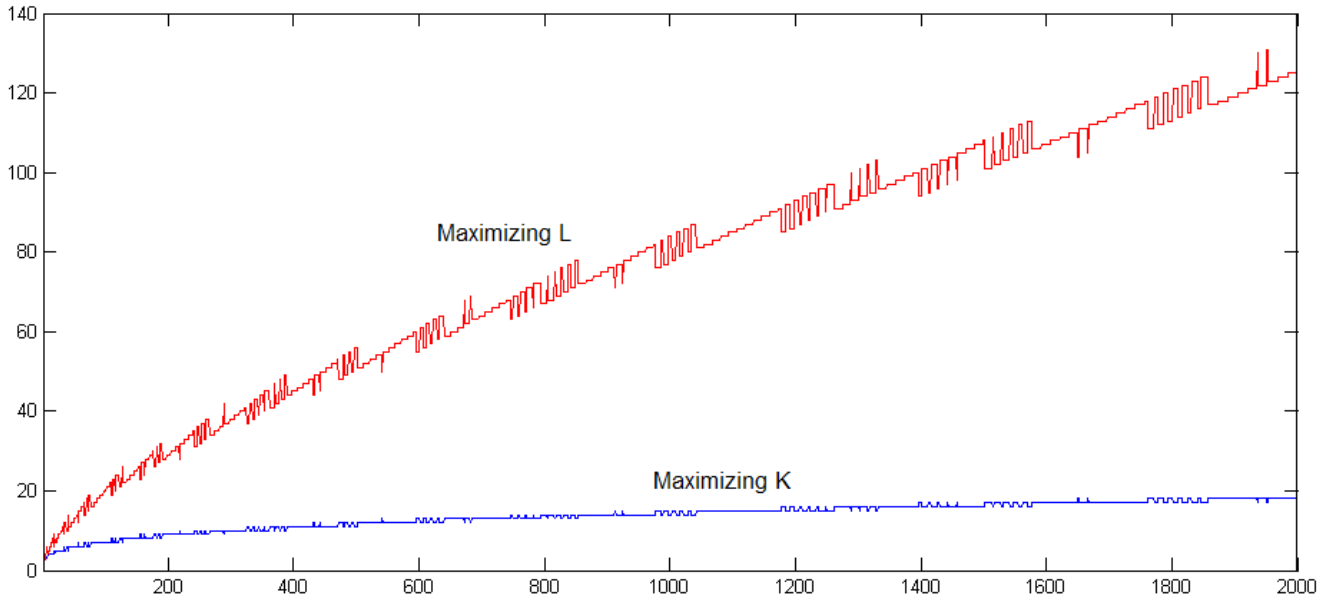


From the figure it can be seen that selecting the maximizing k and l for the new approach allows a better probability of success for every $n > 3$, and equal probability for $n = 3$. The MATLAB code for this experiment is defined below:

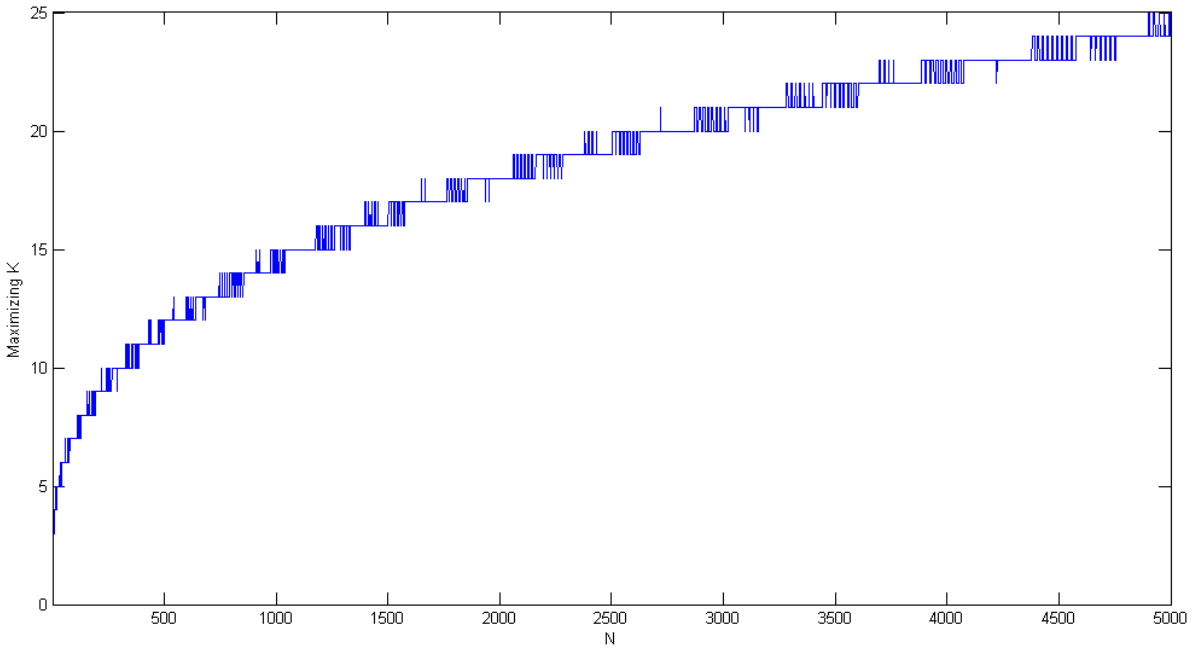
```
n_max = 200; % Define N
Pmax = zeros(n_max,1); % Initialize the array of Pmax
POrig = zeros(n_max,1); % Initialize P of running the original algo twice
format short g

for n = 3:n_max
    Pmax(n) = 0; % Variable to keep the maximum P(Q)
    for i = 3:n
        k = i; % Assign a value of k
        l = floor((n-4+k)/(k-2)); % Calculate l as a f(k,n)
        PA = (k*(k-1)) / (n*(n-1)); % Calculate P(A)
        PQA = 1 - (1 - 2/(k*(k-1)))^l; % Calculate P(Q|A)
        PQ = PA * PQA; % Calculate P(Q) = P(Q|A)P(A) since they are independent
        if PQ > Pmax(n);
            Pmax(n) = PQ;
        end
    end
    POrig(n) = 1 - (1 - 2 / (n*(n-1)))^2; % Calculate P(Original)
end

%Plotting stuff
plot(1:n, Pmax, 1:n, POrig, '-.r'  )
xlabel('N');
ylabel('Best P(Success) ');
yt = get(gca,'YTick');
set(gca,'YTickLabel', sprintf('%.4f|',yt))
xlim('manual')
xlim([1 n_max]);
ylim('manual')
ylim([0.0 0.6]);
```

Even though we have an O(n) algorithm for calculating the maximizing k and l, we would like to have a lower/upper bound so that we know that selecting a k' within such boundaries (in constant time) will approximate to the original k' with a bounded error margin ε.

For this, it would be useful to see how the maximizing k and l evolve as a function of n. Such evolution can be seen in the following plot:



It can be initially observed that both elements show a somewhat logarithmic progression. However, since *l* is a function of *k*, we are only interested in bounding *k*. The following plot shows only the maximizing *k* for every *n* from 3 to 5000:



The MATLAB code to generate both plots is shown below:

```
n_max = 5000; % Define N
Pmax = zeros(n_max,1); % Initialize the array of Pmax
Kmax = zeros(n_max,1); % Initialize the array of Kmax
Lmax = zeros(n_max,1); % Initialize the array of Lmax

for n = 3:n_max
    Pmax(n) = 0; % Variable to keep the maximum P(Q)
    for i = 3:n
        k = i; % Assign a value of k
        l = floor((n-4+k)/(k-2)); % Calculate l as a f(k,n)
        PA = (k*(k-1)) / (n*(n-1)); % Calculate P(A)
        PQA = 1 - (1 - 2/(k*(k-1)))^l; % Calculate P(Q|A)
        PQ = PA * PQA; % Calculate P(Q) = P(Q|A)P(A) since they are independent
        if PQ > Pmax(n);
            Pmax(n) = PQ;
            Kmax(n) = k;
            Lmax(n) = l;
        end
    end
end

%Plotting stuff
figure
plot(1:n, Kmax )
xlabel('N');
ylabel('Maximizing K');
xlim('manual')
xlim([1 n_max]);
figure
plot(1:n, Kmax, 1:n, Lmax, 'r' )
xlabel('N');
xlim('manual')
xlim([1 n_max]);
```

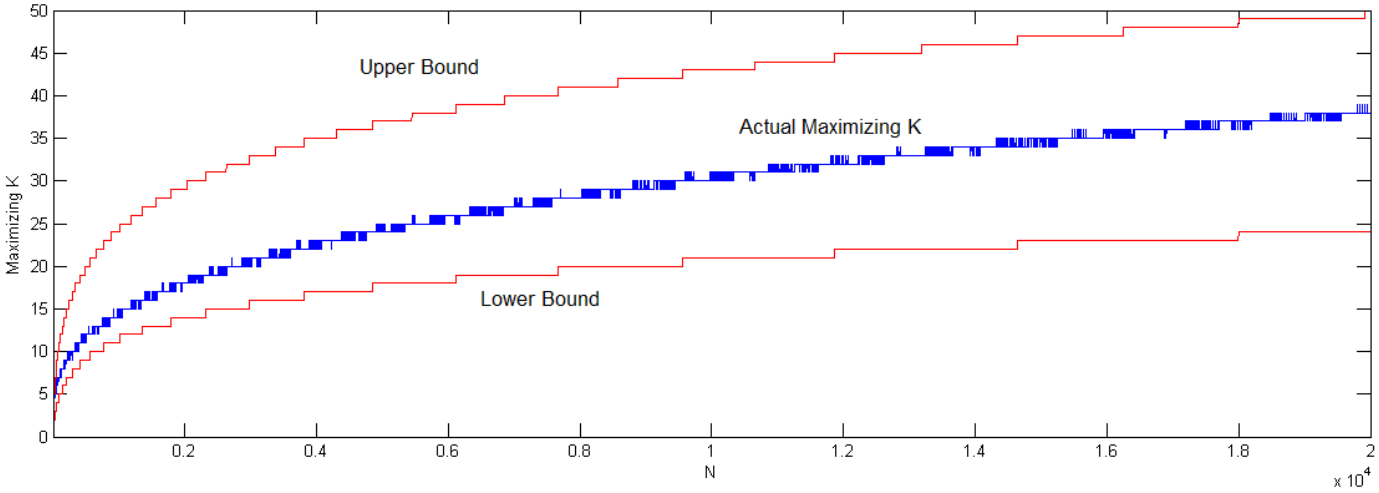
After a series of trials, my best approximation to such boundaries is:

$$\lfloor \frac{\ln^2(n)}{4} \rfloor < k < \lceil \frac{\ln^2(n)}{2} \rceil$$

This holds true at least for all $n < 20000$. The error margin $\varepsilon(n)$ will be the maximum difference between the two boundaries for a given n, thus:

$$max\,\varepsilon(n) = \frac{\ln^2(n)}{2}$$

The plot for the upper and lower bound can be seen in the following plot:



The MATLAB code to generate this plot is shown below:

```
n_max = 20000; % Define N
Pmax = zeros(n_max,1); % Initialize the array of Pmax
Kmax = zeros(n_max,1); % Initialize the array of Kmax
Lmax = zeros(n_max,1); % Initialize the array of Lmax
LowerBound = zeros(n_max,1);
UpperBound = zeros(n_max,1);


for n = 3:n_max
    Pmax(n) = 0; % Variable to keep the maximum P(Q)
    for i = 3:n
        k = i; % Assign a value of k
        l = floor((n-4+k)/(k-2)); % Calculate l as a f(k,n)
        PA = (k*(k-1)) / (n*(n-1)); % Calculate P(A)
        PQA = 1 - (1 - 2/(k*(k-1)))^l; % Calculate P(Q|A)
        PQ = PA * PQA; % Calculate P(Q) = P(Q|A)P(A) since they are independent
        if PQ > Pmax(n);
            Pmax(n) = PQ;
            Kmax(n) = k;
            Lmax(n) = l;
        end
    end
    LowerBound(n) = floor(log(n)*log(n)/4); % Calculate lower bound for given n
    UpperBound(n) = ceil(log(n)*log(n)/2); % Calculate upper bound for given n
end

%Plotting stuff
plot(1:n, Kmax, 1:n, LowerBound, 'r', 1:n, UpperBound, 'r' )
xlabel('N');
ylabel('Maximizing K');
xlim('manual')
xlim([1 n_max]);
```