Let $G$ be a connected weighted undirected graph. Suppose that the maximum weight in $G$ is $m$, and there is only one edge, $e^*$, that has that weight. For example, $G$ might look like this, where $m = 5$ and $e^* = \{d, e\}$:

![Graph Diagram]

Prove the following statement:
If there is some spanning tree that does not contain the edge $e^*$, then no Minimum Spanning Tree can contain $e^*$.

[We are expecting: A formal proof.]

Let $T$ be a spanning tree that contains the edge $e^*$. We want to show that $T$ is not an MST. Let $T^*$ be the spanning tree that does not contain the edge $e^*$, as guaranteed by the problem statement.

If we remove $e^*$ from $T$, we are left with two trees, call them $T_1$ and $T_2$, and consider the cut formed by the vertices of $T_1$ and the vertices of $T_2$.

Since $T^*$ is a spanning tree, it must cross this cut. Say that $\{u, v\}$ is the edge in $T^*$ that crosses this cut. Since $e^*$ has the unique max weight, the weight of $\{u, v\}$ must be smaller. Consider the spanning tree that you get by removing $e^*$ from $T$ and adding in $\{u, v\}$ to $T$. This still spans, because we haven’t changed the number of vertices it touches, and it is still a tree since it spans and has only $n - 1$ edges.

But now the cost is smaller than that of $T$, so $T$ cannot have been an MST.
Consider the graph $G$ below.

1. In what order does Prim’s algorithm add edges to an MST when started from vertex C?
2. In what order does Kruskal’s algorithm add edges to an MST?

[We are expecting: For both, just a list of edges. You do not need to draw the MST, and no justification is required.]

1. Prim’s algorithm adds edges in the order:
   \{C,F\}, \{F, E\}, \{E, D\}, \{A, D\}, \{A, B\}

2. Kruskal’s algorithm returns the same tree, and adds edges in the order:
   \{A,D\}, \{A, B\}, \{D, E\}, \{F, C\}, \{E,F\}
Consider the following graph:

1. What is the global minimum cut of this graph?
   **[We are expecting: Just the global minimum cut. No explanation is required.]**
2. What is the probability that Karger’s algorithm chooses an edge crossing the minimum cut with its first choice?
   **[We are expecting: Just the probability. No explanation is required.]**
3. What is the exact probability that one run of Karger’s algorithm returns a minimum cut on this graph? How does it compare to the bound of $1/(\binom{n}{2})$ that we saw in class?
   **[We are expecting: Your numerical answer (no justification required), as well as a statement about how this compares to $1/(\binom{n}{2})$.]**

1. The minimum cut is $\{A, B, C, D\}, \{E\}$.
2. The probability that Karger’s algorithm chooses the edge $\{B, E\}$ which crosses the cut on the first round is $1/5$.
3. We look at the probability that Karger’s algorithm fails first. We claim that this is given by
   \[
   \frac{1}{5} + \frac{4}{5} \left( \frac{1}{4} + \frac{3}{4} \cdot \frac{1}{3} \right) = \frac{3}{5}.
   \]
   This is because, as in part (a), the probability of choosing $\{B, E\}$ is $1/5$ in the first step. If we don’t do that (with probability $4/5$), then we consider the probability of choosing $\{B, E\}$ in a graph where one of the other four edges has been collapsed, which looks like this:

   ![Diagram](image1)

   This probability is $1/4$ (the probability that we choose $\{B, E\}$), plus $3/4$ times the probability that we choose $\{B, E\}$ in a graph that looks like this:

   ![Diagram](image2)

   That final probability is $1/3$. Putting this all together gives the answer above, which is that the probability of choosing $\{B, E\}$ is $3/5$. Thus the probability that this never happens (that Karger’s algorithm returns the minimum cut) is $1 - 3/5 = 2/5$.

On the other hand $1/(\binom{n}{2}) = 1/(\binom{5}{2}) = 1/10$, which is much smaller. So that bound was pessimistic.

**Note:** Another way to compute the probability is to consider the fact that at each step, we succeed if we avoid the one edge $\{B, E\}$. In the first step we have 5 edges, then 4, and then finally 3 (as in the diagrams above), so the probability is
\[
\frac{4}{5} \cdot \frac{3}{4} \cdot \frac{2}{3} = \frac{2}{5}.
\]
You join a nomadic tribe traveling through their fixed route of $n$ ancient queendoms, going through queendoms in increasing order. This is a cool adventure, but also a nice opportunity to make money! In each queendom, you can sell any foreign currency and receive the local currency in exchange (but no other exchanges are allowed). You have an $n \times n$ table of the exchange rates between every pair of currencies. You start the journey with 1 unit of currency of the first queendom. Your goal is to buy and sell during the trip in order to maximize the money you’ll have in the $n$-th queendom’s currency, since you plan to stay there for a while.

There is one more restriction: in order to not attract too much attention to your side business, you should only exchange money in $k$ queendoms.

Design an algorithm to determine the maximum of the $n$-th queendom’s currency you can have.

**Input:** $n$: the number of queendoms; $k$: the number of exchanges allowed; $A$: an $n \times n$ table of exchange rates.

**Output:** the number of units of the $n$-th queendom’s currency

**Additional assumptions:** You may assume that $k$ is much smaller than $n$, but not as small as $O(1)$. Also, you do not exchange in the first queendom (since you already have that currency), and you always exchange in the last queendom.

**Example 1** In this example, the optimal strategy is to exchange in both the second and third queendom, and end up with $2 \times 2 = 4$ units of the last currency (exchanging only in the last one would give 2.5).

Note: In the example $A[i][j] = c$ means that exchanging 1 unit of currency $i$ yields $c$ units of currency $j$.

**Input:**
$n = 3$
$k = 2$
$A = \begin{bmatrix} 1.0 & 2.0 & 2.5 \\ 0.5 & 1.0 & 2.0 \\ 0.4 & 0.5 & 1.0 \end{bmatrix}$

**Output:**
4

**Example 2** In this example, the optimal strategy is to exchange only in the third queendom, and end up with 2.5 units of the last currency (exchanging in both second and third one would give $0.5 \times 2 = 1$).

**Input:**
$n = 3$
$k = 2$
$A = \begin{bmatrix} 1.0 & 0.5 & 2.5 \\ 2.0 & 1.0 & 2.0 \\ 0.4 & 0.5 & 1.0 \end{bmatrix}$

**Output:**
2.5
Give a short but clear English description of your algorithm.

[We are expecting: A clear yet thorough English description of your algorithm.]

We maintain an \( n \times k \) one-indexed array \( M \), in which entry \( M[i][j] \) gives the optimal amount of currency \( i \) that can be obtained after \( j \) exchanges and output \( M[n][k] \). To compute each entry \( M[i][j] \) where \( j = 1 \), directly convert currency \( i \) to currency \( j \). For remaining entries \( M[i][j] \), take the maximum of the product of the exchange rate and the currency held for \( j - 1 \) exchanges among the set of previous queendoms.

Is your algorithm correct (Yes/No)?

[We are expecting: A clear answer of YES or NO. If YES, no explanation is necessary, if NO, explain why your algorithm is incorrect.]

Yes

Provide the pseudocode for your algorithm.

[We are expecting: Detailed pseudocode that matches your English description. You are free to use an interface of any of the algorithms covered in lecture.]

```python
def exchangeCurrency(n, k, A):
    M = an n by k array
    for i in 1..n:
        M[i][1] = A[1][i]
    for j in 1..k:
        M[1][j] = 1
    for all remaining (i, j):
        M[i][j] = max of M[i'][j-1] * A[i'][i] for all i' < i
    return M[n][k]
```

Provide an analysis of the runtime of your algorithm.

[We are expecting: A detailed analysis of the runtime of your algorithm including the Big-O time in terms of \( n \) and \( k \).]

There are \( O(nk) \) entries in \( M \). To compute each entry, we must check \( O(n) \) prior entries for an overall runtime of \( O(n^2k) \).
There are $n$ mice and $n$ holes along a line. Each hole can accommodate only 1 mouse. A mouse can stay at its position, move one step right from $x$ to $x + 1$, or move one step left from $x$ to $x - 1$. Any of these moves consumes 1 minute. Mice can move simultaneously. Assign mice to holes such that the time it takes for the last mouse to get to a hole is minimized, and return the amount of time it takes for that last mouse to get to its hole.

Example: Mice positions: 4, -4, 2
Hole positions: 4, 0, 5
Best case: The last mouse gets to its hole in 4 minutes (\{4 \rightarrow 4, -4 \rightarrow 0, 2 \rightarrow 5\} and \{4 \rightarrow 5, -4 \rightarrow 0, 2 \rightarrow 4\} are both possible solutions)

Sort the mice locations and the hole locations. For $0 \leq i < n$, have the $i$th mouse go to the $i$th hole. The maximum distance will be the max distance between each mouse and its corresponding hole.

Justification: We’ll first prove a lemma that will be useful in our induction proof later. **Lemma:** For $i_1 < i_2, j_1 < j_2$ and $\text{dist}(x, y) = |x - y|$. 

\[
\max(\text{dist}(i_1, j_1), \text{dist}(i_2, j_2)) \leq \max(\text{dist}(i_1, j_2), \text{dist}(i_2, j_1))
\]

Without loss of generality, let’s say $i_1 \leq j_1$. Our cases are then that $i_1 \leq i_2 \leq j_1 \leq j_2$, $i_1 \leq j_1 \leq i_2 \leq j_2$, or $i_1 \leq j_1 \leq j_2 \leq i_2$. In any of these cases, the lemma holds.

Now, we proceed with induction.

**Inductive hypothesis.** By sending the $i$th(sorted) mouse to the $i$th(sorted) hole, there is a minimal solution that extends the current solution.

**Base case.** If we haven’t sent any mice to any holes, we haven’t eliminated the ideal solution.

**Inductive Step.** Suppose that we have sent the first $k - 1$ sorted mice to the first $k - 1$ sorted holes. Now suppose there is an optimal solution where the $k$th mouse is sent to the $p_0$th hole, where $k < p_0 < n$, the $p_0$th mouse is sent to the $p_1$th hole, and so on until the $p_d$th (for some $d$) mouse is sent to the $k$th hole. We could then swap the $p_0$th hole with the $k$th hole; we know by our lemma that the result will not be worse than the optimal solution. Therefore, by sending the $k$th mouse to the $k$th hole, we have not eliminated an optimal solution.

**Conclusion.** By the $n$th step, we have not ruled out the optimal solution. Therefore, the solution we chose is optimal.