14.5: Generating a random permutation:

Initially, let $arr[i] = i$ for $i = 1$ to $n$
for $i = 1$ to $n-2$
    $j = \text{random}(i)$ //generates a random integer between 1 and $i-1$
    swap($arr[i]$, $arr[j]$)
end for

Assuming $\text{random}(i)$ takes $O(1)$ time, this works in linear time.
The correctness of this algorithm can be proved by induction.
14.8: Analysis of a Las Vegas algorithm based on a Monte Carlo Algorithm:

```
Algorithm montecarloToLasVegas

A = monte carlo algorithm
A' = Las Vegas Algorithm

1. correct <- false

2. while !correct 
3.   A 
4.   verify correctness(A) 
5.   if correct solution 
6.     correct <- true 
7.   end if 
8. end while 
```

Looking at the algorithm above, we see that to go from a Monte Carlo algorithm to a Las Vegas algorithm in this instance, you need to keep calling the Monte Carlo algorithm until the verified solution is correct.

To do this, it takes $T(n)$ time to run the Monte Carlo algorithm and $T'(n)$ time to verify that the solution is correct. So the running time so far is $T(n) + T'(n)$.

Now to take into account the probability that the algorithm (Monte Carlo) will give a correct solution, we know we are going to have to divide by this probability to account for the number of times the algorithm (Monte Carlo) will have to be run to get the correct answer.

This gives us: \[
\frac{T(n) + T'(n)}{p(n)}
\]
14.9: Probability amplification of a Monte Carlo Algorithm:

The algorithm works correctly with probability at least $1 - \epsilon_1$.

$\Rightarrow$ The algorithm produces incorrect answer with probability at most $\epsilon_1$ in one iteration.

In $t$ iterations, the algorithm produces incorrect answer in all $t$ iterations with probability $(\epsilon_1)^t$.

However, we want to make the correctness probability at least $1 - \epsilon_2$ i.e. the incorrectness probability at most $\epsilon_2$.

Therefore, $\epsilon_1^t \leq \epsilon_2 \Rightarrow t \ln(\epsilon_1) \leq \ln(\epsilon_2) \Rightarrow t \geq \frac{\ln(\epsilon_2)}{\ln(\epsilon_1)}$ since $\epsilon_1 < 1 \Rightarrow \ln(\epsilon_1) < 0$ so the direction of the inequality reverses.

Therefore, you need at least $\frac{\ln(\epsilon_2)}{\ln(\epsilon_1)}$ iterations.

14.10: Comparison between linear search and randomized search:

Let $x$ be the required element, and we need to find an index $j$ such that $x_j = x$.

1. Randomized algorithm:
   Let $X$ be the random variable that is 1 if $x_j = 1$ and 0 otherwise. Since there are $k$ occurrences of $x$ in the array and the index is generated uniformly at random, $E[X] = \Pr[X = 1] = \frac{k}{n}$.

   Now expected value of the number of random generations until the $x$ is found is similar to calculating expectation of a geometric variable, and so in expectation there will be $\frac{k}{\frac{k}{n}}$ iterations until the element is found.

2. Linear search:
   Linear search takes $O(n)$ time to find an element in worst case. The actual running time depends on the distribution of the array.

   Therefore, both algorithms take $O(n)$ time.

14.13: Randomized algorithm to check if matrix product $AB = C$.

```
Algorithm - checking AB = C
Inputs: A, B, C matrices and x, a vector of n random entries.

1. Compute BX - takes $\Theta(n^2)$
2. Compute CX - takes $\Theta(n^2)$
3. Compute ABX - takes $\Theta(n^2)$
4. If ABX = CX then return true else return false.
```

When testing $AB = C$, if we get that $AB \neq C$, then there is at least one row in $C$ that is different than that corresponding row in $AB$. Since $AB = C$ and $x$ is an $n \times 1$ matrix, the probability that $ABX \neq CX$ is at most $0.5$. For the above algorithm to work, $ABX = CX$ has to be true so the probability of it failing is $0.5$. 