Solution for Homework 4

4.1

No unique answers. A possible one may be:

In each direction above, the object has a cylinder and a cone frustum joint together.

4.2

Given polygon P with n edges (facet), its mold would contain n-1 ordinary edges (facets) of P. The outward normal of these edges are \( \{ \eta_{1, x}, \eta_{1, y} \}, \{ \eta_{2, x}, \eta_{2, y} \}, \ldots, \{ \eta_{n-1, x}, \eta_{n-1, y} \} \). Suppose we have a translation direction \( \{ d_x, d_y \} \). To test whether we can move P out of the mold by this direction, we only have to test whether this set of linear constraint can be satisfied simultaneously, according to Lemma 4.1:

\[
\begin{align*}
\eta_{1, x} d_x + \eta_{1, y} d_y &\leq 0 \\
\eta_{2, x} d_x + \eta_{2, y} d_y &\leq 0 \\
\vdots \\
\eta_{n-1, x} d_x + \eta_{n-1, y} d_y &\leq 0
\end{align*}
\]

Thus to decide whether P can be removed from a single translation, we only have to check whether the feasible region of the constraint set above is empty or not. We can achieve this by calling the randomized linear programming algorithm in Page 77, the expected running time of which is \( O(n) \).
4.12

(a) 
This problem can be expressed as: suppose we have n empty “seats”, and we want to assign n entries of Array A into these seats one by one. Apparently, the number of all possible assignments is $n!$, so each possible assignment should bear identical $\frac{1}{n!}$ probability. We now check whether this is true for RANDOMPERMUTATION(A) algorithm (Page 77).


When k=n, variable “rndindex” is uniformly distributed between 1 and n, so Seat[n] has equally probable choices $\frac{1}{n}$ to A[1], A[2], ..., A[n]. Suppose A[in] was selected;

When k=n-1, “rndindex” is uniformly distributed between 1 and n-1, so Seat[n-1] has equally probable choices $\frac{1}{n-1}$ to A[1], A[2], ..., A[n] EXCEPT A[in]. Suppose A[in-1] was selected;

....

When k=2, A[in], A[in-1], ..., A[in-3] has been selected by Seat[n], Seat[n-1], ..., Seat[n]. Thus Seat[2] only has two equally probable choices, which give Seat[1] the last one choices.

Therefore in general, any possible assignment has $\frac{1}{n} \cdot \frac{1}{n-1} \cdot 1 \cdot 1 = \frac{1}{n!}$ probability, thus the algorithm can generated uniformly random permutations.

(b)
Changing k in line 2 to n means each seat has full n choices, regardless of their indexes. So the total number of possible choice paths is $n^n$, each of which has probability $\frac{1}{n^n}$. As the number of possible permutation is $n! < n^n$, there should be multiple choice paths that render a common permutation.

Further more, as $n!$ cannot divide $n^n$, it is not possible that each permutation can be reached by the same number of choice paths. So, different permutations MUST have different number of reachable paths, thus has different probability to happen.

4.13

Given an integer k, its binary number requires $\left\lfloor \log k \right\rfloor + 1$ bits.

Procedure:
BIT_RANDOM (k)
DO
    Num = 0;
    FOR i = 1: \( \lfloor \log k \rfloor + 1 \)
        Num = Num + Bit() \* 2^{i-1};  // Bit() generate random 0 or 1 in constant time
    END
    WHILE Num > k or Num == 0
With this procedure to replace the RANDOM (k) (Page 77), we can still generate a random permutation.

**Complexity:**

Given k, we randomly generate a binary integer Num with identical number of bits, \( \lfloor \log k \rfloor + 1 \). The probability that Num is within \([1, k]\) is \( \frac{k}{2^{\lfloor \log k \rfloor + 1} - 1} \). Therefore, the times of “DO WHILE” in BITRANDOM (k) obeys a Geometric distribution with \( p = \frac{k}{2^{\lfloor \log k \rfloor + 1} - 1} \), and the expected times of loop-calls is \( \frac{1}{p} = \frac{2^{\lfloor \log k \rfloor + 1} - 1}{k} \). Meanwhile, each loop-call requires \( O(\lfloor \log k \rfloor + 1) \) time. So the total expected time of BITRANDOM (k) is \( O \left( \frac{2^{\lfloor \log k \rfloor + 1} - 1}{k} \cdot (\lfloor \log k \rfloor + 1) \right) \).

As \( \frac{2^{\lfloor \log k \rfloor + 1} - 1}{k} \cdot (\lfloor \log k \rfloor + 1) < 2 \cdot \frac{2^{\log k} }{k} \cdot (\lfloor \log k \rfloor + 1) = 2 \cdot (\lfloor \log k \rfloor + 1) \).

To generate a random permutation requires n-1 times’ call to BITRANDOM (k), so the total time is bounded by:

\( n \cdot 2^\left( \lfloor \log n \rfloor + 1 \right) = O(n \log n) \)

4.14

In worst-case, each time during recursion, the condition at line 5 is not satisfied. So we have to make comparison for card(A)-1 times. In this way, the complexity is recursively expressed as:

\( T(n) = T(n-1) + O(n) \)

T(1)=1, T(2)=T(1)+2=1+2, T(3)=T(2)+3=1+2+3, \ldots, T(n)=1+2+3+\ldots+n=n(n-1)/2. So:
\[ T(n) = O(n^2) \]

In random-case, if the condition at line 5 satisfies, it means the randomly selected \( x \) is not the biggest one. The probability it happens is \( \frac{n-1}{n} \), and line 6 of course requires \( O(1) \) time. Otherwise, \( x \) is the biggest. The probability it happens is \( \frac{1}{n} \), and the checking process at line 7 requires \( O(n) \) time. In general, the expected complexity is recursively expressed as:

\[
T(n) = T(n-1) + \frac{n-1}{n} \cdot O(1) + \frac{1}{n} \cdot O(n) = T(n-1) + O(1)
\]

So:

\[
T(n) = O(n)
\]

### 4.15

Suppose the simple polygon \( P \) is expressed as double-connected edge lists. We can check whether the intersection of half-planes bounded by \( P \)'s half-edges is empty or not. If it's empty, then \( P \) isn't star-shaped, vise versa. To decide the direction of half-plane associated with each half-edge \( e \) is easy: we can extract \( e\).origin(), \( e\).next().origin() thus establish the normal vector to its left side. The half-plane must contain the normal vector.

Now that we have a set of \( n \) linear constraints, we can use the randomized linear programming algorithm (Page 77) to decide whether the feasible region is empty or not, thus decide whether or not \( P \) is star-shaped. The expected running time is \( O(n) \)