1. What does de Casteljau algorithm do? Implement the algorithm in C and test the results using arbitrary control points.

곡선 위의 임의의 위치를 표현하는 식이 control points를 잇는 선의 연속된 내분 과정으로 바뀌었다. 이 과정에서 하나의 곡선을 표현하기 위한 control points의 집합이 유일하지 않음을 알 수 있다. 따라서 곡선의 형태를 바꾸지 않고 곡선을 둘로 나눌 수 있게 되었다.

```
/* File:
 *     deCasteljau.cpp
 *
 * Purpose:
 *     Implement the algorithm in C and
 *     test the results using arbitrary control points.
 *
 * Compile:
 *    g++ -g -Wall -o deCasteljau deCasteljau.cpp
 * Run:
 *    ./deCasteljau
 *
 * Input:
 *
 * Output:
 *    Coordinates at some value of t by Bezier and
 *    de Casteljau's algorithm.
 *
 */
#include <iostream>
#include <cmath>

#define DEGREE      3
#define PARAMETER   0.5

/**
 * Class:       Vertex
 * Purpose:     Generating vertice and printing their own coordinates.
 */
class Vertex {
public:
    double m_x, m_y, m_z;

/**
 * Function:    Default constructor
 */
    Vertex()
        : m_x(0.0), m_y(0.0), m_z(0.0) {};

/**
 * Function:    Copy constructor
 */
    Vertex( const double x,
            const double y,
            const double z)
        : m_x(x), m_y(y), m_z(z) {};

/**
 * Operator:    << (overloading)
 * Purpose:     Print coordinates to output stream.
 */
    friend std::ostream &operator<<(std::ostream& strm,
                                    const Vertex& obj) {
```

```
        strm << "(" << obj.m_x << ", "
                    << obj.m_y << ", "
                    << obj.m_z << ")";
    };

/**
 * Function:    Destructor
 */
    ~Vertex() {};
};

/* Binomial Coefficient */
int N_Choose_K (int degree_n,
                int index_k);

/* Curve Algorithms */
Vertex Bezier(  const Vertex* arr_control_points,
                int degree_n,
                double parameter_t);
Vertex DeCasteljau( const Vertex* arr_control_points,
                    int degree_n,
                    double parameter_t);

/**
 * Main
 */
 int main(int argc, char* argv[]) {
    /* Initialize control points as constant variable */
    const Vertex P0(0.0, 0.0, 0.0);
    const Vertex P1(0.0, 3.0, 0.0);
    const Vertex P2(1.0, 3.0, 0.0);
    const Vertex P3(1.0, 0.0, 0.0);
    /* Insert control points to array */
    const Vertex arr_control_points_list[DEGREE+1] = {P0, P1, P2, P3};

    std::cout << "Degree of curve: " << DEGREE
                                     << std::endl;
    std::cout << "Control points: "  << P0 << ", "
                                     << P1 << ", "
                                     << P2 << ", "
                                     << P3
                                     << std::endl;
    std::cout << "Parameter: "       << PARAMETER
                                     << std::endl;

    /* Calculate vertex on Bezier curve */
    Vertex curve = Bezier(arr_control_points_list, DEGREE, PARAMETER);
    std::cout << "Bezier curve: " << curve << std::endl;
    /* Calculate vertex by using de Casteljau algorithm */
    curve = DeCasteljau(arr_control_points_list, DEGREE, PARAMETER);
    std::cout << "de Casteljau: " << curve << std::endl;

    return 0;
} /* main */

/**
 * Function:    Bezier
 * Purpose:     Calculation of coordinates on Bezier curve
 *              w.r.t. the value of t.
 * In args:     arr_control_points  (list of control points)
 *              degree_n            (degree of curve)
 *              parameter_t         (the value of parameter)
 * Out arg:     trace_on_curve      (vertex on Bezier curve)
 */
Vertex Bezier(  const Vertex* arr_control_points,
```

```
                int degree_n,
                double parameter_t) {
    Vertex trace_on_curve(0.0, 0.0, 0.0);

    /* Check parameter's errors */
    if (parameter_t < 0 || parameter_t > 1) {
        std::cerr   << "ERROR::FUNC::Bezier: "
                    << "Variable parameter_t must be the value from 0 to 1."
                    << std::endl;
    }
    /* Check degree's errors */
    if (degree_n < 2) {
        std::cerr   << "ERROR::FUNC::Bezier: "
                    << "Control points must be more than or equal to 3 for Bezier curve."
                    << std::endl;
    }
    /* Calculate coordinates of Bezier curve */
    else {
        for (int k = 0; k <= degree_n; k++) {
            /* Calculate Bernstein basis function */
            double bernstein =   N_Choose_K(degree_n, k)
                               * std::pow(1 - parameter_t  , degree_n - k)
                               * std::pow(parameter_t      , k);
            /* Multiply control point and berstein basis */
            trace_on_curve.m_x += arr_control_points[k].m_x * bernstein;
            trace_on_curve.m_y += arr_control_points[k].m_y * bernstein;
            trace_on_curve.m_z += arr_control_points[k].m_z * bernstein;
        }
    }

    return trace_on_curve;
} /* Bezier */

/**
 * Function:    N_Choose_K
 * Purpose:     Calculation of binomial coefficient
 * In args:     degree_n            (degree of binomial)
 *              index_k             ('k'th picking value)
 * Out arg:     result              (binomial coefficient)
 */
int N_Choose_K (int degree_n,
                int index_k) {
    int result, denominator = 1, numerator = 1;

    /* Check degree's errors */
    if (degree_n < 0) {
        std::cerr   << "ERROR::FUNC::N_Choose_K: "
                    << "Variable degree_n must be greater than 0"
                    << std::endl;
        result = 0;
    }
    /* Check restriction */
    if (degree_n < index_k) {
        std::cerr   << "ERROR::FUNC::N_Choose_K: "
                    << "Variable degree_n must be greater than index_k."
                    << std::endl;
        result = 0;
    }
    /* Calculate N choose K */
    else {
        if (degree_n == 0 || index_k == 0) {
            result = 1;
        }
        else if (index_k == 1) {
            result = degree_n;
```

```
        }
        /* Compare which of two values, N and K, is greater */
        else if (degree_n > index_k<<1) {
            /* n*(n-1)*...*(n-k+1) / k*(k-1)*...*1 */
            for (int i = 1; i <= index_k; i++) {
                denominator *= i;
                numerator *= degree_n - (i-1);
            }
            result = numerator / denominator;
        }
        else {
            /* n*(n-1)*...*(k+1) / (n-k)*(n-k-1)*...*1 */
            for (int i = 1; i <= degree_n - index_k; i++) {
                denominator *= i;
                numerator *= degree_n - (i-1);
            }
            result = numerator / denominator;
        }
    }

    return result;
} /* N_Choose_K */

/**
 * Function:    DeCasteljau
 * Purpose:     Calculation of de Casteljau algorithm
 * In args:     arr_control_points  (list of control points)
 *              degree_n            (degree of curve)
 *              parameter_t         (the value of parameter)
 * Out arg:     trace_on_curve      (vertex on Bezier curve)
 */
Vertex DeCasteljau( const Vertex* arr_control_points,
                    int degree_n,
                    double parameter_t) {
    Vertex trace_on_curve(0.0, 0.0, 0.0);

    /* Check parameter's errors */
    if (parameter_t < 0 || parameter_t > 1) {
        std::cerr   << "ERROR::FUNC::DeCasteljau: "
                    << "Variable parameter_t must be the value from 0 to 1."
                    << std::endl;
    }
    /* Check degree's errors */
    if (degree_n < 2) {
        std::cerr   << "ERROR::FUNC::DeCasteljau: "
                    << "Control points must be more than or equal to 3 for Bezier curve."
                    << std::endl;
    }
    /* Calculate coordinates of Bezier curve by using de Casteljau algorithm */
    else {
        /* Initialize list of points for curve fitting */
        Vertex* arr_curve_fitting_points = new Vertex[degree_n + 1];
        for (size_t i = 0; i <= degree_n; i++) {
            arr_curve_fitting_points[i] = arr_control_points[i];
        }
        /* 'k'th curve fitting point at step 'r' =
         *                      (1 - t) * ('k-1'th curve fitting point at step 'r-1')
         *                          +t  * ('k'th curve fitting point at step 'r-1')
        */
        for (int r = 1; r <= degree_n; r++) {
            for (int k = 1; k <= degree_n - r + 1; k++) {
                arr_curve_fitting_points[k - 1].m_x
                        =   (1 - parameter_t) * arr_curve_fitting_points[k - 1].m_x
                            + parameter_t     * arr_curve_fitting_points[k].m_x;
                arr_curve_fitting_points[k - 1].m_y
```

```
                        =   (1 - parameter_t) * arr_curve_fitting_points[k - 1].m_y
                            + parameter_t     * arr_curve_fitting_points[k].m_y;
                arr_curve_fitting_points[k - 1].m_z
                        =   (1 - parameter_t) * arr_curve_fitting_points[k - 1].m_z
                            + parameter_t     * arr_curve_fitting_points[k].m_z;
            }
        }

        trace_on_curve = arr_curve_fitting_points[0];
        delete[] arr_curve_fitting_points;
    }

    return trace_on_curve;
} /* DeCasteljau */
```

```
RESULTS:

Degree of curve: 3
Control points: (0, 0, 0), (0, 3, 0), (1, 3, 0), (1, 0, 0)
Parameter: 0.5
Bezier curve: (0.5, 2.25, 0)
de Casteljau: (0.5, 2.25, 0)

Process returned 0 (0x0)   execution time : 0.005 s
Press ENTER to continue.
```

2. In Bezier curves, explain the end tangent interpolation.

곡선의 양 끝점(첫 점과 끝 점)에서의 기울기는 control points가 만들어내는 볼록 다각형의 양 끝 점에서의 기울기와 반드시 같아야 한다는 것이다. 따라서 end point interpolation을 동시에 생각한다면, 곡선은 항상 양 끝점에서 폴리곤에 접한다.

$$\overrightarrow{r}(t) = \sum_{k=0}^{n} \overrightarrow{P_k} b_k^n(t)$$

$$\frac{d}{dt}\overrightarrow{r}(t) = \sum_{k=0}^{n-1} (\overrightarrow{P}_{k+1} - \overrightarrow{P}_k) n b_k^{n-1}(t)$$

$$\text{subject to } t = 0, t = 1$$

$$\left.\frac{d}{dt}\overrightarrow{r}(t)\right|_{t=0} = n(\overrightarrow{P}_1 - \overrightarrow{P}_0)$$
$$\left.\frac{d}{dt}\overrightarrow{r}(t)\right|_{t=1} = n(\overrightarrow{P}_n - \overrightarrow{P}_{n-1})$$

위의 식을 통해서, 첫 점($t = 0$)과 끝 점($t = 1$)에서 곡선의 기울기는 폴리곤의 벡터 방향과 동일하다는 것을 증명하였다.

3. Explain the convex hull property.

convex hull property를 이해하기 위해서는 convex combination을 알고 있어야 한다.

먼저, 결합의 종류 중에 각 항의 계수가 양수이면서 계수의 합이 1인 결합을 convex combination이라 정의한다.

$$a_0\phi_0 + a_1\phi_1 + a_2\phi_2 + \ldots + a_n\phi_n$$
$$\forall k : a_k \geq 0 \ \wedge \ \sum_{k=0}^{n} a_k = 1$$

만약 ϕ를 어떤한 위치라고 가정하고 식의 의미를 살펴보면,
ϕ들로 이루어지는 어떠한 공간이 있고, ϕ를 이용해 convex combination으로 표현되는 공간 상의 값은 각각의 ϕ를 직선으로 잇는 경계와 그 내부를 표현하게 된다. 즉, convex combination으로 표현된 공간은 ϕ들이 이루는 경계를 넘어서지 않는다.

그렇다면 Bernstein basis function ($b_k^n(t)$)과 control point($\overrightarrow{P_k}$)로 표현되는 Bezier curve는 convex combination이기 때문에 Bezier curve는 각각의 control points를 이어서 만들 수 있는 최소 볼록 다각형 안에 존재하게 된다.

이것이 Bezier curve가 갖는 convex hull property다.

4. Why do we need the rational form?

분수 형식이 아닌 Bezier 곡선으로는 원뿔 곡선을 표현할 수 없다. 2차원 원뿔 곡선을 표현하기 위해서는 2차원 Bezier 곡선을 평면에 투영하는 방식(perspective projection)을 사용하면 된다. 이를 통해 나타나는 식이 분수 형식이기 때문에, 우리는 Bezier 곡선으로 원뿔 곡선을 표현하기 위해서 분수 형식을 사용한다.