```
public class MonteCarloNCubeBase1TestPart1 {
	static Random r = new Random();

	public static void main(String[] args) {
//		System.out.println("Monte Carlo Integration:");
//		for (int i = 1; i < 10; i++) {
//			System.out.println("d=" + i + " answer:" + Answer.answer(i));
//			int[] res = new int[10];
//			for (int j = 0; j < 10; j++) {
//				System.out.println("no." + j + " test:");
//				res[j] = MonteCarloIntegration(i);
//			}
//			Arrays.sort(res);
//			System.out.println("Conclusion sample count: 4*10^" + res[0]);
//		}

		System.out.println("Cube based Integration:");
		for (int i = 1; i < 4; i++) {
			System.out.println("d=" + i + " answer:" + Answer.answer(i));
			CubebasedIntegration(i);
		}
	}

	private static int MonteCarloIntegration(int d) {
		double ans = Answer.answer(d);
		long sampleCount, start = 0, end = 0;
		double res = 0.0;
		for (sampleCount = 4000; Math.abs(res - ans) > 0.001; sampleCount *= 10) {
			int threadnum = 16;
			long[] temp = new long[threadnum];
			double[] halftemp = new double[threadnum];
			MonteCarloMultiHelper[] ths = new
MonteCarloMultiHelper[threadnum];
			start = System.currentTimeMillis();
			for (int i = 0; i < threadnum; i++) {
				ths[i] = new MonteCarloMultiHelper(i, temp, d, sampleCount /
threadnum, halftemp);
				ths[i].start();
			}
			for (MonteCarloMultiHelper th : ths) {
				try {
					th.join();
				} catch (InterruptedException e) {
					// TODO Auto-generated catch block
```

```
                    e.printStackTrace();
                }
            }
            end = System.currentTimeMillis();
            long count = 0;
            double half = 0;
            for (long i : temp) {
                count += i;
            }
            for (double i : halftemp) {
                half += i;
            }
            res = ((double) count + half / 2) / (double) sampleCount * Math.pow(2,
d);
        }

        int zero = 0;
        while (sampleCount > 4) {
            sampleCount /= 10;
            zero++;
        }

        System.out.println("Time: " + (end - start) + "ms sampleCount: 4*10^" + (zero -
1));
        return zero - 1;
    }

    private static void CubebasedIntegration(int d) {
        double ans = Answer.answer(d);
        long sampleCount, start = 0, end = 0;
        double res = 0.0;
        for (sampleCount = 64; Math.abs(res - ans) > 0.001; sampleCount *= 2) {
            start = System.currentTimeMillis();
            long[] count = new long[1];
            double[] half = new double[1];
            helper(0, d, count, sampleCount, half);
            end = System.currentTimeMillis();
            res = ((double) count[0] + half[0] / 2) / (double) Math.pow(sampleCount,
d) * Math.pow(2, d);
            System.out.println(sampleCount);
        }
        System.out.println(
                "CubebasedIntegration" + "d: " + d + " Time: " + (end - start) + "ms
sampleCount: " + sampleCount / 2);
```

```
    }

    private static void helper(double sum, int remain, long[] count, long sampleCount,
double[] half) {
        if (sum > 1)
            return;
        if (remain == 0) {
            if (sum < 1) {
                count[0]++;
            }
            if (sum == 1) {
                half[0]++;
            }
            return;
        }
        double unit = (double) 1.0 / sampleCount;
        for (int j = 0; j < sampleCount; j++) {
            double temp = j * unit;
            helper(sum + temp * temp, remain - 1, count, sampleCount, half);
        }
    }

}
```

```
public class MonteCarloNCubeBase1TestPart2 {
    static Random r = new Random();

    public static void main(String[] args) {
        for (int i = 2; i < 40; i++) {
            System.out.println("d=" + i);
            double res1 = MonteCarloIntegration(i);
            double res2 = CubebasedIntegration(i);
            double answer = Answer.answer(i);
            double rerror1 = Math.abs(res1 - answer) / answer;
            double rerror2 = Math.abs(res2 - answer) / answer;
            double absolutediff = res1 - res2;
            System.out.println(
                    "absolute diff:" + Math.abs(absolutediff) + "  relative diff：
" + Math.abs(absolutediff / answer));
            System.out.println("Monte Carlo Integration relative error:" +
Math.abs(rerror1));
            System.out.println("Cube based Integration relative error:" +
Math.abs(rerror2));
            System.out.println("Res 1:" + res1);
            System.out.println("answer:" + answer);
            System.out.println();
        }
    }

    private static double MonteCarloIntegration(int d) {
        long sampleCount = 1000000;

        int threadnum = 16;
        long[] temp = new long[threadnum];
        double[] halftemp = new double[threadnum];
        MonteCarloMultiHelper[] ths = new MonteCarloMultiHelper[threadnum];
        for (int i = 0; i < threadnum; i++) {
            ths[i] = new MonteCarloMultiHelper(i, temp, d, sampleCount /
threadnum, halftemp);
            ths[i].start();
        }
        for (MonteCarloMultiHelper th : ths) {
            try {
                th.join();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        long count = 0;
        for (long i : temp) {
            count += i;
        }
        double half = 0;
        for (double i : halftemp) {
            half += i;
        }
        System.out.println("count: " + count);

        double res = ((double) count + half / 2) / (double) sampleCount * Math.pow(2,
d);
        return res;
    }

    private static double CubebasedIntegration(int d) {
        int sampleCount = (int) Math.round(Math.pow(1000000, (double) 1.0 / (double)
d));
        int sampleCountError = (int) Math.abs(1000000 - Math.pow(sampleCount, d));
        System.out.println("Cube based Integration sample count:" +
Math.pow(sampleCount, d) + " sample count error:"
                + sampleCountError);
        long[] count = new long[1];
        double[] half = new double[1];
        helper(0, d, count, sampleCount, half);
        double res = ((double) count[0] + half[0] / 2) / (double) Math.pow(sampleCount,
d) * Math.pow(2, d);
        return res;
    }

    private static void helper(double sum, int remain, long[] count, long sampleCount,
double[] half) {
        if (sum > 1)
            return;
        if (remain == 0) {
            if (sum < 1) {
                count[0]++;
            }
            if (sum == 1) {
                half[0]++;
            }
            return;
        }
        double unit = (double) 1.0 / sampleCount;
        for (int j = 0; j < sampleCount; j++) {
            double temp = j * unit;
            helper(sum + temp * temp, remain - 1, count, sampleCount, half);
        }
    }

}
```

```java
import java.util.Random;

import Helper.MonteCarloMultiHelper;

public class MonteCarloNCubeBase1TestPart3 {
    static Random r = new Random();

    public static void main(String[] args) {
        System.out.println("For example, N=1,000,000");
        double answer = 3.1415926535897932;
        for (int i = 2; i < 20; i++) {
            System.out.println();
            double res = pi(i, MonteCarloIntegration(i));
            double error = Math.abs(answer - res);
            System.out.println("d=" + i + " res=" + res + " error=" + error);
        }
    }

    private static double MonteCarloIntegration(int d) {
        long sampleCount = 1000000;

        int threadnum = 16;
        long[] temp = new long[threadnum];
        double[] halftemp = new double[threadnum];
        MonteCarloMultiHelper[] ths = new MonteCarloMultiHelper[threadnum];
        for (int i = 0; i < threadnum; i++) {
            ths[i] = new MonteCarloMultiHelper(i, temp, d, sampleCount /
threadnum, halftemp);
            ths[i].start();
        }
        for (MonteCarloMultiHelper th : ths) {
            try {
                th.join();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        long count = 0;
        for (long i : temp) {
            count += i;
        }
        double half = 0;
        for (double i : halftemp) {
                half += i;
            }
            System.out.println("count of points in the hypersphere: " + count);

            double res = ((double) count + half / 2) / (double) sampleCount * Math.pow(2, d);

            return res;
        }

        private static double pi(int d, double res) {
            int divisor = 1;
            if (d % 2 == 1) {
                res /= Math.pow(2, d / 2 + 1);
                for (int i = 1; i <= d; i += 2) {
                    divisor *= i;
                }
            } else {
                for (int i = 2; i <= d / 2; i++) {
                    divisor *= i;
                }
            }
            res *= (double) divisor;

            return Math.pow(res, (double) 1.0 / (double) (d / 2));
        }
}
```

```
class CubeType {
    public static final int UNKNOW = 0;
    public static final int INSIDE = 1;
    public static final int OUTSIDE = 2;
    public static final int EDGE = 3;
}

class CubeBaseArr {
    public static int generateNum;
    public static int dim;
    double[] arr;

    /**
     * 
     * @param len
     */
    CubeBaseArr(int dimension) {
        dim = dimension;
        arr = new double[dimension];
        for (int i = 0; i < arr.length; i++) {
            arr[i] = 0.5;
        }
        generateNum = (int)Math.pow(2, dimension);
    }

    protected CubeBaseArr(double arr[]) {
        this.arr = new double[arr.length];
        for (int i = 0; i < arr.length; i++) {
            this.arr[i] = arr[i];
        }
    }

    /**
     * give a grain to decide if arr inside the hypersphere
     * 
     * @param grain
     * @return
     */
    public int isInside(double grain) {
        double highsum = 0.0;
        double lowsum = 0.0;
        for (int i = 0; i < arr.length; i++) {
            double temp1 = (arr[i] + grain);
            highsum += temp1 * temp1;
            double temp2 = (arr[i] - grain);
            lowsum += temp2 * temp2;
        }
        if (highsum < 1.0) {
            return CubeType.INSIDE;
        } else if (lowsum > 1.0) {
            return CubeType.OUTSIDE;
        } else {
            return CubeType.EDGE;
        }
    }

    public LinkedList<CubeBaseArr> generate(double grain) {
        LinkedList<CubeBaseArr> cbas = new LinkedList<CubeBaseArr>();
        for (int i = 0; i < generateNum; i++) {
            CubeBaseArr cur = new CubeBaseArr(arr);
            for (int d = 0, shift = dim - 1; d < dim; d++, shift--) {// dimension
                if ((i & (1 << shift)) == 0) {
                    cur.arr[d] -= grain;
                } else {
                    cur.arr[d] += grain;
                }
            }
            cbas.add(cur);
        }
        return cbas;
    }
}

class CubeBaseProxy {
    public double sampleCount;
    private int dimension;
    private double curGrain;
    private double lastGrain;
    private double curVolume;
    private double curWeight;
    private LinkedList<CubeBaseArr> cbas;

    CubeBaseProxy(int dimension) {
        this.dimension = dimension;
        lastGrain = 1;
        curGrain = 0.5;
        cbas = new LinkedList<CubeBaseArr>();
        CubeBaseArr cba = new CubeBaseArr(dimension);
```

```
		cbas.add(cba);
		curWeight = 1.0;
		sampleCount = 0.0;
	}

	/**
	 * for old one
	 * 
	 * @return
	 */
	public double pushFoward() {
		int size = cbas.size();
		lastGrain /= 2;
		curGrain /= 2;
		long count = 0;
		for (int i = 0; i < size; i++) {
			CubeBaseArr cur = cbas.removeFirst();
			sampleCount++;
			int temp = cur.isInside(lastGrain);
			if (temp == CubeType.EDGE) {
				cbas.addAll(cur.generate(curGrain));
			} else if (temp == CubeType.INSIDE) {
				count++;
			}
		}
		curVolume += curWeight * count;
		curWeight /= Math.pow(2, dimension);
		return curVolume;
	}
}

/**
 * 
 * @author wangbicheng
 * Cube-Base Test
 */
public class CubeBase2Test {

	public static void main(String[] args) {
		long start = System.currentTimeMillis();

		int dimension = 3;
		int grainLevel = 12; // the sample number is about (2 ^ dimension) ^ grainLevel
		double estimateVolume = Math.pow(2, dimension);
```

```
        CubeBaseProxy cbp = new CubeBaseProxy(dimension);

        for(int i = 0; i < grainLevel - 1; i++) {
            cbp.pushFoward();
        }
        estimateVolume *= cbp.pushFoward();

        long end = System.currentTimeMillis();
        System.out.println("estimate sample: " + Math.pow(Math.pow(2, dimension),
grainLevel));
        System.out.println("actually sample:" + cbp.sampleCount);
        System.out.println("standard volume:" + Answer.answer(dimension));
        System.out.println("estimate volume:" + estimateVolume);
        System.out.println("cost time:" + (end - start) / 1000 + "s");

        return;
    }
}
```

```
/**
 * wtf because we can only submit file by file and limit to 10...
 * @author wangbicheng
 *
 */
class CubeBaseMultiHelper extends Thread {

    public double[] ds;
    public double[] newds;
    public int len;
    public int end;
    public int startIndex;
    public int total;
    public int ip;

    public CubeBaseMultiHelper() {}
    public CubeBaseMultiHelper(int i, int startIndex, int len, double[] ds) {
        this.ip = i;
        this.startIndex = startIndex;
        this.len = len;
        this.end = Math.min(ds.length, startIndex + len);
        this.ds = ds;
        this.newds = new double[ds.length];
        this.total = ds.length;
    }

    @Override
    public void run() {
        for (int i = startIndex; i < end; i++) {
            double tempX = ds[i];
            if(tempX == 0) continue;
            for (int j = 0; j < this.total - i; j++) {
                newds[i + j] += tempX * ds[j];
            }
        }
    }

}
class CubeBaseModArr {
    private int zoom;
    public double[] arr;

    public CubeBaseModArr(int zoom) {
        super();
        this.zoom = zoom;
        this.arr = new double[zoom];
    }

    /**
     * value: 0 ~ ZOOM
     *
     * @param value
     */
    public void insertValue(int value, double freq) {
        this.arr[value] += freq;
    }

    public double sumFreq() {
        double sumFreq = 0.0;
        for (int i = 0; i < zoom; i++) {
            sumFreq += arr[i];
        }
        return sumFreq;
    }
}

class ModifiedMethodProxy {
    protected int THREAD_MAX;
    protected int basicZoom;
    protected int sampleTime;
    protected int dimension;
    protected CubeBaseModArr mca;
    protected CubeBaseModArr newMca;
    protected double[] freq;

    public ModifiedMethodProxy(int dimension, int zoom, int threadNum) {
        this.basicZoom = zoom;
        this.dimension = dimension;
        this.THREAD_MAX = threadNum;
        mca = new CubeBaseModArr(zoom);
        initSample();
    }

    public ModifiedMethodProxy(int dimension, int zoom, int sampleTime, int threadNum)
{
        this.basicZoom = zoom;
        this.sampleTime = sampleTime;
        this.dimension = dimension;
        this.THREAD_MAX = threadNum;
        mca = new CubeBaseModArr(zoom);
        randomInitSample();
    }

    public void insertSample(double value) {
        value = value * value * basicZoom;
        mca.insertValue((int) value, 1);
    }

    public void initSample() {
        for (long i = 0; i < basicZoom; i++) {
            mca.arr[(int)Math.round((double)i * i / basicZoom)] += 1.0 / basicZoom;
        }
    }

    public void randomInitSample() {
        Random r = new Random();
        int time = basicZoom * sampleTime;
        for (long i = 0; i < time; i++) {
            double temp = r.nextDouble();
            mca.arr[(int)Math.floor(temp * temp * basicZoom)] += 1.0 / time;
        }
    }

    private void generate() {
        newMca = new CubeBaseModArr(this.basicZoom);
        for (int i = 0; i < basicZoom; i++) {
            double tempX = mca.arr[i];
            for (int j = 0; j < basicZoom - i; j++) {
                newMca.insertValue(i + j, tempX * mca.arr[j]);
            }
        }
        mca = newMca;
    }

    public double sumFreq() {
        int level = dimension;
        while (level > 2) {
            long start = System.currentTimeMillis();
            mutliGenerate();
            level /= 2;
            long end = System.currentTimeMillis();
        }
```

```java
        long start = System.currentTimeMillis();
        doubleDimension();
        long end = System.currentTimeMillis();
        double res = newMca.sumFreq();
        return res;
    }

    private void mutliGenerate() {
        newMca = new CubeBaseModArr(this.basicZoom);

        CubeBaseMultiHelper[] ths = new CubeBaseMultiHelper[THREAD_MAX];

        int len = this.basicZoom / THREAD_MAX + 1;
        for (int i = 0; i < THREAD_MAX; i ++) {
            ths[i] = new CubeBaseMultiHelper(i, i * len, len, mca.arr);
            ths[i].start();
        }
        try {
            for (int i = 0; i < THREAD_MAX; i++) {
                ths[i].join();
            }

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        for (int i = 0; i < this.basicZoom; i++) {
            int threadEnd = i / len + 1;
            for (int j = 0; j < threadEnd; j++) {
                newMca.arr[i] += ths[j].newds[i];
            }
        }
        mca = newMca;
    }

    /**
     * can double the dimension according to lower dimension distribution
     */
    private void doubleDimension() {
        this.freq = new double[this.basicZoom];
        double freqSumD8 = 0;
        for (int i = 0; i < this.basicZoom; i++) {
            freqSumD8 += mca.arr[i];
            this.freq[i] = freqSumD8;
        }
        for (int i = 0; i < this.basicZoom; i++) {
            mca.arr[i] *= this.freq[this.basicZoom - i - 1];
        }
        newMca = mca;
    }

    public double volume() {
        double temp = sumFreq();
        temp = temp * Math.pow(2.0, dimension);
        return temp;
    }
}


public class ModifiedTest {

    public static void main(String[] args) {
        int dimension = 16; // 8
        /**
         * please according the dimension change the gap
         */
        int gap = (int) Math.pow(2, 16); // gap means divide 1 by 2^gap;
        int sampleTime = (int) Math.pow(2, 10); // sampe number = sampleTime * 2 ^
gap
        int thread = 32; // 32 best
        /**
         * this is for monte carlo
         */
        // monteCarloTest(dimension, gap, sampleTime, thread);
        /**
         * this is for cube base
         */
        cubeBasedTest(dimension, gap, thread);
    }

    private static double monteCarloTest(int d, int g, int sampleTime, int t) {
        int time = 100;
        int count = 0;
        double result = 0;
        double errAvg = 0;
        for (int i = 0; i < time; i++) {
            double err = Math.abs(testStart(d, g, sampleTime, t, true));
            errAvg += err;
            if (err < 0.0005) {
```

```
                        count++;
                    }
                }
                errAvg /= time;
                result = count;
                result /= time;
                System.out.println("percent: " + result + " err avg: " + errAvg);
                return result;
        }

        private static double cubeBasedTest(int d, int g, int t) {
                return testStart(d, g, 1, t, false);
        }

        /**
         * return is in the 4 percision region
         *
         * @param dimension
         * @param gap
         * @param thread
         * @param openMonteCarlo
         * @return
         */
        private static double testStart(int dimension, int gap, int sampleTime, int thread,
boolean openMonteCarlo) {
                long start = System.currentTimeMillis();

                int sampleNum = gap;
                if (openMonteCarlo) {
                    sampleNum = sampleTime * gap;
                }

                double mR = 0.0;
                ModifiedMethodProxy mcp;
                if (openMonteCarlo) {
                    mcp = new ModifiedMethodProxy(dimension, gap, sampleTime, thread);
                } else {
                    mcp = new ModifiedMethodProxy(dimension, gap, thread);
                }

                mR = mcp.volume();
                double sR = Answer.answer(dimension);

                if (!openMonteCarlo) {
```

```
            System.out.println("sample number: " + sampleNum);
            System.out.println("Estimate Value: " + mR);
            System.out.println("Standard Value: " + sR);
        }

        long end = System.currentTimeMillis();
        long time = (end - start);
        if (time < 10000) {
            System.out.println("Time: " + (time) + "ms");
        } else {
            System.out.println("Time: " + (time / 1000) + "s");
        }

        return sR - mR;
    }

}
```

```java
public class Answer {

    public static double answer(int d) {
        double res = 1.0;
        for (int i = 0; i < d / 2; i++) {
            res *= 3.141592653589793;
        }

        int divisor = 1;
        if (d % 2 == 1) {
            res *= Math.pow(2, d / 2 + 1);
            for (int i = 1; i <= d; i += 2) {
                divisor *= i;
            }
        } else {
            for (int i = 2; i <= d / 2; i++) {
                divisor *= i;
            }
        }

        res /= (double) divisor;
        return res;
    }
}
```

```java
public class FactorialHelper {
    // attention overflow
    public final static int Fsize = 18;
    public static int[] F = new int[Fsize];
    public static int[][] C = new int[Fsize][Fsize];
    static {
        F[1] = 1;

        for (int i = 2; i < Fsize; i++) {
            F[i] = F[i - 1] * i;
            C[i][0] = 1;
            C[i][i] = 1;
        }

        for (int i = 0; i < Fsize; i++) {
            for (int j = 1; j < i; j++) {
                C[i][j] = F[i] / F[j] / F[i - j];
            }
        }
    }

}
```

```
public class MonteCarloMultiHelper extends Thread {

    private int ip;
    private long[] temp;
    private int d;
    private long sampleCount;
    private Random r;
    private double[] halftemp;

    public MonteCarloMultiHelper(int i, long[] temp, int d, long sampleCount, double[]
halftemp) {
        this.ip = i;
        this.temp = temp;
        this.d = d;
        this.sampleCount = sampleCount;
        r = new Random();
        this.halftemp = halftemp;
    }

    @Override
    public void run() {
        long count = 0;
        double half = 0;
        for (int j = 0; j < sampleCount; j++) {
            double sum = 0;
            for (int i = 0; i < d && sum <= 1; i++) {
                double randomValue = r.nextDouble();
                sum += randomValue * randomValue;
                if (sum > 1) {
                    break;
                }
            }
            if (sum < 1) {
                count++;
            }
            if (sum == 1) {
                half++;
            }
        }
        temp[ip] = count;
        halftemp[ip] = half;
    }
}
```