# Homework3

## CSI3108-1 Algorithms

27167033 남다예

# 1. 알고리즘

## 클래스

| | |
|---|---|
| FFT | FFT를 수행한다. |
| FFT | 점들의 좌표들을 저장하는 클래스. x, y좌표를 저장하고, count 안에는 몇 개의 collinear라인이 지나는지를 저장한다.<br><br>`FFT(ArrayList<complex> cl, complex c){`<br>`  this.n = cl.size();`<br>`  this.complex = c;`<br>`  this.complex_list = cl;`<br>`}`<br><br>coefficient들을 arraylist안에 저장하고, w는 complex로 저장한다.<br>처음에 coefficient들의 갯수를 구하여 2의 제곱승이 되지 않으면 0을 추가하여 2의 제곱승으로 만들어준다. 이 갯수를 통해 w를 구한다. |
| functions | `public void doFFT(){`<br>`  if(this.n == 1)`<br>`  return;`<br><br>`  ArrayList<complex> even_list = new ArrayList<complex>();`<br>`  ArrayList<complex> odd_list = new ArrayList<complex>();`<br><br>`  for(int i=0; i<n/2; i++){`<br>`  even_list.add(this.complex_list.get(i*2));`<br>`  odd_list.add(this.complex_list.get(i*2+1));`<br>`  }`<br><br>`  FFT even_FFT = new FFT(even_list,`<br>`complex.multiplyComplex(this.complex, this.complex));`<br>`  FFT odd_FFT = new FFT(odd_list,`<br>`complex.multiplyComplex(this.complex, this.complex));`<br><br>`  even_FFT.doFFT();`<br>`  odd_FFT.doFFT();`<br><br>`  complex ww = new complex(1, 0);`<br>`  for(int i=0; i<n/2; i++){`<br>`  complex offset = complex.multiplyComplex(ww,`<br>`odd_FFT.complex_list.get(i));`<br>`  this.complex_list.set(i,`<br>`complex.addComplex(even_FFT.complex_list.get(i), offset));`<br>`  this.complex_list.set(i+n/2,`<br>`complex.subtractComplex(even_FFT.complex_list.get(i),offset));`<br>`  ww = complex.multiplyComplex(ww, this.complex);`<br>`  }`<br>`}`<br><br>coefficient의 리스트를 홀수차항과 짝수차항으로 나누어 재귀적으로 함수를 계속 시행한다.<br>함수가 시행될 때마다 coefficient_list에 들어있는 w이 바뀌면서 제일 처음에 call했던 doFFT에서 최종 결과를 리턴한다. |

| | |
|---|---|
| complex | 복소수 기본 단위와 연산 함수들이 포함되어 있는 클래스이다. |
| complex | 복소수의 실수와 허수 부분을 각각 double type의 real_part와 imaginary_part로 저장한다.<br><br>`private double real_part;`<br>`private double imaginary_part;`<br>`public complex(double rp, double ip){`<br>`this.real_part = rp;`<br>`this.imaginary_part = ip;`<br>`}` |
| functions | 직선들의 정보를 저장하는 클래스. 클래스 내의 함수로 두 점의 좌표를 넣으면 기울기와 x절편을 계산하여 저장한다.<br><br>`public String toString()`<br>복소수를 실수 + 허수i 형태로 출력한다<br>`public complex addComplex(complex c1, complex c2)`<br>복소수의 덧셈을 수행한다.<br>`public complex subtractComplex(complex c1, complex c2)`<br>복소수의 뺄셈을 수행한다.<br>`public complex multiplyComplex(complex c1, complex c2)`<br>복소수의 곱셈을 수행한다.<br>`public void roundUp(){`<br>`this.setRP(Math.round(this.getRP()*1000000d)/1000000d);`<br>`this.setIP(Math.round(this.getIP()*1000000d)/1000000d);`<br>`}`<br>복소수를 소수 6번째자리까지 표기되도록 반올림한다. |

## ㄴMAIN ALGORITHM

1. 주어진 인풋을 차례로 읽으면서 FFT 클래스 안에 coefficients들을 저장한다.
2. 입력이 끝나면 FFT 클래스 안의 instructor가 주어진 coefficients의 개수를 보고 2의 제곱승인 수가 아니면 FFT를 시행하기 위해 0들을 더해 2의 제곱승개로 만든다. coefficients의 개수에 따라 아래의 식을 통해 초기 w를 정한다.

$$e^{2\pi i/n} = \cos\left(\frac{2\pi}{n}\right) + i \cdot \sin\left(\frac{2\pi}{n}\right),$$

3. doFFT는 재귀적으로 실행되는데 하나의 리스트를 홀수차항과 짝수차항으로 나누어 리스트를 만들고 각 리스트마다 다시 doFFT를 실행한다. 리스트에 coefficient가 하나만 존재할 때까지 함수를 실행하고, 각 함수가 반환하는 값들로 complex_list를 업데이트하면서 다시 가장 처음 call 했던 함수까지 실행하고 난 complex_list가 원하는 최종 output이다.

## 2. 예시 결과 및 실행시간

#1

6.000000 0.000000
-2.000001 -2.000000
-2.000000 0.000000
-1.999999 2.000000

#2

120.000000 0.000000
-8.000013 -40.218722
-8.000006 -19.313710
-8.000010 -11.972848
-8.000003 -8.000000
-8.000005 -5.345427
-8.000005 -3.313707
-8.000010 -1.591295
-8.000000 0.000000
-7.999999 1.591301
-7.999999 3.313710
-7.999999 5.345435
-7.999997 8.000000
-7.999993 11.972848
-7.999990 19.313707
-7.999969 40.218709

#3

496.000000 0.000000
-16.000053 -162.450755
-16.000026 -80.437444
-16.000040 -52.744950
-16.000013 -38.627421
-16.000022 -29.933896
-16.000021 -23.945696
-16.000037 -19.496068
-16.000005 -16.000000
-16.000010 -13.130856
-16.000009 -10.690854
-16.000017 -8.552172
-16.000010 -6.627413
-16.000015 -4.853539
-16.000021 -3.182590
-16.000046 -1.575851

-16.000000 0.000000
-15.999999 1.575867
-15.999999 3.182602
-16.000002 4.853558
-15.999998 6.627421
-15.999997 8.552188
-15.999998 10.690869
-16.000005 13.130887
-15.999995 16.000000
-15.999988 19.496061
-15.999987 23.945695
-15.999979 29.933910
-15.999980 38.627413
-15.999958 52.744930
-15.999939 80.437417
-15.999832 162.450685

| Test Case | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 평균 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 6 | 5 | 6 | 6 | 8 | 6 | 6 | 6 | 6 | 6 |
| 2 | 6 | 2 | 2 | 3 | 3 | 3 | 2 | 3 | 3 | 3 | 3.4 |
| 3 | 5 | 7 | 5 | 5 | 5 | 5 | 8 | 6 | 5 | 5 | 5.6 |

실행시간을 분석 해 본 결과, 테스트케이스 1의 경우가 평균 6.0 ms, 2의 경우 3.4 ms, 3의 경우가 5.6 ms이 걸리는 것을 확인할 수 있었다. 가장 짧은 테스트 케이스의 실행시간이 가장 높은 것이 논리적지 않게 느껴져 더 많은 테스트케이스들을 만들어 실행시켜 보았고, 첫번째 케이스 이후에는 input의 길이와 비례해 실행시간이 달라지는 것을 볼 수 있었다. 이로인해 첫번째 케이스에서는 초기에 필요한 것들 때문에 실행시간이 길어진다고 유추할 수 있다.