Coursework 2 Part A

COMS21103: Data Structures and Algorithms

Kristian Krastev
kk12742

Q1. Complex numbers

This question asked us to complete the complex abstract data type by adding minus and conjugate methods. These were rather straightforward being familliar with complex numbers and how they work: subtraction is analogical to the addition - works by subtracting real from real and imaginary from imaginary parts, and conjugate implements the unary operation by returning a complex number with an imaginary part of the oposite sign. I also implemented two auxillary methods “getReal” and “getImagine” returning the two respective structural components stored privately by the object. Further, I changed the “toString” method to comply with the formatting requirements for Question 2. These are all tested using print statements in the main method that can be invoked by running “java Cwk2Question1”, as desired. I should note here, that I chaged the original name of the java class “ComplexADT” to just “Complex”, for convenience, since it is used throughout the rest of the code as a data type.

Q2. The FFT

For the implementation of the Fast Fourier Transform, as well as for core functionalities used by solutions to questions 3 and 4, I created a class called "FFT". The FFT itself is implemented in a method in it, also named "FFT", that takes as input an array of floating digit numbers, representing the coefficients of a polynomial, and returns an array of Complex objects - its FFT. It resembles the pseudocode given in slides.

First, the base case of the recursion is checked for, in which case the single coefficient itself is returned. Then the general one is worked through: to begin with the angle between roots of unity **omega1** is calculated as the Complex $\cos 2\pi/N + i \sin 2\pi/N$, depending on their number N (using java.lang.Math for trigonometric funtions and constants), and **omega** is initialized to the first of them - $1 + 0i$. Next, the input array **A** is split in two: **A0** and **A1**, containing its odd and even elements respectively using a custom method **split**. The function then recursively calls itself on both of them - storing the results in two Complex solution-vectors **y0** and **y1** each of size $N/2$. Upon returning from the recursions the two are reassembled into the one to be outputed by the current call - **y** of size N. This is done in N/2 steps given the halving property of the complex roots of unity. Finally **y** is returned.

The running time is $\Theta(nlogn)$ as we split the problem in two subproblems of half the size recursively and have $\Theta(n)$ work at each level where n is the size of the problem at that level (Case 2 of the Master Theorem).

FFT.java: the FFT method

```
public static Complex[] FFT (double[] A)
{
    int N = A.length;

    //Base case for the recursion
    if (N==1)
    {
        Complex[] y = new Complex[1];
        y[0] = new Complex(A[0], 0);
        return y;
    }
    Complex omega1 = new Complex(Math.cos(2*(Math.PI)/N), Math.sin(2*(Math.PI)/N));
    Complex omega = new Complex(1,0);

    double[] A0 = new double[A.length/2];
    double[] A1 = new double[A.length/2];
    split(A, A0, A1);

    Complex[] y0 = FFT(A0);
    Complex[] y1 = FFT(A1);

    Complex[] y = new Complex[N];
    for (int k=0; k<N/2; k++)
    {
        y[k] = y0[k].plus(omega.times(y1[k]));
        y[k+N/2] = y0[k].minus(omega.times(y1[k]));
        omega = omega.times(omega1);
    }
    return y;
}
```

In "Cwk2Question2.java", I/O matters are taken care of and the solution to this question is wrapped and formated. It can be tested by running "java Cwk2Question2 filename" as required, assuming the input length is a power of two. I tested it using the example inputs provided as well as others and it all works as desired.

Q3. The IFFT

The Inverse Fast Fourier Transform is essentially the same as the FFT but works *the other way round*. For the purposes of this report I shall take for granted as *deus ex machina* the fact that if $y_k = \sum_{j=0}^{N-1} a_j e^{i2\pi jk/N}$ then $a_j = 1/N \sum_{k=0}^{N-1} y_k e^{-i2\pi jk/N}$. Hence we are left with the actual implementation:

FFT.java: the IFFT method

```
public static Complex[] IFFT (Complex[] Y)
{
    int N = Y.length;
    if (N==1)
    {
        return Y;
    }
    Complex omega1 = new Complex(Math.cos(2*(Math.PI)/N), Math.sin(2*(Math.PI)/N));
    Complex omega = new Complex(1,0);

    Complex[] Y0 = new Complex[Y.length/2];
    Complex[] Y1 = new Complex[Y.length/2];
    split(Y, Y0, Y1);

    Complex[] a0 = IFFT(Y0);
    Complex[] a1 = IFFT(Y1);

    Complex[] a = new Complex[N];
    for (int k=0; k<N/2; k++)
    {
        a[k] = a0[k].plus((omega.conjugate()).times(a1[k]));

        a[k+N/2] = a0[k].minus((omega.conjugate()).times(a1[k]));
        omega = omega.times(omega1);
    }
    return a;
}
```

As you can see here everything happens in very much the same way as in **FFT** with the difference that the roles of the two vectors is swapped. This is reflected by the namings - we have an input point-value representation vector **Y** and an output coefficient vecor **a** rather than the other way round, as with **A** and **y** in FFT. The method **split** is redefined on **Complex** for the purposes of the **IFFT**. The main difference is that when we are reassembling the solutions from the recursive calls we use the conjugate of the corresponding omega as we have $e^{-i2\pi jk/N}$ in the inverse formula, rather than $e^{i2\pi jk/N}$.

IFFT however does not give us the actuall IFFT because we still need to divide by N, as seen in the inverse formula. This not being part of the recursion is done in a separate method named **normalize** as it only needs to happen once after the out-most recursion has returned.

The running time is $\Theta(nlogn)$ analogically to the FFT.

Again I/O is implemented in the file "Cwk2Question3.java" according to the specified format. It can be tested by running "java Cwk2Question3 filename" after compiling.

Q4. Polynomial squaring

Below is the squaring procedure based on the FFT and IFFT that works as follows. First, the input coefficient vector **A** is copied to the new vector **polly2N** of size $2N$ - padded with 0s to fill the empty terms. Then we compute its FFT in the **Complex** vector **pollyFFT** and create another one **fftSQRD** where we store the point-wise squares of the FFT iterating through it in the for loop (the extra variable is used rather for clarity). Next, the IFFT of the point-wise squared FFT is obtained in **pollySQRD** and normalised (terms are devided over N). Finally a vector of floating point numbers **polly** of size $N-1$ is created(the highest posssible power for a term in the square), and in it are copied the real components of **pollySQRD**, rounded to to 6 places after the dcimal sign. The later is necessary because our actual answer obtained from the IFFT is only a very precise approximation to the real solution, there is slight precison loss due to the limitations of digital computation - once when the FFT is computed, again when the point-value fractions are squared and finally when the IFFT is computed. But with this ammendment the error is hidden and we get the desired output.

FFT.java: the squaring method

```
public static double[] square (double[] A)
{
    int N = A.length;
    double[] polly2N = new double[2*N];
    Arrays.fill(polly2N, 0);
    System.arraycopy(A, 0, polly2N, 0, N);

    Complex[] pollyFFT = FFT(polly2N);
    Complex[] fftSQRD = new Complex[2*N];

    for (int i=0; i<2*N; i++)
    {
        fftSQRD[i] = pollyFFT[i].times(pollyFFT[i]);
    }
    Complex[] pollySQRD = IFFT(fftSQRD);
    normalize(pollySQRD);

    double[] polly = new double[2*N-1];
    for (int i=0; i<2*N-1; i++)
    {
        polly[i] = pollySQRD[i].getReal() * Math.pow(10,6);
        polly[i] = Math.round(polly[i]) / Math.pow(10,6);
    }
    return polly;
}
```

The running time of this procedure is dominated by the FFT and IFFT as both of the loops are executed once and take linear time and all other operations take constant time. The total running time is hence $O(nlogn)$.

Once more I/O is implemented in the file "Cwk2Question4.java" according to the specified format. It can be tested by running "java Cwk2Question4 filename" after compiling.

Q5. Exact string matching

This question explores solving the *exact string matching* problem usig the FFT. We have our text $t = t_1, t_2, \ldots, t_n$ and pattern $p = p_1, p_2, \ldots, , p_m$ represented as ASCII values. And we know that p matches $t[i, \ldots, i + m - 1]$ iff $S_i = \sum_{j=1}^{m}(p_j - t_{i+j-1})^2 = 0$.

Multiplying out we get $S_i = \sum_{j=1}^{m} p_j^2 - \sum_{j=1}^{m} 2p_j t_{i+j-1} + \sum_{j=1}^{m} t_{i+j-1}^2$. Calculating the first and third sums takes linear time in m, the aim is to use the FFT to compute $\sum_{j=1}^{m} 2p_j t_{i+j-1}$ efficiently for each of the $n - m$ i-s we are interested in. I started by simply considering the product s of polynomials t and p and observing that the ith coefficient can be expressed as $s_i = \sum_{j=0}^{i} p_j t_{i-j}$, reasoning about polynomial multiplication in the traditional school-book way and noting how each power in the product is formed. It is therefore obvious that this is not too far from our desired sum - we just need to reareange some indices. In fact if we just do the multiplication it is as if we are checking if the pattern would match backwards.

The basic idea is:

1. reverse the order of the coefficients in t
2. multiply t and p using the FFT and IFFT
3. consider the coefficients in the product in reversed order

This is exactly what my implementation does: it reverses the text in the beginning when it is processing it from the input lines using the method **formatInput**. Then it does FFT multiplication on both the text and pattern storing it in **pXt** - for this part I wrote a new method for polynomial multiplication **multiply** in "FFT.java" - simillar to the squaring one. Next it stores the sum of squares of all coefficients in the patten **patSqrSum**, and of the ones in the first segment of the text to be matched - in **matSqrSum**. A loop follows, that iterates down through the product of the multiplication **pXt**, updating the sum of coefficients in the segment of the text to be matched **matSqrSum** and checking for a match by comparing the sum of the two sums of squares and twice the coeffiicient at thecurrent index in the product to zero. When a match occurs the loop breaks and it's correct position in the regular order of the text-coeffcients is output.

The **formatInput** and **formatPattern** procedures take $\Theta(n)$ and $\Theta(m)$ time respectively, the two loops for the sums of squares are also $\Theta(m)$ and the loop at the end runs in $O(n)$ time depending on where the first match ocurs. The sum of squares in the text-segment to bbe matched is updated in constant time. The most important line of code for the time complexity is thus the one where **FFT.multiply** is called. The running time of this procedure is dominated by the running time of the FFT and the IFFT, which both run in $\Theta(nlogn)$. Thus the total running time is $O(nlogn)$.

The string matcher can be tested by running "java Cwk2Question5 filename" after compiling. I tested it and it works as required. The code-excerpt follows.

Cwk2Question5.java: main method

```
        ... ... ...
    //read input file argument
    File inputFile = new File(args[0]);
    String[] lineIn = readFile(inputFile);

    int matLen = lineIn[0].length();
    int patLen = lineIn[1].length();

    //convert to reversed 0-padded array of ASCIIs of power-of-2 length
    double[] text = formatInput(lineIn[0]);
    int len = text.length;

    //convert to 0-padded array of ASCIIs of same length as input text
    double[] pattern = formatPattern(lineIn[1], len);

    double[] pXt = FFT.multiply(text, pattern);

    double patSqrSum = 0;
    for (int i=0; i<patLen; i++)
        patSqrSum += Math.pow(pattern[i], 2);

    double matSqrSum = 0;
    for (int j=1; j<=patLen; j++)
        matSqrSum += Math.pow(text[len-j],2);

    double[] matches = new double[len];
    for(int i=len-1; i>=len-matLen+patLen-1; i--)
    {
        if (i<len-1)
            matSqrSum = matSqrSum - Math.pow(text[i+1], 2)
                        + Math.pow(text[i-patLen+1], 2);

        matches[i] = patSqrSum + matSqrSum - 2*pXt[i];
        if (matches[i] == 0)
        {
            System.out.print(len-i + " ");
            break;
        }
    }
    System.out.println();
}
```

Q6. Exact string matching with don't cares

This question explores solving the *exact string matching with don't cares* problem in a simillar way we did for the *exact string matching* in the previous question.

Here we need to make sure that $S_i = \sum_{j=1}^{m} p_j t_{i+j-1}(p_j - t_{i+j-1})^2 = \sum_{j=1}^{m} p_j^3 t_{i+j-1} - \sum_{j=1}^{m} p_j^2 t_{i+j-1}^2 + \sum_{j=1}^{m} p_j t_{i+j-1}^3 = 0$. This has to do with the fact that we encode the "don't kow" characters - the "?"s as 0s. That is where the $p_j t_{i+j-1}$ comes from - as it will equal 0 if either of the characters being matched is the special character causing the whole expression at this instance of the sum to equal 0. In other words we allow for either a regular match or a "don't know" one to occur at each position.

The problem seems a bit more complicated but is essentially analogical to the one in the previous question. Using the idea about reversing the input string and performing the all necessary computations in the right order the solution takes shape as follows.

In "Cwk2Question6.java" the **formatInput** and **formatPattern** methods are updated to check for the occurrance of a "?" with ASCII value 63 and encode it as 0 in the coefficient vectors of both the text and the pattern. This time round I created four auxillary vectors - two for the squares, and two for the cubes of both the pattern and text coefficient vectors **patSqrs, patCubs, textSqrs, textCubs**. This is what the first two loops do. Then I call **FFT.multiply** three times to obtain all the instances of each of the three sums needed, at each indice of the three corresponding output vectors in reverse order just like last time. Then in the exact same way as in the previous question but with less complications I iterate down in reversed order, which is actually correct order to check for the first occurance. Works like magic.

Of course this runs in $O(nlogn)$ as the FFT and IFFT for obvious reasons analogical to the ones I described in the previous section on question 5. The program can be run by typing "java Cwk2Question6 yourtestfilename" after compiling. The code-excerpt follows.

Cwk2Question6.java: main method

```
    //read input file argument
    File inputFile = new File(args[0]);
    String[] lineIn = Cwk2Question5.readFile(inputFile);

    int matLen = lineIn[0].length();
    int patLen = lineIn[1].length();

    //convert to reversed 0-padded array of ASCIIs of power-of-2 length
    //replacing "?"s encoded as 63 with 0s
    double[] text = formatInput(lineIn[0]);
    int len = text.length;

    //convert to 0-padded array of ASCIIs of same length as input text
    //replacing "?"s encoded as 63 with 0s
    double[] pattern = formatPattern(lineIn[1], text.length);

    double[] patSqrs = new double[len];
    double[] patCubs = new double[len];
    Arrays.fill(patSqrs, 0);
    Arrays.fill(patCubs, 0);
    for (int i=0; i<patLen; i++)
    {
        patSqrs[i] = Math.pow(pattern[i], 2);
        patCubs[i] = Math.pow(pattern[i], 3);
    }

    double[] texSqrs = new double[len];
    double[] texCubs = new double[len];
    Arrays.fill(texSqrs, 0);
    Arrays.fill(texCubs, 0);
    for (int i=len-1; i>len-matLen-1; i--)
    {
        texSqrs[i] = Math.pow(text[i], 2);
        texCubs[i] = Math.pow(text[i], 3);
    }

    double[] p3Xt  = FFT.multiply(patCubs, text);
    double[] p2Xt2 = FFT.multiply(patSqrs, texSqrs);
    double[] pXt3  = FFT.multiply(texCubs, pattern);

    double[] matches = new double[len];
    for(int i=len-1; i>=len-matLen+patLen-1; i--)
    {
        matches[i] = p3Xt[i] - 2*p2Xt2[i] + pXt3[i];
        if (matches[i] == 0)
        {
            System.out.print(len-i + " ");
            break;
        }
    }
    System.out.println();
}
```