COMS21103: Coursework 2

2014/2015

Due: 12th January 2015 (by 23:59) via the online submission system.

This coursework consists of two parts. The first (and larger) part of this assignment is to investigate how string matching problems can be solved using the tools given in lectures. Given a pattern p of length m and a text t of length n, where both p and t may contain optional "don't care" symbols which match any single character in the alphabet, it is possible to find all exact matches of p in t in $O(n \log m)$ time. The central algorithm to achieve this in the required efficiency is the FFT. The main task of this coursework given in Part A to implement a complete solution to pattern matching with don't cares.

The second part then asks you to solve a dynamic programming problem about planning your social life. There is no marked implementation component for Part B. However, you may wish to implement some of your ideas to test them out.

Marks *The parts are not equally weighted.* Part A has a total mark of 80. Part B has a total mark of 40. Your raw mark, out of 120, will be converted to a percentage and rounded up. For example, 100/120 would be awarded 84% (a rather good score).

Submission You shouldn't necessarily expect to do all the questions; part of the assessment is how far you manage to get in the available time. Remember that you may be able to do some of the later stages without completing the earlier ones. Electronic submissions should be made using the online submission system. Please include your name and user id on the top of *every* page you submit (this is to help us mark them once they are printed).

Part A

The general aim of this part is an understanding of the theory rather than only being a programming exercise. Where there is some debate about the right way to proceed, the coursework generally demands you make informed design decision and back it up with a reasonable argument described in your written answers. You may find Chapter 30 of CLRS helpful for Part A as well as Chapter 2, Section 2.6 of DPV.

You must submit written answers as well as code for every question in this part. Where a question asks you for an implementation, your written answer

should state what you have done, any important decisions you have made and what works and what doesn't work in your code and how you tested it. It should also state the time complexity of the code you submit and why you believe it to be as you claim. Pseudocode should be presented in a style similar to that given in CLRS or the lecture slides but clear explanations will form the most important parts of your written answers.

All implementations must be in Java with the main file name for each question named "Cwk2QuestionX.java". E.g. for question 1 the command line needed to run your code in Linux after it is compiled will be "java Cwk2Question1 `<filename>`", where `<filename>` is the name of an input file. Your code should compile cleanly and be tested using the default version of the JDK available in the lab before submission. Any error messages should be written to stderr. The only output printed on stdout should be that specified within the assignment description; you should not include, for example, debugging code that prints anything else to stdout or stderr. The code you submit must also be properly commented. For the written answers for Part A, you should submit them in *pdf* format electronically as a file called "cwk2PartA.pdf".

Remember, your goal is to communicate. Full marks will be given only to correct solutions which are described clearly. Convoluted and obtuse descriptions will receive low marks.

Suggested exercises: These will not affect your marks, but you are encouraged to try them.

- Exercise 2.7, 2.8, 2.9 and 2.10 from DPV
- Exercise 30.1-3 from CLRS
- Exercise 30.1-6 from CLRS
- Exercise 30.2-4 from CLRS

Listing 1: Incomplete Complex ADT

```
public class Complex {
    private final double re;   // real part
    private final double im;   // imaginary part

    // create a new object
    public Complex(double real, double imag) {
        re = real;
        im = imag;
    }

    // return string representation Complex object
    public String toString() {
        if (im == 0) return re + "";
        if (re == 0) return im + "i";
        if (im <  0) return re + " - " + (-im) + "i";
        return re + " + " + im + "i";
    }

    // return new Complex object plus b
    public Complex plus(Complex b) {
        Complex a = this;             // invoking object
        double real = a.re + b.re;
        double imag = a.im + b.im;
        return new Complex(real, imag);
    }

    // return new Complex object times b
    public Complex times(Complex b) {
        Complex a = this;
        double real = a.re * b.re - a.im * b.im;
        double imag = a.re * b.im + a.im * b.re;
        return new Complex(real, imag);
    }

    // complete as needed...
}
```

Questions:

1. Complete the Complex ADT by adding the methods, *minus* and *conjugate*. Implement a *main* method that tests all the different methods you have defined using the following code snippet as a basis.

```
public static void main(String[] args) {
    Complex a = new Complex(7.0, 4.0);
    Complex b = new Complex(-2.0, 6.0);

    System.out.println("b + a = " + b.plus(a));
    System.out.println("a - b = " + a.minus(b));
    System.out.println("a * b = " + a.times(b));
    System.out.println("Conj[a] = " + a.conjugate());
    System.out.println("Conj[b] = " + b.conjugate());
}
```

   [10 points]

2. By following the pseudocode description given in the FFT lecture slides (see slide 22), implement the FFT (assume the input length is a power of two). You can test your implementation using $(0, 0, 1, -1)$ as input (corresponding to the polynomial $x^2 - x^3$) whose four point Fourier transform is $(0, -1 + i, 2, -1 - i)$. You should also create other test cases for yourself of different lengths. Your code should take in a file with a single line of space separated real coefficients as input and output its FFT as space separated complex coefficients. Each complex number in the output will be represented as a comma separated pair. For example, if the input file is

   ```
   0 0 1 -1
   ```

   the output should be

   ```
   0,0 -1,1 2,0 -1,-1
   ```

   If the values you calculate are not whole numbers, then they should be accurate to at least two decimal places.

   More examples can be found at
   `http://www.cs.bris.ac.uk/~sach/fft-examples.txt`.
   To make these into the correct input format you will need to copy the input lines into their own files.

   [20 points]

3. Implement the inverse FFT (see slide 32). You can assume the input length is a power of two. Test your implementation using $(0, -1+i, 2, -1-i)$ as input whose inverse FFT is $(0+0i, 0+0i, 1+0i, -1+0i)$. Your code should take in a file with a single line of space separated complex coefficients as input and output its inverse Fourier transform. For example, if the input file is

```
0,0 -1,1 2,0 -1,-1
```

the output should be

```
0,0 0,0 1,0 -1,0
```

If the values you calculate are not whole numbers then they should be outputted to at least two decimal places. You should also try the outputs from your implementation of the FFT to make sure you can recover the correct inputs accurately.

[10 points]

4. Implement a polynomial squaring algorithm using FFTs. Your code should take in a file with a single line of space separated coefficients of the polynomial and output the coefficients of its square. The time complexity of your implementation should be $\Theta(n \log n)$. For example, if the input file is

```
1 2 3 4
```

the output should be

```
1 4 10 20 25 24 16
```

[10 points]

5. We will now investigate how to perform string matching using the fast Fourier transform. Assume we are given a text $t = t_1, t_2, \ldots, t_n$ and a pattern $p = p_1, p_2, \ldots, p_m$. We assume that each character in the pattern and text has been mapped to an integer value as a preliminary step. For example we can take the ASCII values of the characters they represent so that *abc* becomes $p = 97, 98, 99$. Now we have that p matches $t[i, \ldots, i + m - 1]$ if and only if $S_i = \sum_{j=1}^{m}(p_j - t_{i+j-1})^2 = 0$. Using this fact, find a way to use the FFT and inverse FFT to calculate S_i for every i in $\Theta(n \log n)$ time[1]. Implement your solution as part of an exact string matching algorithm that reports the position of the first match of p in t.

[1] Hint: multiply out $(p_j - t_{i+j-1})^2$ and consider the three resulting sums separately. Now think of t and p as the coefficients of polynomials and work out how you can use polynomial multiplication to calculate the middle sum $\sum_{j=1}^{m} p_j t_{i+j-1}$.

You may assume that $m \leq n$ but should make no further assumptions about the sizes of m and n.

Your code should take in a file with two strings separated by a newline as input and output a single integer value for the position of the match. For example, if the input file is

```
The cat sat on the mat
at
```

the output should be

```
6
```

Your code will be tested on inputs of arbitrary length which are much longer than the example given above.

[20 points]

6. Exact string matching can be solved in linear time using KMP, for example, so it might seem a little odd to use the Fourier transform for this task. We will now see how the method from Part A question 5 can be extended to solve a problem called *exact matching with don't cares* for which simpler methods are not known. The overall time complexity for this algorithm will be $\Theta(n \log n)$. Again, assume that the input has been converted to strings of integer values and this time that all don't care characters have been encoded as 0. Consider

$$S_i = \sum_{j=1}^{m} p_j t_{i+j-1}(p_j - t_{i+j-1})^2 = \sum_{j=1}^{m} (p_j^3 t_{i+j-1} - 2p_j^2 t_{i+j-1}^2 + p_j t_{i+j-1}^3)$$

which equals 0 if and only if there is an exact match with don't cares. Figure out how to compute S_i in $\Theta(n \log n)$ time and then implement your solution to exact matching with don't cares.

Your code should take in a file with two strings separated by a newline as input and output a single integer value for the first position of a match. For example, if the input file is

```
Th? cat?sat??n the ?at
bin?th? ca?
```

the output should be

```
12
```

Observe that in the input, the don't care characters are encoded as ? symbols. Your code will be tested on inputs of arbitrary length which are much longer than the example given above.

[10 points]

Part B

This part is about dynamic programming. For Part B, you should submit your answers in *pdf* format electronically as a file called "cwk2PartB.pdf". There is no implementation submission for Part B. However, as mentioned above, you may (or may not) wish to implement some of your ideas to test them out. This may be a good way to find intuition. All pseudocode should be presented in a style similar to that given in CLRS or the lecture slides alongside a brief but clear explanation of why it works.

Remember, your goal is to communicate. Full marks will be given only to correct solutions which are described clearly. Convoluted and obtuse descriptions will receive low marks.

Suggested exercises and reading: There are more dynamic programming problems on problem sheet 9 and many more online (e.g. JE Chapter 5). I highly encourage you to attempt some of them as practice for this part. Finding Dynamic Programming recurrences is a skill and like any skill you can get better by practicing. I strongly believe that this is a skill worth acquiring.

You may also find Chapter 15 of CLRS or Chapter 5 of JE (available online and linked from the course webpage) helpful. These chapters contain explanations of solutions to several Dynamic Programming problems that we haven't discussed. You may find inspiration for the questions below by reading these.

The SOCIAL-LIFE problem Being a very organised person you decide to plan your social life for the next n days in advance. On each day you can decide between three options: Go out for the evening (denoted `G`), have friends over (denoted `F`) or do nothing (denoted `N`). Of course, the happiness you will get from day i depends on the option you pick. For each day i, let $g_i > 0$ denote the happiness (in some arbitrary, integer happiness units) you will get from going out on day i. Similarly, $f_i > 0$ denotes the happiness you will get from having friends over. The happiness you get from doing nothing is always zero. Your goal is to pick a plan which maximises your total happiness (summed over all days). However, there is a twist! If you go out (`G`) on day i, you are too tired to go out on day $i + 1$ and must do nothing (`N`) on that day.

The input to the SOCIAL-LIFE problem is a length n array A of pairs $A[i] = (g_i, f_i)$ giving the happiness values for day i. The output is the value OPT which is the maximum total happiness that can be achieved by any plan. Finding OPT is the main focus of Part B. In the final question of Part B you will also be asked how to find the plan itself.

Example Suppose that $n = 5$, in table form, the input is given by:

Day	1	2	3	4	5
g_i	7	50	4	18	16
f_i	5	1	5	12	7

One (greedy) plan is given by `GNFGN`. This plan has total happiness 30. An alternative plan is given by `FGNGN` which has plan has total happiness 73 (much better!). Neither of these is an optimal plan which I claim is `FGNFF` which has total happiness 74. You should check that you agree that this is in fact an optimal plan.

Clarifications You are always allowed to go out or have friends over on the 1st day (let's assume that it's the first day of term and you are well rested from the holiday). You may choose to go out or have friends over on the n-th day if you didn't go out on day $(n - 1)$ (the n-th day is the last day of term so you can rest on the first day of the holiday).

Hint The problem has been chosen so that the algorithms you are asked to develop for questions 2, 3 and 4 should be short (5-10 lines) and straightforward to write down in pseudo-code. If your pseudo-code for any of these algorithms is significantly longer than 10 lines, it is likely you have done something wrong.

Questions:

1. Give a recursive formula for the SOCIAL-LIFE problem. You should explain why your recursive formula is correct but you do not have to give a formal proof. You should also explain how to obtain the value OPT from your recursive formula.

   [10 points]

2. Based on the recursive formula you gave in the previous question, give a (non-memoized) recursive algorithm for the SOCIAL-LIFE problem. You should give the algorithm in clear pseudo-code and also briefly explain what it does.

   [6 points]

We now move on to constructing an efficient solution to the SOCIAL-LIFE *problem. The algorithms that you give in the following questions should each run in O(n) time. You are not asked to prove this for any of the algorithms, but you should be able to justify it to yourself.*

3. Memoize your recursive algorithm. Again, you should give the algorithm in clear pseudo-code and also briefly explain what it does. By referring to your answers to the previous questions (or otherwise), argue that your algorithm is correct.

   [8 points]

4. Derive an iterative algorithm for the SOCIAL-LIFE problem. Once again, you should give the algorithm in clear pseudo-code and also explain briefly what it does. By referring to your answers to the previous questions (or otherwise), prove that your algorithm is correct.

   [8 points]

5. Of course, when planning your social life, it is rather important to know not just how happy your plan will make you but also what the plan actually is! Give an algorithm which outputs both the value OPT and an optimal plan (i.e. a description of which option to take on each day). Predictably, you should give the algorithm in clear pseudo-code and also briefly explain what it does. You may use your algorithms from previous questions as subroutines (but please use clear names). Briefly explain why your algorithm works. You do not need to give a proof of correctness.

   [8 points]