# COURSEWORK 1 FOR INF2B (ADS THREAD, 2015-16)
# STRING MATCHING

**Submission Deadlines:** The coursework consists of two parts (of a different nature) relating to one problem. As shown below these have separate deadlines, you *cannot* swap the parts round or submit either part later than the stated deadline (unless you have permission to do so from the course organiser).

- Part 1 consists of Exercise 1 on page 4, §2.1 and Exercise 2 on page 9, §2.2; these involve fairly straightforward analysis.

  *Submit Part 1 by: 4.00PM, 22 February 2016.*

- Part 2 consists of the tasks in §3; these are all software related.

  *Submit Part 2 by: 4.00PM, 04 March 2016.*

See §4 at the end of this document for instructions on how to submit.

## §1. Introduction

In this practical we will consider two methods of searching for all occurrences of a given pattern within some text. For example we might have a long piece of English text about Scotland and we wish to search for all occurrences of "Edinburgh" within it. Our aim is to return the offset position of all occurrences, if there are no occurrences then we indicate this by a special convention discussed below.

The practical has several aims:

1. Practice at asymptotic analysis (with guidance).
2. Careful implementation of one algorithm (an implementation of the other algorithm is supplied).
3. Carry out timing experiments in order to:
   (a) Compare the algorithm that you implement with one that is supplied and decide the point from which one is more efficient than the other (i.e., the overheads are overweighed by the advantages).
   (b) Determine the constant for the asymptotic analysis as it applies to your particular implementation.

You can concentrate on Part 1 (up to and including §2) first and then on the rest. However it would be a good idea to read the entire document through initially (without going into great detail) so that you know what is expected for the complete coursework. Note that although the document is fairly long the tasks you are required to carry out are quite modest. The emphasis here is on understanding the problem well and so plenty of discussion has been included.



The practical requires you to submit various things. Each of these has some appropriate marks assigned. It would however be a serious error to assume that by carrying out only these tasks you have obtained the most (or indeed the main) benefit from the practical. Throughout the text you are prompted to think about various matters that are not part of the submission. You should consider these carefully along with possibly other issues as they occur to you. In other words avoid the pitfall of just going through the motions in a mechanical fashion; in any case this is not possible for most parts. The practical is issued to help you develop your understanding of the material, the marks are a by-product (of course an important one) and far from being its only rationale.

### Notes

**Good Scholarly Practice:** Please remember the University requirement as regards all assessed work for credit. Details about this can be found at:

http://www.ed.ac.uk/schools-departments/academic-services/students/undergraduate/discipline/academic-misconduct

and at:

http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct

Furthermore, you are required to take reasonable measures to protect your assessed work from unauthorised access. For example, if you put any such work on a public repository then you must set access permissions appropriately (generally permitting access only to yourself, or your group in the case of group practicals).

**Following instructions:** The various parts of this practical place certain requirements with a penalty if these are not followed. *No negotiation of any kind will be entered into where the requirements are not followed.* The point of the requirements is to ensure that in doing the various parts you practice and demonstrate understanding of the appropriate areas of the course. What you submit must be considered work; imagine that your work was being given to you and others as a sample answer. If you, or others, would find it unclear and unhelpful then your work needs further revision.

Regrettably the various requirements and penalties can appear somewhat censorious and limiting. This is not the intention, indeed much of what is stated can be seen in the much more positive light of reassurance that the various tasks have quite simple answers. Please be assured that plenty of variation is possible even when following the requirements, i.e., there isn't necessarily a unique way to do and present things clearly, concisely and correctly.

## §2. String matching

Our discussion is based on the chapter on string matching from the book *Introduction to Algorithms* by Cormen, Leiserson, Rivest and Stein. We are interested in strings over some fixed alphabet (which we need not worry about at this stage, e.g., it could be ASCII characters). We are given some *text* which is simply a string over the alphabet. We will represent the text by

means of an array $T$ which we will assume is indexed starting with 1 (this keeps some of the operations more straightforward). Our convention will be to think of the text as the array $T[1\,..\,n]$, i.e., the array has $n$ entries and $T[i]$ gives us the $i$th character of the text. We are also given a *pattern* which is simply another string over our alphabet. We will always represent the pattern by an array $P[1\,..\,m]$. If $m > n$ then the problem is trivial as there can be no occurrences of the pattern within the text. We will therefore assume throughout that $m \leq n$.

Before going any further let us look at a simple example. Suppose the text is *bacababa* and the pattern is *bab*, so that $n = 8$ and $m = 3$. We can think of their representations diagrammatically as follows:

text $T$

| $b$ | $a$ | $c$ | $a$ | $b$ | $a$ | $b$ | $a$ |
|---|---|---|---|---|---|---|---|

pattern $P$

| $b$ | $a$ | $b$ |
|---|---|---|

Clearly the pattern occurs in the the text starting at position 5 but nowhere else.

We will adopt the following convention. We say that the pattern $P$ occurs in the text with *shift* (or *offset*) $s$ if and only if

$$P[1\,..\,m] = T[s+1\,..\,s+m]. \tag{1}$$

The equation is to be understood as stating that $P[i] = T[s+i]$, for $1 \leq i \leq m$. One advantage of this definition is that it can be expressed as $P[t\,..\,m] = T[s+t\,..\,s+m]$ where $t$ is the base address of the start of our text (so in this passage we have $t = 1$ and in Java $t = 0$). The point is that the offset is independent of $t$.

Our problem is to find all shifts $s$ such that equation (1) holds. We will return these shifts as a list in increasing order. Thus the pattern occurs in the text if and only if the list is not empty. So for our example we would return a list of one cell containing the number 4, i.e., the shift (note that the shift is one less than the position at which a pattern occurs in the text since we number array locations from 1 onwards [what happens if we number them from 0?]). We can think of our list as a queue, though we only ever add items to it (of course subsequent use of the list would remove items to find out where the pattern occurs though we will not do this).

### §2.1 Naive algorithm

There is an obvious algorithm for our problem. We can think of it in terms of placing the pattern under the text in successive positions (starting with 1) and testing to see if the pattern matches the text immediately above it.

```
Naive-Matcher(T, P)
1.   n ← T.length
2.   m ← P.length
3.   create a new empty queue S
4.   for s ← 0 to n − m do
5a.      if P[1 .. m] = T[s + 1 .. s + m] then enqueue(s, S)
```



Of course the test $P[1\,..\,m] = T[s+1\,..\,s+m]$ has to be implemented as a loop, this has not been shown above for the sake of clarity. Line $5a$ should be seen as shorthand for the following code:

```
5. matches ← TRUE
6. for i ← 1 to m do
7.    if P[i] ≠ T[s + i] then
8.       matches ← FALSE
9.       exit for loop
10. if matches then enqueue(s, S)
```

The lines have been numbered so that this can be slotted into the preceding pseudocode and in your analysis assume that this has been done so you can refer to the line numbers as given

We have also shown $S$ as a parameter to enqueue for the sake of clarity, if implemented in Java then we make the appropriate method call. Another point to note is that, as mentioned above, our convention here is to number arrays starting at 1 whereas Java starts at 0. We will discuss this below, for now let us not worry about the implementation language.

**Exercise 1.** For item 1 you *must* use the $O(\cdot)$ notation along with any appropriate properties as discussed in Lecture Note 2. You may *not* use named constants as part of the analysis. For item 2 you could use named constants but in fact this is not necessary, the $\Omega(\cdot)$ notation enables things to be expressed simply and well. Finally, note that for full marks you must justify each expression or claim with reference to the algorithm. *Any answers that do not follow these requirements will be awarded 0.*

Let $T_{NM}(n, m)$ denote the worst case runtime of the algorithm Naive-Matcher on inputs of size $n$ and $m$ respectively.

1. Prove that $T_{NM}(n, m) = O((n - m + 1)m)$. Note that the additive constant $+1$ is needed to take care of the case when $m = n$. [15%]

2. Prove that $T_{NM}(n, m) = \Omega((n - m + 1)m)$. Thus you must produce inputs of size $n$, $m$ that cause the algorithm to have runtime as claimed. You *cannot* fix either of $n$ or $m$. (HINT: There are very simple examples.) [10%]

3. Is it true to say that $T_{NM}(n, m) = \Theta((n - m + 1)m)$? Justify your answer briefly. [5%]

**Note:** The second part of the exercise is a good opportunity to underline the important point that for many algorithms the worst case runtime for given input size(s) does not happen for all possible inputs but only for some appropriate ones. As an example here, if we take the text to consist entirely of the character $a$ and the pattern to consist of anything starting with the character $b$ then the runtime is $O(n)$. We saw the same point in connection with the simple linear search algorithm.

## §2.2 The Knuth-Morris-Pratt algorithm

Let us look again at the simple example. In trying to see if there is a match of the pattern with the text starting at position 1, the first two characters match but the third fails. Our knowledge of the text can be pictured as:

text $T$ | $b$ | $a$ | ? | ... |

pattern $P$ | $b$ | $a$ | $b$ |

(It is true that we looked at the next character in the text but we want to relate our knowledge of the text to the pattern, taking the extra character into account does not help in the following discussion.) This tells us immediately that it is not worth trying a match of the pattern with the text starting at position 2, since the first character of the pattern fails to match its second character. On the other hand it is worth trying position 3. This observation (and its generalisation below) is crucial for the development of the algorithm.

Consider the general situation where we are comparing the pattern against the text starting at position $s+1$ (recall that $s$ is the shift from position 1 of the text). We have a match exactly up to position $q$ of the pattern (i.e., position $s+q$ of the text). We allow $q = 0$ to mean there is no match at all (this makes sense, $q$ is the number of characters matched). We have found a complete match of the pattern in the text if and only if $q = m$ otherwise there is a mismatch with the character at position $q+1$ of the pattern. Whatever is the case we now know that $T[s+1\,..\,s+q] = P[1\,..\,q]$. Given this knowledge, we can decide just from the pattern if it makes sense to try matching it with the text starting at position $s+2$. We do this by comparing the pattern with itself thus:

| $a_1$ | $a_2$ | $a_3$ | $a_4$ | ... | $a_q$ |

| $a_1$ | $a_2$ | $a_3$ | ... | $a_{q-1}$ |

We see that it is only worth trying a match with the text at the next position if and only if the mach above is successful (this is the same as saying that $T[s+2\,..\,s+q] = P[1\,..\,q-1]$). Suppose that this fails, i.e., the shift of the pattern does not match with the pattern up to position $q$. We can again decide just from the pattern if it is worth trying at the next position. Our diagram is:

| $a_1$ | $a_2$ | $a_3$ | $a_4$ | ... | $a_q$ |

| $a_1$ | $a_2$ | ... | $a_{q-2}$ |

Here we need $T[s+3\,..\,s+q] = P[1\,..\,q-2]$.

The preceding discussion shows that what we need is the least shift $s' > s$ such that

$$T[s'+1\,..\,s+q] = P[1\,..\,s+q-s'].$$

We can rephrase this as: the least shift $s' > s$ such that

$$T[s'+1\,..\,s+q] = P[1\,..\,k],$$



where $s'+k = s+q$. We can also write the equation as $T[s'+1\,..\,s'+k] = P[1\,..\,k]$ [why?]. The crucial point here is that we can pre-compute these shifts from the pattern alone (recall that, by assumption, $T[s'+1\,..\,s+q]$ consists of the first $q$ characters of the pattern). An equivalent interpretation is to find the largest $k < q$ such that the stated condition holds. Since $s' = s+q-k$ we can eliminate $s'$ altogether and write the equation as

$$T[s+q-k+1\,..\,s+q] = P[1\,..\,k].$$

(It is important to get the indices right but don't let this disguise the main idea which is quite simple and intuitive.) Note that such a $k$ always does exist since $k = 0$ satisfies the condition except that it might not be the largest. On the other hand we require $k < q$ so that the possible values of $k$ are bounded from above and hence there is a maximum.

We define the function $\pi : \{1, 2, \ldots, m\} \to \{0, 1, \ldots, m-1\}$ by requiring that $\pi(q)$ is the value $k$ as just described. (We do not need to define $\pi(0)$ because $q = 0$ means there was no match at all and so we will automatically move on. ) Note that the function $\pi$ can be represented by an array indexed from 1 to $m$ and we will use this without further comment. From the preceding discussion we know that

$$0 \le \pi(q) < q$$

for all $q$. Make sure that you understand fully the meaning of this function and how it relates to the discussion above. The following example will help.

The function for our simple pattern $bab$ is given by:

| $q$ | 1 | 2 | 3 |
|---|---|---|---|
| $\pi(q)$ | 0 | 0 | 1 |

Let us see exactly why these values are correct by looking at them in turn.

$\pi(1)$ : Here $T[s+1\,..\,s+1]$ is the same as the pattern entries $P[1\,..\,1]$, i.e., $T[s+1]$ is the same character as $P[1]$. We want the largest $k$ such that $P[1\,..\,k] = T[s'+1\,..\,s'+k]$ where $s'+k = s+1$. Since we require $s' > s$ this immediately means that $s' = s+1$ and $k = 0$.

Note that this argument applies to *any* pattern, we always have $\pi(1) = 0$. It is exactly what we would expect: we are told that the attempt to match the pattern with the text starting at position $s+1$ failed at position $s+2$ (i.e., $T[s+2] \neq P[2]$). It follows that the next position at which we should try a match is at $s+2$.

$\pi(2)$ : We have $T[s+1, s+2] = P[1, 2]$. Is it worth trying for a match at position $s+2$ of the text? Well such an attempt would look like:

| $b$ | $a$ |

| $b$ |

Clearly this is bound to fail. Thus we set $k = 0$ and this means that $s' = s+2$, there is no point in attempting a match at position $s+2$ (recall that the next attempted match starts at position $s'+1$).

$\pi(3)$ : We have $T[s+1, s+2, s+3] = P[1, 2, 3]$. Is it worth trying for a match at position $s+2$ or $s+3$ of the text? An attempt at position $s+1$ would look like:

| $b$ | $a$ | $b$ |
|---|---|---|

| $b$ | $a$ |
|---|---|

Clearly this would fail (we already knew this from the previous case of course). So now we ask if it is worth trying for a match at position $s+3$. Here the picture is:

| $b$ | $a$ | $b$ |
|---|---|---|

| $b$ |
|---|

Thus it is worth trying this (and from the information we have we cannot rule out a match). So $k = 1$ and $s' = s + 2$.

It must be stressed that although the discussion above has referred to parts of the text input, all these are simply initial segments of the pattern. For any initial segment (i.e., value of $q$) we have a value for $\pi(q)$. The references to the text were simply a matter of convenience and for aiding understanding. For a given value of $q$ the value of $\pi(q)$ is simply the largest $k$ such that

$$P[q - k + 1 \,..\, q] = P[1 \,..\, k].$$

Note that for $k = 0$ both sides of this equation consist of an empty array so the equation holds. In terms of a picture we require the pattern to look like:

Pattern up to position $q$

| $a_1$ | $\cdots$ | $a_{q-k}$ | $a_1$ | $a_2$ | $\cdots$ | $a_k$ |
|---|---|---|---|---|---|---|

Pattern shifted to position $q - k + 1$

| $a_1$ | $a_2$ | $\cdots$ | $a_k$ |
|---|---|---|---|

In intuitive terms just think about keeping one copy of the pattern up to position $q$. We then place another copy under it but shifted along by one position, ignore the last entry (it falls out of scope), and check for a match. If not we shift the bottom along by one more place and again check for a match with the characters directly above. We keep doing this till we either find a match (in which case $k$ is the number of entries matched) or the bottom copy falls off the right hand end (in which case $k = 0$). You should now be able to see that the lengthy discussion above for the pattern $bab$ can be summarised as:

| $q$ | 1 | 2 | 3 |
|---|---|---|---|
| Pattern | $b$ | $a$ | $b$ |
| Shift 1 | | $b$ | $a$ |
| Shift 2 | | | $b$ |
| $\pi(q)$ | 0 | 0 | 1 |

Make sure you understand how we can work out $\pi(q)$ for each $q$ straight from the array of the pattern and its shifts. Start with $q = 1$, how much of the displayed array is relevant? For each $q$



cover up the irrelevant part, if any, and then look for the maximum number of characters that match completely on any shift line with the relevant part of the pattern.

The method we have used so far to compute $\pi$ is not particularly efficient (as a function of $m$, the length of the pattern $P$). The crucial aspect of the approach is that we can compute the function very efficiently. Here is the pseudocode (recall that we hold the value of $\pi(i)$ in an array which we will also call $\pi$):

Compute-Prefix-Function$(P)$
1. $m \leftarrow P.length$
2. $\pi[1] \leftarrow 0$
3. $k \leftarrow 0$
4. **for** $q \leftarrow 2$ **to** $m$ **do**
5. **while** $k > 0$ **and** $P[k+1] \neq P[q]$ **do** $k \leftarrow \pi[k]$
6. **if** $P[k+1] = P[q]$ **then** $k \leftarrow k + 1$
7. $\pi[q] \leftarrow k$
8. **return** $\pi$

Remember that in this section we assume arrays start at index 1 but of course in Java they start at index 0 so you will need to make an adjustment from the pseudocode to the Java implementation of Compute-Prefix-Function, see the note on p. 12. It is by no means obvious that this algorithm is correct, indeed the runtime is also not obvious. A good way to begin the study of an algorithm is to simulate it on simple examples. In order to help you with this here are two further examples of patterns and their functions:

1. $P = bbcbbabbc$

| $q$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| $\pi(q)$ | 0 | 1 | 0 | 1 | 2 | 0 | 1 | 2 | 3 |

2. $P = ababbababbabc$

| $q$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\pi(q)$ | 0 | 0 | 1 | 2 | 0 | 1 | 2 | 3 | 4 | 3 | 4 | 0 |

You should first check that these functions are correct by computing $\pi$ by the naive method. Then simulate Compute-Prefix-Function, at least for the first few values. In fact the supplied code (see §3.1) has an implemementation of the *naive* method of computing the prefix function. This is in the `Matcher` class and the method is:

```
public static int[] buildPrefixFunctionNaively(String pattern)
```

**Important:** This method is supplied to help you in cross checking your work. You *must not* use it as part of your implementation since it would invalidate all the timing tests (in effect you would be implementing an algorithm that is significantly slower than naive matching!).

A proof of correctness of the Compute-Prefix-Function is not easy. Again it would be a good exercise to try proving the correctness of the algorithm, or more accurately think about

how you might approach such a task (put a limit on the time you spend on this). You can find a complete proof in the book *Introduction to Algorithms* by Cormen, Leiserson, Rivest and Stein.

The worst case runtime is $\Theta(m)$ but a naive analysis does not work because of the presence of a while loop (on line 5) within the overall for loop. We can derive the bound by using a form of *amortized analysis*. We will not go into details for Compute-Prefix-Function because the approach is similar to the one discussed below for KMP-Matcher.

We are now ready to look at a more efficient string matching algorithm.

```
KMP-Matcher(T, P)
1.  n ← T.length
2.  m ← P.length
3.  π ← Compute-Prefix-Function(P)
4.  q ← 0
5.  create new (empty) queue Q
6.  for i ← 1 to n do
7.    while q > 0 and P[q + 1] ≠ T[i] do q ← π[q]
8.    if P[q + 1] = T[i] then q ← q + 1
9.    if q = m then
10.     enqueue(Q, i − m)
11.     q ← π[q]
12.   return Q
```

The correctness of this algorithm is also not obvious but not so hard to establish given the correctness of Compute-Prefix-Function. The next exercise will find the worst case runtime. Note that this is a type of analysis that we do not otherwise cover in the course. It has therefore been broken down into small and simple steps so that the things you have to do are ones that are common with the type of analysis we have seen many times. Finally, although the question shows three main items the last one concludes the analysis for you and is not a question as such.

**Exercise 2.** Define the function $\Phi(s)$ as follows: $\Phi(0) = 0$ while for $s > 0$ the value of $\Phi(s)$ is the value of $q$ just after the body of the for loop of line 6 in KMP-Matcher has executed for the value $i = s$. Consider an execution of the for loop on line 6 when $i = s$ and count the following:

- the number of executions of the statement controlled by the **while** loop on line 7;
- the number of executions of the statement controlled by the **if** on line 8;
- the number of executions of the statements on lines 10, 11 (controlled by the **if** on line 9).

Denote this count by $c_s$. Define

$$\hat{c}_s = c_s + \Phi(s) - \Phi(s-1).$$

This is called the *amortized* cost of the execution of the loop and the function $\Phi$ is called a *potential*.



1. Let $c$ be the total cost of executing the for loop in KMP-Matcher, counted in terms of the statements listed above, and $\hat{c}$ the total amortized cost (i.e., $c = c_1 + \cdots + c_n$ and $\hat{c} = \hat{c}_1 + \cdots + \hat{c}_n$). Prove that $\hat{c} = c + \Phi(n) - \Phi(0)$. Explain briefly why it now follows that $c \leq \hat{c}$. [15%]

2. Prove that $\hat{c}_s = O(1)$ for all $s$, indeed $\hat{c}_s \leq 4$. Break this down into the following easy parts:

   (a) Let $q_0$ be the value of $q$ before the while loop on line 7 is executed (i.e., $q_0 = \Phi(s-1)$) and let $q_1$ be the value of $q$ just after the while loop terminates. Let $w_s$ be the number of times that the while loop body is executed. Prove that $w_s + q_1 - q_0 \leq 0$. Recall that $\pi(q) < q$ for all $q$; what happens to the value of $q$ with each execution of the loop?

   (b) Let $q_2$ be the value of $q$ after line 8 is executed. Prove that $1 + q_2 - q_1 \leq 2$.

   (c) Let $q_3$ the value of $q$ just after the **if** statement starting on line 9 is executed (i.e., $q_3 = \Phi(s)$). Prove that $2 + q_3 - q_2 \leq 2$.

   (d) Now put these observations together to deduce the overall claim. Note that $c_s \leq w_s + 1 + 2$, thus
   $$\hat{c}_s \leq (w_s + 1 + 2) + \Phi(s) - \Phi(s-1).$$

   NOTE: Each of these requires no more than a few lines to justify. So think carefully and express your justification very clearly. *Overlong or unclear answers will receive very much reduced credit and any answers that cannot be understood after 2 minutes (per part) will be awarded 0.* [20%]

3. We now use the preceding two main parts to deduce that the worst case runtime of the algorithm is $O(n)$; there is nothing for you to submit here but you should read through this conclusion of the analysis and make sure you understand it. We use without proof the fact that the worst case runtime of Compute-Prefix-Function is $\Theta(m)$; recall that $m \leq n$.

   Lines 1, 2, 4 and 12 of KMP-Matcher all cost $O(1)$ while line 3 is $O(m)$. The cost of executing the **for** loop on line 6 for $i = s$ is at most $ac_s + b$ for some constants $a$ and $b$; if you have time give a brief justification of this observation. It follows that the overall cost of executing the **for** loop on line 6 is at most $ac + bn$ and hence at most $a\hat{c} + bn$. From the preceding part we deduce that $a\hat{c} + bn = O(n)$. Thus the total cost is $O(1) + O(1) + O(1) + O(1) + O(m) + O(n) = O(n)$ since $m \leq n$.

   In fact we can improve our analysis to show that the worst case runtime is $\Theta(n)$ but we will not do this here (would you expect it to be sub-linear?).

## §3. Software tasks

We first give a guide to some of the suppled classes and methods (some more are mentioned later).

### §3.1 Supplied software

The files that you will need for this coursework can be found on-line at the coursework webpage:

http://www.inf.ed.ac.uk/teaching/courses/inf2b/coursework/cwk1.html

This page contains a file called `inf2bcw1.tar` which you should download. The file is tarred so you need to extract it, e.g. by `tar -xf inf2bcw1.tar`, this will create a subdirectory `src` of your current one that contains the software. The `java` package for this coursework is `package matcher`. The only class that should be changed is the nearly empty class in the file `StudentClass.java` which is where you will implement your part of the project. In fact the amount of code you need to write is fairly modest.

Recall that in the discussion above we assumed that arrays were indexed starting from 1. However Java starts with 0. Rather than invent a new class in Java it is simplest to subtract 1 from all array indices. This is very important otherwise your software will not work.

**Note:** It is a part of the exercise that you translate the description of the algorithm based on arrays starting at 1 to arrays starting at 0 in Java. This is a deliberate choice so that you practice switching conventions. In this case it is a very simple matter. You *cannot* throw away location 0 of Java arrays and start using them from 1 onwards.

**Important:** Do *not* remove any headers from the supplied software including those in the `StudentClass.java` file.

In the software the text and pattern will be represented by a `String`; our alphabet is just any valid Java character (in experiments we will use a restricted alphabet, this is discussed below). The following methods and classes are already supplied for you.

- class `Queue`. This implements the queue we use to hold the offsets for occurrences of the text. In fact for this practical we only need two methods.
  - `public Queue enqueue(Integer s)`
    This puts `s` at the end of the queue.
  - `public Queue toString()`
    This simply returns a comma separated string of all the entries in the queue so that you can print it out for checking your implementation. If the queue is empty then the empty string is returned.
- Class `Matcher` that contains various methods including
  - `public Queue naiveMatcher(String T, String P)`

  This takes the text `T` and pattern `P` and returns a `Queue` structure which consists of all the shifts in the text at which the pattern can be found. For example if our text is `"aababaab"` and the pattern is `"aa"` then the returned `Queue` structure has `0, 5` as its elements in this order. With the same text but with the pattern `"aaa"` the returned `Queue` is empty.



### §3.2 Tasks

**Note:** You are asked to write some code that is in fact fairly simple thanks to the supplied methods. *Any code that is incorrect will be awarded 0.* Correct code will be marked for style as well. This means that it should have useful comments and helpful indentation. Note that pointless comments (e.g., stating the obvious such as "now we add the two variables together") or excessive indentation are almost as bad as the complete absence of either.

**Important:** The checking of your code includes an automated procedure. Partly for this reason, you *must* follow the specification exactly and must not change any of the method names or their parameters. Otherwise your code will fail the test and be awarded 0. In particular remember *not* to remove any headers including those in the `StudentClass.java` file.

The "texts" that we will be searching are in fact DNA sequences. A *DNA sequence* or *genetic sequence* is a succession of letters representing the primary structure of a real or hypothetical DNA molecule or strand, with the capacity to carry information as described by molecular biology. The possible letters are A, C, G, and T (note they are all in upper case) representing the four nucleotide bases of a DNA strand: adenine, cytosine, guanine and thymine that are linked to a phosphodiester backbone. Sequences can be derived from the biological raw material through a process called DNA sequencing. This description is purely for background, you do not need to know it for the exercise, all that is needed is that our alphabet consists of A, C, G, T (and indeed this is only relevant to carrying out tests).

Your programing tasks are as follows.

1. In the `StudentCode` class, provide implementations of the methods
   - `public static int[] computePrexFunction(String P)` [10%]
   - `public static Queue KMPMatcher(String T, String P)` [15%]

   **Note:** Recall that the function $\pi : \{1, 2, \ldots, m\} \to \{0, 1, \ldots, m-1\}$. In §2 we used the convention that arrays start at index 1. However in Java they start at index 0. Since we are just computing the array that defines $\pi$ we now reduce each argument to the abstract function $\pi$ by 1. To be clear if `pi` is the Java array that gives us the values of the function $\pi$ then $\pi(i)$ is given by `pi[i-1]`. If we were implementing a Java method `Pi`, say, then we could hide this so that a call to `Pi(i)` looks up `pi[i-1]` and returns this as the value. However to avoid extra coding we just work directly with the array `pi`.

   Keep your code straightforward, follow the algorithms as outlined in §2.2 (taking care of the issue regarding array indexing). Note that in designing the matching algorithm we made the reasonable assumption that the pattern is not longer than the text. Your code *must* test for this and return an empty queue if the pattern it strictly longer than the text. Remember also that you *must not* use the supplied method `buildPrefixFunctionNaively` as part of your implementation and if you do then *you will be awarded 0 for the entire coding part.*

   Test your implementation by using the following methods in the `Matcher` class:

- `public static Boolean PrexFunctionTest(String P)`

  This checks for occurrences of the pattern `P` with itself as discussed for building the prefix function. Returns `true` if the list of occurrences (represented as shifts) is correct otherwise `false`. You will not get any other information.

- `public static void KMPMatcherTest(int t, int l)`

  Generates `t` random patterns of length `l` and finds their ocurrences in a fixed text using the method `KMPMatcher`. If all results are correct then it returns an appropriate message. Else it reports on the failure. Note that the integer parameters here and elsewhere must not be negative (an exception is raised otherwise).

Naturally the test procedures do not guarantee correctness. All we can say is that if the code fails the tests then there is something wrong. If it passes all tests then it is probably correct.

2. The aim of this part is to compare the runtimes for `naiveMatcher` with those for `KMPMatcher`. We would expect that as the pattern gets larger the differences would be more significant. Note however that there could be patterns for which `naiveMatcher` runs faster (you should think carefully why this can be the case).

   However we have a problem in trying to compare these two algorithms. The runtime of Naive-Matcher depends on both $n$ and $m$ while that of KMP-Matcher depends only on $n$. Of course for any fixed value of $m$ the runtime of Naive-Matcher is linear in $n$, recall that its runtime is $\Theta((n-m+1)m)$[1]. We would expect that for big enough fixed values of $m$ the advantage of KMP-Matcher would be clear (but note that if $m$ is close to $n$ then the runtime of Naive-Matcher is again linear in $n$). In order to avoid complications we will carry out experiments with the following choices of $m$ for any given value $n$: we use $m = 10, 10^2, \ldots, 10^r$ where $r$ is the smallest integer such that $10^r \geq \lfloor (n+1)/2 \rfloor$. The upper bound is chosen because $(n-m+1)m$ has its maximum value at $m = (n+1)/2$. The fact that we multiply the length of the pattern by 10 each time means that the number of patterns is not enormous.

   The supplied `Matcher` class method

   - `public static void getRuntimes(int p, int t, String f)`

   does the following:

   (a) Generates `p` patterns for each size from $10, 10^2, \ldots$ as discussed above using $n = 10000$ as the initial text size (the patterns are *not* random so that experiments are repeatable).

   (b) Searches for all occurrences of each pattern within a text of size $n$ using the methods `naiveMatcher` and `KMPMatcher`, and takes the cpu times for these.

[1] Do not make the mistake of concluding falsely from the statement about fixed values of $m$ that the runtime of Naive-Matcher is linear in $n$. This is clearly nonsense, e.g., consider $m = n/2$ in $\Theta((n-m+1)m)$.



   In fact the text is always taken from the initial portion of size `n` of the same longer text.

   (c) Records the worst case times for each search taken over the `p` patterns (note that it is pefectly possible that the worst case runtimes for the two algorithm implementations occur for different patterns).

   (d) Repeats the above with texts of size $20000, 30000, \ldots, 10000t$.

   (e) Outputs the result for each iteration on the file named `f` in the format

   *pattern-size text-size* `naiveMatcher-`*worst-case-runtime* `KMPMatcher-`*worst-case-runtime*

   For `f` you can give a path name but the simplest thing is to run code from your working directory and use just the name for `f`; the file will be placed in your working directory.

(Note that the data above are given on a single line of the file, the text is too long to fit on one line of this document.) Carry out experiments with increasing values of `t` until you find a clear difference between the two algorithms. In order to avoid excessive waiting choose the value of `p` to be fairly moderate. Naturally it makes sense to continue the experiment a little beyond the point of the first clear difference (indeed due to various factors there might be a temporary turn around, this doesn't happen for long).

Finally, run the experiment with `p=10`, `t=100` and keep the results in a file with the name `matcherTimes.txt`. Study the output to see what happens in terms of one algorithm outperforming the other. [5%]

Use this file to find the crossover point(s) at which `KMPMatcher` is consistently faster than `naiveMatcher`. This is a rough and ready way of obtaining an estimate for the settling in period before the overheads outweigh the benefits. In this case our approach is open to criticism (think about why this is the case) but is not too misleading.

From now on we will focus on `KMPMatcher`.

3. We know that the worst case runtime $T(n)$ of `KMPMatcher` satisfies $T(n) \leq cn$ for some constant $c$. Of course asymptotic notation allows for a settling in period, i.e., from some value $n_0$ of $n$ onwards. In fact this period is not really necessary at least from $n = 1$ onwards because we can just take a larger constant if needed (with the obvious downside that this gives a pessimistic performance guarantee for all large enough values of $n$). It should be fairly clear from the algorithm that the settling in period is not long; essentially we just need $n$ to be big enough so that $cn$ dominates the cost of initializing variables and setting up data structures.

   The supplied `Matcher` method

   - `public static void getRatios(int p, int t, int xOver, String g)`

is similar to `getRuntimes` except that for each experiment it only uses `KMPMatcher`. Once it has found the worst case runtime for a text size $n$ it records the value of that runtime divided by $n$. The results are output to the file `g` (use the name `matcherRatios.txt` for `g`) in the format

*integer-size* *ratio*

one per line as before. At the end of this output it also appends four extra lines:

```
Ignoring ratios before cross over point:  xOver
Sorted ratios are:  r1,r2,...
Maximum ratio is:  max-ratio
Average ratio is:  ave-ratio
```

Thus the final two lines give us the overall maximum as well as the average of all the ratios. The case for taking the average is that it smooths out to some extent the effects of any garbage collection. In a more detailed study we need to be more sophisticated but here we will keep things simple and rely on the plot (see below) to give us an idea of how useful the average is. You can also use the sorted ratios to get an idea of the effects of garbage collection (we would expect ratios affected by this to be significantly bigger than the "real" ones).

Use the following values for the parameters:

- `p` set to 10,
- `t` set to 100.
- `xOver`: use the crossover point (`TextLen` in `matcherTimes.txt`) you found where `KMPMatcher` becomes more efficient than `naiveMatcher`.

Finally you will plot the data from the file `matcherTimes.txt` of the previous part and the worst case runtime as determined by the theoretical analysis together with the two estimates for the constant found by the preceding experiment (the maximum and average). Use the supplied `Matcher` method

- `public static void plotRuntimes(double c, double a, String f)`

to produce your plot. For `c` use the maximum constant found above, for `a` use the average and for the file `f` pass the data in the file `matcherTimes.txt` from the preceding part. This will bring up a plot which you should save in a file called `plot.jpg`. [5%]

**Compiling and Running**

From the commandline, type `cd` *path-to-src-folder* and compile with the command `javac matcher/*.java`. Run the program with the command `java matcher.`*ClassName* where *ClassName* is the `.java` file where your `main()` target is.



**Notes**

1. Java provides pattern matching. You *must not* use this in your implementation. The point is to do things from scratch. *If you ignore this requirement you will be awarded 0 for the coding part.*
2. As stated above your code must be well laid out showing its logical structure. Just like good prose or good mathematical writing it must be set out to aid understanding. This way you and others can maintain it as well as spot any errors more easily.

## §4. Submitting the Coursework

You must submit the two parts of your work by the deadlines given at the head of this document. If you complete the coursework in full, you should be submitting:

**Part A:** Analysis of algorithms (both hardcopy).

1. Your answer to Exercise 1 of §2.1.
2. Your answer to Exercise 2 of §2.2.

**Part B:** Software.

1. Your code in the class file `StudentClass.java` (electronic and hardcopy).
2. `matcherTimes.txt` (electronic only).
3. `plot.jpg` (electronic only).

**Note:** Your answers to the two exercises of Part 1 are best handwritten (unless you have a medical condition that prevents this) and neatly presented following the guidelines of Lecture Note 2 (and the notes in general). You are required to submit a hardcopy printout of your Java software (`StudentClass.java`) so that the marker can put brief comments on it. We also need an electronic copy for automated testing.

Your hardcopy submissions should be made at the appropriate times to:

- ITO, Room FH-1.B15, Forrest Hill, 5 Forrest Hill, Edinburgh, EH1 2QL.

**Important:** Put your *matriculation number* very clearly at the top of your submissions. You do not have to put your name; marks will be returned to the ITO by matriculation number (this is how the `submit` command (see below) organises things. If you need to submit more than one piece of paper for a part please staple the sheets together and put your matriculation number on the top of each sheet (in case they get separated). Do not use folders to hold several sheets as this holds up the marking process significantly.

For electronic submission follow these instructions.

- First put your submission files (and *nothing else*) into one directory called `inf2b-cwk1`.
- Submit this directory using the following command (when logged into DICE):

```
submit inf2b 1 inf2b-cwk1
```

**Warning:** The rule for courseworks is "We mark what is submitted." Before you submit your coursework, make sure that you are submitting the correct files. In some previous years students submitted the wrong files and lost marks that way.

**Bibliography**


1. D. E. Knuth, J.H. Morris, Jr. and V.R. Pratt, Fast pattern matching in strings. *SIAM Journal on Computing*, 6 (2), 1977, 323–350.

2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*. McGraw-Hill, 2002.


*Kyriakos Kalorkoti*

*Software by Prachya Boonkwan, Xuan Huang and Sirkanth Ronanki*