EX6

1. Introduction. This is a literate program which solves the problem set in lab six—Karp-Rabin pattern matching.

We will start with the, by now familiar, outline of the program. This time we will not use global data and we have no functions other than *main*.

⟨ Headers 4 ⟩
using namespace std;
⟨ The main program 2 ⟩

2. Main. OK, let's start writing *main*. The skeleton of the *main* program is as follows.

```
⟨ The main program 2 ⟩ ≡
  int main( )
  {
    ⟨ Variables of main 3 ⟩
    ⟨ Open and the input file 5 ⟩
    ⟨ Store the target and search strings 7 ⟩
    ⟨ Calculate the initial hash values 13 ⟩
    ⟨ Perform the Karp-Rabin search 16 ⟩
    ⟨ Finish and clean up 18 ⟩
  }
```

This code is used in section 1.

3. The first thing we need to do is declare the variables we need to input data. Let's start with the character array *filename* and the input stream *fin*.

```
⟨ Variables of main 3 ⟩ ≡
  char filename[20];
  ifstream fin;
```

See also sections 6, 8, and 14.

This code is used in section 2.

4.

```
⟨ Headers 4 ⟩ ≡
#include <iostream>
#include <fstream>
```

See also section 9.

This code is used in section 1.

5. Right—now we can get the file opened, ready for input. We will prompt for the input file name using *cerr* so that we can redirect the output without getting the prompt in the output file and so that we can see the prompt even when we redirect standard output. We will then read in the file name and open an input stream. We should test for errors too, I guess.

```
⟨ Open and the input file 5 ⟩ ≡
  cerr ≪ "Please␣enter␣the␣name␣of␣the␣input␣file:␣";
  cin ≫ filename;
  fin.open(filename);
  if (¬fin) {
    cerr ≪ "Error␣opening␣file␣" ≪ filename ≪ ".␣Program␣will␣exit." ≪ endl;
    return 0;
  }
```

This code is used in section 2.

6. Before we start reading the file we need to declare the data structure we are going to use to store our two strings. The specification states that the search string, S, is no longer than 10 characters and the target string, T, is no longer than 5000. I will declare two integer constants S_MAX and T_MAX with these values.

It should be noted that, while changing T_MAX will not cause problems, increasing T_MAX may not be so straightforward. The reason for this will be explained later, as well as a remedy.

```
⟨ Variables of main 3 ⟩ +≡
  const int S_MAX = 10;
  const int T_MAX = 5000;
  char S[S_MAX];
  char T[T_MAX];
```

7. Store values in the table. We are now ready to read in our strings. we will read them both in the order T followed by S and calculate their lengths.

⟨ Store the target and search strings 7 ⟩ ≡
  $\mathit{fin} \gg T \gg S$;
  $\mathit{sLen} = \mathit{strlen}(S)$;
  $\mathit{tLen} = \mathit{strlen}(T)$;

This code is used in section 2.

8. We need to add these to our variables.

⟨ Variables of main 3 ⟩ +≡
  int *sLen*, *tLen*;

9. We also need the include file for *strlen*( ).

⟨ Headers 4 ⟩ +≡
#**include** `<cstring>`

10. Karp-Rabin Hashing. (An aside).

To start the process we need to calculate the Karp-Rabin hash values for S and the initial substring of T. in each case we do this by summing the result of multiplying each character of the string—or some representation of it—by a prime number raised to a power related to its position in the string.

That was confusing! Lets look at an example.

Consider the string "ACAG" which has a length of 4 characters. Its hash would be calculated as p^3`'A'` $+ p^2$`'C'` $+ p$`'A'` $+$ `'G'`, where p is a prime number that is greater than the integer equivalent of the largest element of the search string.

Choice of p is critical because we need to calculate large powers of p in our Karp-Rabin hash. If we have a string of length n characters, the first character will be multiplied by p^{n-1} and the sum will be $\in O(p^n)$. These are large numbers and it is not hard to overflow the storage capacity of an **int** data value.

If we take a naïve approach to the current problem we get the following analysis...

- Largest character = `'T'`, with ASCII value 84.
- Smallest prime $> 84 = 89$.
- Largest power of 89 that fits in an **int** $= 4$.

This means that, as soon as the search string gets to 5 characters in length we will overflow our 32-bit integer. Even if we use **long int** to store our hash values that will still overflow at 9 characters.

There are two possible solutions:

1. Reduce the size of the prime number, p.
2. Reduce the size of the hash value.

The first of these approaches is computationally cheaper and is the strategy we will use in this program. The second approach is computationally more expensive so we will avoid it as, in this case, it will not be needed.

If, even after we have found the smallest usable prime, we still overflow an **int** variable we can still use Karp-Rabin. In this case we introduce a further variable *modulus* which we use to limit the size of the calculated hash values to ensure that they can never overflow our **int**. We choose *modulus* so that it is as large as possible but still small enough that any intermaediate value in our calculation cannot cause an overflow. In effect this means that the inequality *modulus* $* p <$ `INT_MAX` must be maintained. The idea being that, every time we mutiply something by p, we immediately reduce it modulo *modulus*.

In the following code I have introduced comments showing the modulus-enabled version of the calculations.

11. To reduce the size of p we must reduce the encoded value of the characters in our alphabet. In the present case we have only 4 different input characters, `'A'`, `'C'`, `'G'` and `'T'`. Thus, the size of our alphabet is 4.

If we can recode the input letters into the range $0 \dots 3$ or $0 \dots 4$ we can use the prime number 5 in our hash calculation. It turns out that we can do this efficiently by replacing the ASCII value of the character $S[i]$ with $S[i] \% 5$. This gives the following encoding:

`'A'`	ASCII value 65, after mod 0;
`'C'`	ASCII value 67, after mod 2;
`'G'`	ASCII value 71, after mod 1;
`'T'`	ASCII value 84, after mod 4.

12. With p set to the value 5 we can now handle strings up to 13 characters in length and we are safe!

Note that using other approaches to reducing the alphabet encoding may be computationally costly. For example, the following is a Bad Idea (tm).

```
switch (char) { case 'A': char = 0; break; case 'C': char = 1; break; case 'G': char = 2; break;
case 'T': char = 3; }
```

13. The initial hash. OK, Let's do it.

```
⟨ Calculate the initial hash values 13 ⟩ ≡
  for (int i = 0; i < sLen; i++) {
    sHash = sHash * p + S[i] % crunch;      /* sHash = (sHash * p + S[i] % crunch; ) % modulus */
    tHash = tHash * p + T[i] % crunch;      /* tHash = (tHash * p + T[i] % crunch; ) % modulus */
    power *= p;
  }
  power /= p;
```

This code is used in section 2.

14. We introduced a few new variables there. In order, *sHash* is the Karp-Rabin hash value for the search string S, *tHash* is the hash of the initial substring of T, and *power* is used to precalculate the value of p^{sLen-1} which we will need in the rolling hash phase. We haven't declared *crush*, the value we use to reduce the alphabetic encoding, or p, our prime number, so we should do that too.

```
⟨ Variables of main 3 ⟩ +≡
  int sHash = 0, tHash = 0;
  int p = 5, crunch = 5;
  int power = 1;
```

15. It just occurred to me that the way I calculated the initial hash values for S and T may not be entirely obvious. The technique used was something known as *Horner's Compact Evaluation.* This allows the computation of polynomial expressions in a very efficient way and works as follows:

Consider the polynomial:

$$P = \sum_{i=0}^{n} a_n x^n$$

a polynomial in x with $n + 1$ terms having coefficients $a_0, a_1, \ldots, a_{n-1}, a_n$.

If we evaluate this in the obvious way, term by term, we will need to perform many more multiplications than is strictly necessary. As an alternative we can rewrite the polynomial in a nested form as follows:

$$P = a_0 + x(a_1 + x(a_2 + (... + x(a_n)...)))$$

which takes only n rather than $O(n_2)$ multiplications.

The loop in our program evaluates the hash values using exactly this scheme, evaluating the innermost term first and working outwards.

16. Rolling, rolling, rolling. Now we can proceed with the actual search. We do this by comparing the hash code *sHash* with the hash codes of each substring of T with the same length as S, i.e. *sLen*.

To calculate the next *tHash* value we simply:

- subtract the contribution of the first character from *tHash*;
- multiply this by p;
- add the contribution of the new character at the end of the substring;

```
⟨ Perform the Karp-Rabin search 16 ⟩ ≡
  for (int i = 0; i < tLen − sLen; i++) {
    if (sHash ≡ tHash) {
      ⟨ Brute-force compare S with T [ i..1 + sLen − 1 17 ⟩
    }
    tHash = (tHash − power ∗ (T[i] % crunch)) ∗ p + (T[i + sLen] % crunch);
      /∗ tHash = ((tHash − power ∗ (T[i] % crunch)) ∗ p) % modulus + (T[i + sLen] % crunch); ∗/
  }
```

This code is used in section 2.

17. The brute-force comparison is easy.

```
⟨ Brute-force compare S with T [ i..1 + sLen − 1 17 ⟩ ≡
  bool same = true;
  for (int j = 0; j < sLen; j++) {
    if (S[j] ≠ T[i + j]) {
      same = false;
      break;
    }
  }
  if (same) cout ≪ "Match␣found␣starting␣at␣T[" ≪ i ≪ "]" ≪ endl;
```

This code is used in section 16.

18. We should also close our input file.

```
⟨ Finish and clean up 18 ⟩ ≡
  fin.close( );
  return 0;
```

This code is used in section 2.

19. Index. This index is automatically created. It lists all the variables used in the program and the section(s) in which they are used. Underlined entries indicate where a variable is defined. The remaining sections of this document are also created automatically.