

Pattern Searching in a Single Genome

Abigail Linton
Bartholomew School

Working with
Oxford Centre for Genome Function

| | |
|--|-----------|
| ABSTRACT | 3 |
| 1. INTRODUCTION | 4 |
| 1.1 THE PROJECT BRIEF | 4 |
| 2. METHODOLOGY – EXACT SEARCHES | 5 |
| 2.1 THE NAÏVE SEARCHING METHOD | 5 |
| 2.2 THE RABIN-KARP ALGORITHM | 7 |
| 2.3 THE KNUTH-MORRIS-PRATT ALGORITHM | 8 |
| 2.4 APPLYING THESE ALGORITHMS TO CHROMOSOME 21 | 9 |
| 3. METHODOLOGY – PAIRWISE ALIGNMENT..... | 10 |
| 3.1 SUBSTITUTIONS ONLY | 10 |
| 3.2 GLOBAL ALIGNMENT | 10 |
| 3.2.1 GLOBAL ALIGNMENT – FIRST EDITION | 11 |
| 3.2.2 GLOBAL ALIGNMENT – SECOND EDITION | 13 |
| 3.2.3 GLOBAL ALIGNMENT – THIRD EDITION | 13 |
| 3.2.4 GLOBAL ALIGNMENT – FOURTH EDITION..... | 13 |
| 3.3 LOCAL ALIGNMENT | 14 |
| 3.3.1 LOCAL ALIGNMENT – FIRST EDITION | 14 |
| 3.3.2 LOCAL ALIGNMENT – SECOND EDITION | 14 |
| 3.3.3 LOCAL ALIGNMENT – THIRD EDITION | 15 |
| 4. METHODOLOGY – OTHER WAYS OF SEARCHING | 16 |
| 4.1 POSITION WEIGHT MATRICES | 16 |
| 4.1.1 INVERSIONS WITH PWMs | 17 |
| 4.2 COMPARING POSITION WEIGHT MATRICES AND LOCAL ALIGNMENT | 17 |
| 4.3 SUFFIX TREES | 19 |
| 4.3.1 SUFFIX TREES – EXACT SEARCHING | 19 |
| 4.3.2 SUFFIX TREES – SEARCHING WITHOUT A PATTERN | 20 |
| 5. RESULTS AND DISCUSSION | 21 |
| 5.1 EXACT MATCHING IN CHROMOSOME 21 | 21 |
| 5.1.1 FURTHER ANALYSIS | 23 |
| 5.2 LOCAL ALIGNMENT AND POSITION WEIGHT MATRICES | 25 |
| 5.3 SUFFIX TREES | 27 |
| 6. CONCLUSION..... | 29 |
| 7. EVALUATION | 30 |
| 8. APPENDIX | 31 |

KNUTH-MORRIS PRATT DIAGRAM 31
8.1 PROGRAMMES 32

9. REFERENCES.....37

10. BIBLIOGRAPHY37

11. ACKNOWLEDGEMENTS37

Abstract

Pattern searching holds much importance for biologists, for the understanding of DNA (and its functionality) can be more than a matter of satisfying curiosity, but also give answers to many issues such as medical conditions. However, there are a number of ways of searching within a single chromosome, and the chosen method could depend on the specific task at hand, as well as the efficiency of the programme (how fast it is, how much memory it requires) in respect to the computer system being used.

1. Introduction

Pattern searching is a problem that can arise in many different areas, from finding text in a document, to compressing data. However, it is also very important in biology, and in looking at DNA sequences; it can lead to being able to search for particular patterns in DNA (referred to as “motifs”), and perhaps to align the sequences in order to allow comparison within, and between, species. This could help in a variety of areas, for example in finding a common ancestor to species by comparing how similar their DNA is.

This could sound like a relatively simple task, however the sheer complexity of it may not be realised: the quantity of data is a very important factor. The haploid human genome (one half of the whole human genome) alone contains – fundamentally – 23 pieces of text, all together with length of about 3×10^9 (where each nucleotide represents one piece of text: one of A, C, G, T). This is a very large amount of data to search through, and thus could take a long time, and require a lot of processing power.

For this reason, finding an effective and efficient pattern-searching algorithm is very important for the understanding of genomes. However there must be many different forms of pattern searching. For example, some signals may be known, but others may only be partially known, or subject to containing errors, such as substitutions (mutations) and insertion/deletions (when parts of the text are removed and/or new text is inserted. Nonetheless, these types of error are quite different, and sometimes it can be preferable to allow for one and not the other. Furthermore, the whole sequence may not be known at all, and therefore it could be necessary to simply search for common patterns within the text, that is not already known; however there is again the same problem with errors, and how to/which to account for.

Another important aspect to pattern searching in DNA is to ensure that the number of occurrences of a pattern is more than the number of occurrences in a random string of the same length. This would make the pattern statistically significant, and thus help to be sure of it as a motif.

| | | | |
|-------------|-------------|--------------|--------------|
| CGTA | CGTA | C-GTA | CTGTA |
| CGTA | CGGA | CCGGA | CCGGA |

$$W(p1, p2) = \sum_{i=1}^5 w(p1[i], p2[i])$$

In general four cases can be described: i. we look for a perfect match. ii. we allow errors due to substitutions iii. We allow errors due to substitutions and insertion-deletions. iv. We rank the possible matches according to a weight function and keep matches above a certain threshold or the best match. $p1$ and $p2$ are here the two patterns of length 5, W the weight of the complete patterns defined via a nucleotide-nucleotide weight function $w(,)$.

Professor Jotun Hein, 07/09

Fortunately, there have been many methods devised for solving these problems. There are exact searching methods, for when no errors should be accounted for; there are methods to search using position weight matrices, which allow common errors; and there are suffix trees for finding recurring patterns.

1.1 The Project Brief

Within this project, the idea was to be able to implicate some different methods of searching for a string within a longer text string. This would involve looking at methods that have already been established (such as the naïve searching method, Knuth-Morris-Pratt, and Rabin-Karp), and perhaps adapting these slightly for the purposes of this project. Therefore, it could be possible to test for exact matches with motifs in Chromosome 21, and by adapting these it could be possible to begin allowing errors.

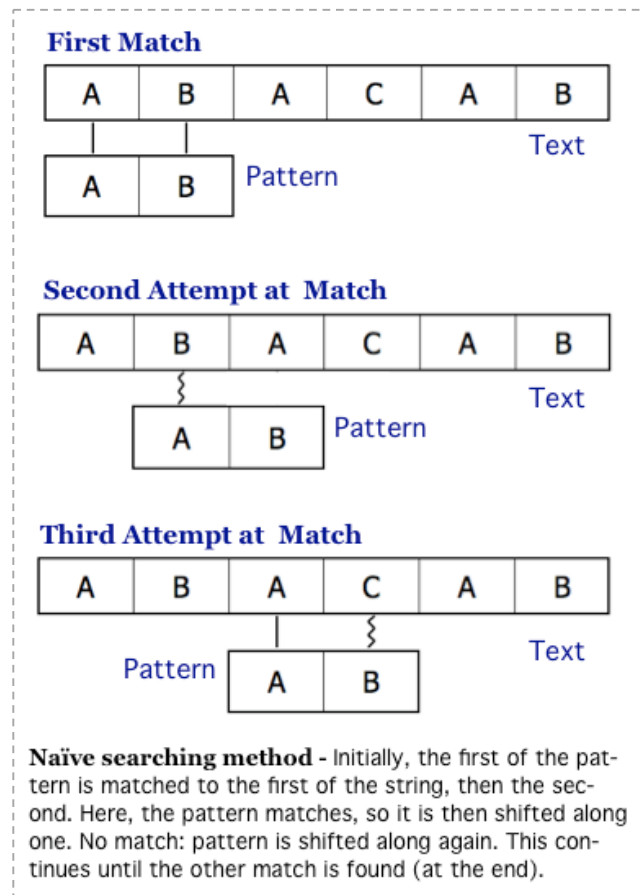
Further, the naïve method could be developed and used as a basis for alignment techniques, while position weight matrices and suffix trees would be more advanced ways of looking at the problem.

2. Methodology – Exact Searches

There are many algorithms that search for a keyword, or pattern (string), in a longer word/pattern/string (“string” is the term for a sequence of characters referred to by another character, or word etc. For example: string = “sequence of characters”; or s = “sequence of characters”).

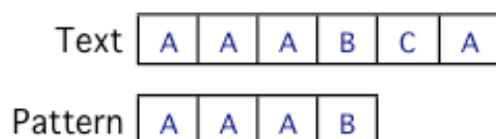
2.1 The naïve searching method

The naïve method for string searching works by checking each element both of these strings (although, there is no point searching the last characters of the text, when the pattern cannot be fully aligned due to its length – therefore the naïve method actually checks the $n-m$ positions (for the start of the pattern), where “ n ” is the length of the string and “ m ” is the length of the pattern). It checks the first character of the pattern against the first of the text; if there is a match, the search continues with the second character of each string, but if there is no match, the pattern is shifted along the text, and the attempt to match begins again (with the first of the pattern against the second of the text). This process continues until either the whole pattern is found in the text string, or until it has come to the end of the text string (actually, the ‘end’ described here would be the $(n-m)$ position, as explained earlier). After finding an entire match with the pattern, the whole process begins again, at the next position in the text.



Therefore, the worst case running time of this algorithm is achieved when each of the characters in the pattern must be compared to each of the characters in the text, except for the last few of length of the pattern – for example, when both the pattern and text are comprised of only one letter (such as searching for “aaaaa” in “aaaaaaaaaaaaa”). In this case, it is – as expected – quite long: $\Theta((n-m+1)m)$. [“ Θ ” denotes the worst-case running time, and “ n ” and “ m ” are again the lengths of the text and pattern respectively.] This means that the running time is effectively proportional to the length of the text multiplied by the length of the pattern. This time is the total amount of time for the matching process to take place, as there is no pre-processing required (i.e. nothing has to be done in order for the matching to begin, unlike in some other algorithms).

The naïve string-matching algorithm should take considerably longer than other methods. It is so inefficient because it does not take note of the information it finds when matching for the first time, when matching subsequent times.



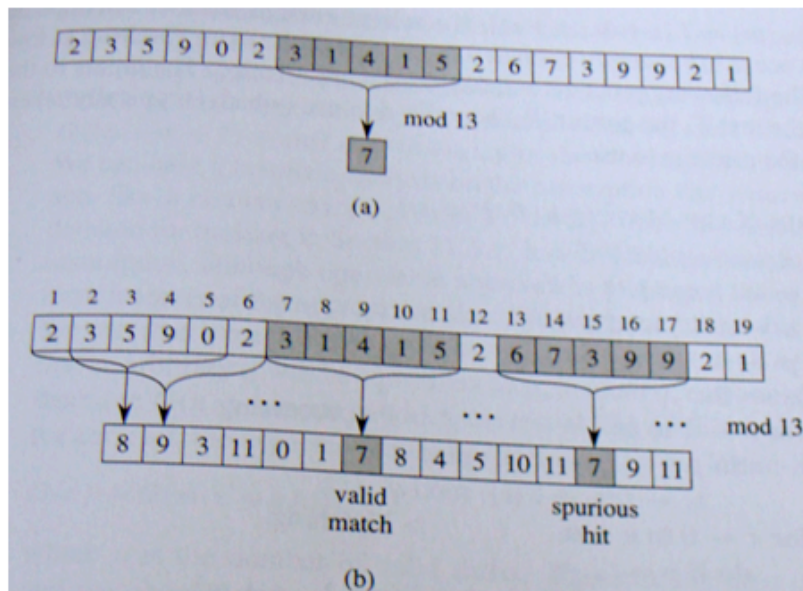
At shift 0, there is a match, so it is obvious that there cannot be a match at shift 1, 2, or 3 because $T[4]$ is “B” and no other value in P is “B”. However, the naïve algorithm will search the next few shifts anyway.

Nonetheless, there are a few different ways that the naïve string matching method can be implemented. Firstly, there would be a “for” loop for all of them, which would be sufficient to increment the position in the text that is being looked at and pass this position to the function that will search in that position. The three different ways come into play in the actual searching, and determining whether there is a match or not (“True” or “False”). One method should be faster, where a slice from the current position to the length of the pattern is compared to the pattern. Here, python is checking for itself, so the programme is very short, but it is more efficient. Other methods include actually determining the conditions for a true and false (and writing the programme that tests each character against each other character). In this sense, two methods could arise from testing the whole pattern against the text, or stopping the comparison of the pattern as soon as there is no match. This would obviously cause a difference in time required to run these different methods, though they would still be proportional to the length of the pattern and string, and thus would still have the same worst-case running time (as shown above).

2.2 The Rabin-Karp algorithm

Rabin-Karp operates in a different way; here, the characters are assigned numbers (for example from zero upwards, possibly producing a binary set of numbers (if there are two characters, where one is assigned a 0 and the other assigned a 1), or tertiary, or decimal etc. This is the radix (or base), and can be found in different ways, such as converting the characters into ordinal numbers (simply the ASCII code, which is basically a set of numbers denoting characters) and calculating the highest minus the lowest; or putting the characters in a dictionary and taking the size of the dictionary. After the assignment of the pattern, slices of the text that are the same length as the pattern are also assigned a number; however, these could be very large numbers if working with a large radix and a long pattern, so a modulus is taken to reduce it. This modulus ("q") is generally a prime number, which – when multiplied by the radix – is able to fit within one computer word (which may vary depending on the computer, though in many cases it could be about four bytes long).

The algorithm subsequently compares these moduli. This could sometimes lead to a spurious match, where the slice of the text is not the same as the pattern, but the assigned values (modulo q) are equivalent. Therefore, the programme also compares the actual slice and pattern, in the same way that the naïve method does so, but otherwise the difference in the modulo q is an indication of a mismatch, and the next position is tested.



(a) Converting section into modulus - A section of the text (converted into numbers) of the same length as the pattern is taken as a number (31,415), and converted into a smaller number (by taking the modulus).

(b) Finding matches - Sections of the text are iterated through, and produce different moduli. A spurious hit is where the modulus is the same as that of the pattern, but there is not a match. This section would then be compared to the pattern, using the naïve matching method.

Taken from: T H. Cormen, C E. Leiserson, R L. Rivest; **Introduction to Algorithms**

The worst-case running time of the Rabin-Karp algorithm is also similar to that of the naïve algorithm ($\Theta((n-m+1)m)$), because it must process each valid shift. This could occur where there are always spurious hits, and therefore each time this slice must be compared to the pattern (in a similar way to the naïve matching method. However the average running time is

much better, particularly if the “q” is chosen to be a large enough prime number, so as to reduce the number of spurious hits.

However, when this algorithm was tested, it was found that it was in fact very slow. By looking at the computer’s memory usage during the running of Rabin-Karp, it could be noticed that required more memory than the other algorithms – to an extent to which the computer was even writing to the hard drive, which would ultimately slow it down. Therefore, the Rabin-Karp algorithm was adjusted to not store as much information and have fewer operations. For example, the radix is calculated in advance (and stored as a number, rather than calculating it for each iteration). Further, it was found that some operations were unnecessary, and appeared to reconstruct the string and pattern from a dictionary that was made from them. Nonetheless, when the results are taken, comparing the different algorithms, the original version is used.

2.3 The Knuth-Morris-Pratt algorithm

This algorithm (also known as KMP) originated from Knuth, Morris, and Pratt; it is one algorithm that is thought to be better than some other methods (such as the “naïve string-matching algorithm”) because it is more efficient (faster, thus better for a user). However, it may also be thought of as having a lower “worst-case” running time than Rabin-Karp as well, because KMP will test for a match in constant time, whereas the Rabin-Karp must check for the pattern at each shift, as well as the more extensive pre-processing time (even if it is only a simple mathematical process, rather than matching anything).

Further, it is also well renowned for being a complicated and difficult algorithm to fully comprehend, which makes it very difficult to explain to others, and thus also to learn. Nonetheless, this algorithm basically works by first defining a function (the “ π ” function, or the prefix function, as it looks at prefixes) to look at the pattern to be matched, and producing something that represents the length of the longest prefix (sequence at the start of the string), that can be found at the end of the sequence, up until the position in it that is being looked at. For example, if the pattern were “ababca”, the algorithm would produce the list of numbers “[0, 0, 1, 2, 0, 1]”, because when looking at the first character, a whole string cannot count as either a suffix or prefix, so the result is zero. Looking at the second character, the same thing applies, so also zero; the third character (“a”) can be found there, and as a prefix. Therefore, the value “one” is produced (as the character/s that can be found both at the start and at the end has length one). When the position holding the “c” is considered with the algorithm, this suffix does not appear previously in the pattern, so the value is zero again. The algorithm is actually looking at how much it can shift this pattern against itself (see diagram, in appendix).

Subsequently, the first character of the pattern is compared to the first of the text (longer string). If this is a match, the second of each string is compared, and it so continues until the whole pattern is found. However, if there is not a match, then the compared position in the pattern decreases (the whole pattern is shifted along, so that – for example – the first of the pattern is compared to the second of the text). This algorithm is faster because it is able to use the function (above) – indicating repetition in the pattern – to mean that the process does not need to keep going through to check the whole pattern, but refer to a particular position in string produced by the computed “ π ” function instead. Therefore, when it comes to shifting, the “ π ” function also allows an indication of how much to shift the pattern by, and still be likely to achieve a match with all the text before the position that is currently being looked at.

The worst case running time of KMP ($\Theta(n+m)$) is perhaps the best of the three discussed here, and it is reached because the pre-processing (the processing required before actually

searching) is proportional to the length of the pattern (as the pattern is all that is used to create the pi function), and the processing time (the actual searching) is proportional to the length of the text (which is used with the pi function). However, Rabin-Karp may come close to this if there are no false matches, but generally, KMP should be the fastest.

2.4 Applying these algorithms to Chromosome 21

Initially, small samples of chromosome 21 were taken from the UCSC Genome Browser (genome.ucsc.edu) and the algorithms above were applied; the times that each of them took were noted. These times allowed approximation of how long the whole chromosome could take with each of the algorithms, and subsequently the whole chromosome was tested using this same method. This was made simpler by creating a programme that would access a separate, shared file, containing the nucleotide sequence of the whole chromosome.

Further, code for instructing the printing of shifts was removed, in an attempt to reduce the running time's dependency on printing (which would be considerably higher if there were lots of matches, and thus many shifts to print). The measurement of timing was also made more accurate by using the "time" Unix (another operating system) command in the command line, which would save having to time it with a stop-watch, and thus immediately reduce the potential for human error, or variances in the time taken to put the results on the screen (the time was always given in terms of the "real" time, the "user" time, and the "sys" time. The "real" time was the one that was recorded, as it is more representative of the total time used for the operation.)

Command-line entry:

```
time ./DNATestNaive.txt Chromosome21.txt TATAAA
```

Printed result:

```
Number of matches: 26122
```

```
real    0m59.004s
user    0m57.647s
sys     0m1.331s
```

Each algorithm searched the whole of chromosome 21 for a number of different motifs. This was in an attempt to compare the running time for different lengths of pattern, where the faster algorithms would probably overtake the others in terms of speeds, when - for example - the running time of the naïve method is $\Theta((n-m+1)m)$, but KMP is $\Theta(n+m)$, and thus should increase less, with increased pattern length.

3. Methodology – Pairwise Alignment

The term “pairwise” means that only two sequences are being aligned. Sometimes, it is useful to align many DNA sequences, which could allow similarities between certain species to be found; however, the more sequences involved in the alignment, the more the running time is increased, therefore it would be simpler to use only two sequences.

Alignment is a way of representing the relationship of two sequences, where evolutionary events (insertions/deletions and mutations) are considered. It is basically another way of pattern searching, although it allows the possibility of errors, but is representative of the fact that DNA is not always conserved from one member of a species to its offspring: fertilisation causes genes from the parents to be swapped and alternated (so sections of DNA may be inserted into, or lost (deleted) from a sequence from either original parent). Further, there may always be random mutations from one generation to another. None of these factors could help biologists to liken one DNA sequence to another, and thus it would be more difficult to trace species back to a common ancestor. Therefore, it is important to be able to make matches that are not necessarily exact matches, such as those that were being searched for previously.

3.1 Substitutions only

Firstly, this task was only attempted with substitutions (substituting one character for another, i.e. a mutation). Doing this, it was possible to set a limit of how many substitutions should be made, on the basis of how exact the match should be. This was tested with some shorter strings, but then also used with the whole of chromosome 21, being implemented in a very similar way to the exact-searching algorithms.

It works by using the naïve-matching algorithm to check for a match, but if there is no a match,

```
total matches are 402190
number of mismatches: 0 ; number with this many: 26122
TATAAA 26122
number of mismatches: 1 ; number with this many: 376068
TAAAAA 45830
AATAAA 43254
TTTAAA 42772
TATATA 34335
TAGAAA 21360
TACAAA 21276
TATAAT 18938
TATTAA 18458
TGTA AA 17394
CATAAA 17061
TCTAAA 14909
TATACA 14526
GATAAA 14072
TATGAA 13928
TATCAA 10721
TATAGA 10179
TATAAG 9450
TATAAC 7605
```

The result produced when searching for “TATAAA” in chromosome 21, but allowing up to one substitution: the matched strings are ordered according to the number of matches.

the programme adds to a count of mismatches, and – if this exceeds the specified limit (that the user provides) – the programme discounts it as a match, and thus shifts the pattern along. The actual version of the pattern that was matched was then printed along with the shift at which it occurred.

This basic version was then modified so that it would print the number of exact matches (where no substations occurred), followed by the number of matches with one substitution, then the second, etc. until the limit. Furthermore, it also was set up so that it only printed the matched sequence once, and ordered these by the number of matches with the particular string.

3.2 Global Alignment

Global alignment is the aligning of two sequences: the whole of the pattern is aligned to the whole of the text string. This task involved creating a programme that would search for matches, allowing insertions and deletions (or, gaps in either the pattern or text). This began with simpler algorithms, which could lead towards a more efficient algorithm (shown in local alignment). Therefore, the first programme that was developed worked by having scores set for a match (positive), gap (negative), or no match (negative). From these scores, an optimum

alignment could be found by adding the scores together and taking the highest. This would mean that different alignments could occur, depending on the scores chosen – hence why techniques used in practice tend to use statistical methods, which allow these scores to be estimated from the data, or there are more advanced models which can estimate other things such as the insertion/deletion ratio, or the expected length of insertions. Therefore, this reduces the responsibility of the human user; however this would be very complicated, so scores were just selected for this model).

3.2.1 Global Alignment – First Edition

The first attempt at this had scores for a match, gap, and no match. The basis for this programme could be thought of as a matrix (which is initially empty), of dimensions m by n (where “m” is the pattern length and “n” is the text length), containing the most optimum scores for alignment up to that point. The programme would use the scores for a match, gap and no match, in order to find the optimal route (as shown in the diagram below) by calculating the next score from one of the previous three (above, to the left, and diagonally upwards). The optimum alignment can be found in this way because it is a way of breaking the whole problem into many different sub-problems (a technique known as dynamic programming). In this way, the programme can begin by aligning no pattern against the first of the text, and the second, until it has aligned no pattern against the whole the text (resulting in many gaps), then perhaps the first letter of the pattern against the text, and so on. At each stage, the sub-problem is very similar in structure to the whole problem, and a score could be taken from any of the three previous alignments, up to that point.

| | | A | G | T | C | |
|---|----|----|----|----|----|----|
| | | 0 | -2 | -4 | -6 | -8 |
| A | -2 | 3 | 1 | -1 | -3 | |
| T | -4 | 1 | 2 | 4 | 2 | |
| C | -6 | -1 | 0 | 2 | 7 | |

Global Alignment - The text is along the top, the pattern along the side. The first box is 0, because the score for aligning nothing against nothing is 0, but an arrow down represents an insertion of the pattern into the text (so the text does not increase in length), an arrow right is an insertion of the text, and a diagonal arrow is either a match (if the characters match) or a substitution (if there is not a match). A gap (insertion/deletion) is -2, a substitution is -1 and a match is +3. A route must be found for the most optimum score (the highest score in the bottom right box). The optimum is 7, when the alignment is “AGTC” with “A-TC”, which appears intuitive.

Recursion was initially used, in the belief that it would be easier; therefore, the algorithm works by starting from the last position (where the seven is, in the diagram above), and running the same function again to find the other three that it requires, and again to find the three that those three each require. Each time an optimum is calculated, it is stored within the

matrix/table, and if it must be used again, the value from the table is used, rather than calling the recursive function repeatedly (for a value that is already known), otherwise it would require exponential time (see diagram below).

| | | A | G | T | C |
|---|----|----|----|----|----|
| | 0 | -2 | -4 | -6 | -8 |
| A | -2 | 3 | 1 | -1 | -3 |
| T | -4 | 1 | 2 | 4 | 2 |
| C | -6 | -1 | 0 | 2 | 7 |

| | | A | G | T | C |
|---|----|----|----|----|----|
| | 0 | -2 | -4 | -6 | -8 |
| A | -2 | 3 | 1 | -1 | -3 |
| T | -4 | 1 | 2 | 4 | 2 |
| C | -6 | -1 | 0 | 2 | 7 |

| | | A | G | T | C |
|---|----|----|----|----|----|
| | 0 | -2 | -4 | -6 | -8 |
| A | -2 | 3 | 1 | -1 | -3 |
| T | -4 | 1 | 2 | 4 | 2 |
| C | -6 | -1 | 0 | 2 | 7 |

Alignment using Recursion - Each position in the matrix could have been reached by many different ways, therefore the recursive function - which calculates the values - must be called many times, to calculate the same values again and again. For example, the value in the highlighted square (above) could be calculated a very large number of times (it would be 20 times, even without allowing the diagonal paths), due to the number of different paths to it from the last position. Therefore, it is important to store the information, to prevent a very long running time, in which unnecessary processing occurs.

This continues until the starting position is reached (which is set as being zero), and finally the calculations can begin and be traced back. Each time, the score in the box was found by doing the appropriate sum to the other three around it, and identifying the highest. For example, in the box (in the “Global Alignment” diagram above) which represents a match between “G” in the text and “A” in the pattern, the value could be obtained from above (an insertion), the left (deletion), or diagonally (substitution). [Whether it is an insertion or deletion depends on how the situation is to be looked at, for example which of the sequences is the ancestor, and therefore how the other sequence has changed relative to it. Here, the text is regarded as the ancestor, therefore coming from above results in the new sequence (pattern) missing something of the text, and from the left is having something in the pattern, that was not before in the ancestor (text).] If from above, the score would be -6 (-4-2), from the left it would be 1 (3-2), from the diagonal it would be -3(-2-1). Therefore, the obvious optimum is from the left, and so a gap rather than a substitution.

The function produces a matrix (representative of the grid above), which contains tuples (similar to a list, although tuples are unchangeable, and cannot be modified), which in themselves contain the text so far (possibly inclusive of dashes, representing a gap (insertion)), the pattern so far (also with gaps in (representing deletions)), and the optimum score of such an alignment. Once the function has finished running, the end tuple can be taken and split up, in order to be able to print the text, the pattern and the score.

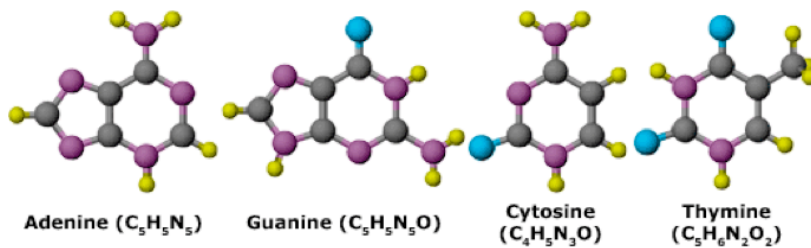
3.2.2 Global Alignment – Second Edition

This programme was then modified, in a way that could help achieve the final goal of local alignment, and further generalisation.

For example previously, all substitutions were given the same score (as “no match”). However, this is not particularly realistic: T and C are much more likely to mutate into one another than into either A or G, similarly the same is true the other way around. This is because the structure of T and C are very similar to one another, as are A and G, where they are either made of one ring or two (see diagram). Therefore, an adjustment to the last programme could be a matrix that consists of the scores, and would be the look-up source for the programme to obtain the scores from (see diagram for example).

| | A | C | G | T |
|---|----|----|----|----|
| A | 10 | -5 | -2 | -5 |
| C | -5 | 10 | -5 | -2 |
| G | -2 | -5 | 10 | -5 |
| T | -5 | -2 | -5 | 10 |

Using a Matrix for scores –
Matches are worth 10, substitutions of bases similar in structure are slightly negative, and more unlikely substitutions are more negative.



Taken from: *Understanding Science*; http://undsci.berkeley.edu/article/0_0_0/dna_02

Nucleotide similarities - C and T, A and G are much more likely to mutate into each other, rather than A into C. This is because A and G have double rings (are Purines) and C and T have single rings (are Pyrimidines).

3.2.3 Global Alignment – Third Edition

This time, the programme was modified so that it could overcome the problem of memory and running time. In order to combat this and reduce the memory usage and running time, the strings were not stored as the recursion progressed (before they were in the tuples, however each possible alignment had to be stored this way); if tuples were to be used like this, and the global alignment were to be applied to long strings such as whole chromosomes, the text strings would be extremely long, and the algorithm would take a very long time to operate due to the amount of data that must be stored.

Therefore, two matrices were produced from the function: one contained the optimum scores, and the other contained information to enable the trace back, and thus formation of the two strings. This latter matrix used numerical representations of direction; for example, travelling right was “0”, diagonally down was “1”, and down was “2”, therefore one of these numbers would be stored in each position in the matrix, where that particular direction resulted in the optimum score (when adding the optimum score to one matrix, the direction this score came from was added at the same time - in the form of the number - into the other matrix).

From this matrix, the two strings can be reconstructed, following the instruction of the direction matrix. This works in a similar way as the construction of the aligned sequences worked before, except now it is in another function and would leave all of it to the end: there are two empty lists, which are filled with the characters (either from the text and pattern, or dashes) according to the matrix.

3.2.4 Global Alignment – Fourth Edition

Previously, recursion had been used as it seemed to be the method that was easiest to implement. However, when working towards the goal of a fast local alignment, recursion would not be the easiest way of completing this goal. Recursion and iteration (going through line by line, position by position) can generally both be used for the same tasks, although to reduce

the memory usage (and thus the running time) it would be useful to traverse the matrix (row by row or column by column) so that only a couple of them have to be remembered at any one time (rather than use the memory to remember a four-by-47,000,000 matrix). Therefore iteration could be the most sensible option to use, with that aim in mind (though recursion could be used to fill in the rows or columns) – nonetheless, recursion was a reasonable initial idea for the previous alignments.

Therefore, there were still the two matrices (one of the optimum scores, the other of the direction), but these were constructed using “for” and “while” loops – rather than calling the function repeatedly – and the matrix is filled in row by row. Similarly, the function constructing the text and pattern as lists - from the matrix instructing the direction - is not recursive, but works by subtracting one from the length being looked at within a “while” loop (so that the loop will iterate through the length of the pattern, and lookup the direction from the matrix).

3.3 Local Alignment

Global alignment will find the most optimum way of aligning the whole of two sequences. Local alignment differs because it only aligns the pattern with a section of text, which applies most (and gives an optimal score) to the pattern.

Local alignment could be more useful in the case of printing the alignment of a short string against a long string (as this task demands), and again there were several attempts made in order to achieve an effective programme. However, all “N”s must be removed from the DNA (they occur where the sequencing is unknown), to prevent problems and errors occurring. This should also have to happen in order for the global alignment to work too, except this was never run on long strands of DNA, because it would be pointless – so many dashes, and considerable memory usage for storing strings, so it would be slow, and the computer would struggle to find this space. They must be removed because they are not in the dictionary, and not preferable to include in the searching (as they tend to be large blocks of “N”s, rather than a single one in the middle of text, therefore matches are not going to occur where “N”s are anyway).

3.3.1 Local Alignment – First Edition

This edition built on some of the earlier versions of global alignment, but the process was iterative and now went through the pattern-text matrix in columns, rather than rows, and only two columns were processed (and stored) at any one time. Each column is stored as a list (an empty list - made to be the length of the pattern plus one, the same length as the side of the original matrix used before - so as to reduce the amount of processing time). As the algorithm progresses, a tuple is added into each position of the current list (the other list represents the previous column, thus meaning that the squares above, left, and diagonally up can still be used to calculate the new score). This tuple contains the aligned strings so far, with the score for such an alignment, and the strings are constructed in the same way as before: the scores of the three around are compared, and give a value for the source (direction). Using this direction, the strings (by adding the subsequent character) can be constructed from the previous tuples. When the column is complete, the next column is completed with reference to the previous one. The way that the overall optimum local alignment is stored, is in another tuple that contains the score, with the strings that achieve that score; this tuple is rewritten whenever there is a higher score. This is the tuple that is returned from the function, and finally printed.

3.3.2 Local Alignment – Second Edition

For this version of implementing local alignment, there was no saved optimum that gets replaced each time there is a higher score. This edition was an attempt to resolve the issue that the previous version had: only one alignment was printed, and this was – most often an exact match. The only information that was given was the fact that “TATAAA” aligned with

“TATAAA”, which could not always be particularly useful; therefore, this time there is a certain threshold score set. Anything that scores above this threshold, causes the tuple (containing the two strings and the score) to be added to a list. When the whole text has been searched through, this list is returned, and the alignments can be printed by accessing the elements of the list.

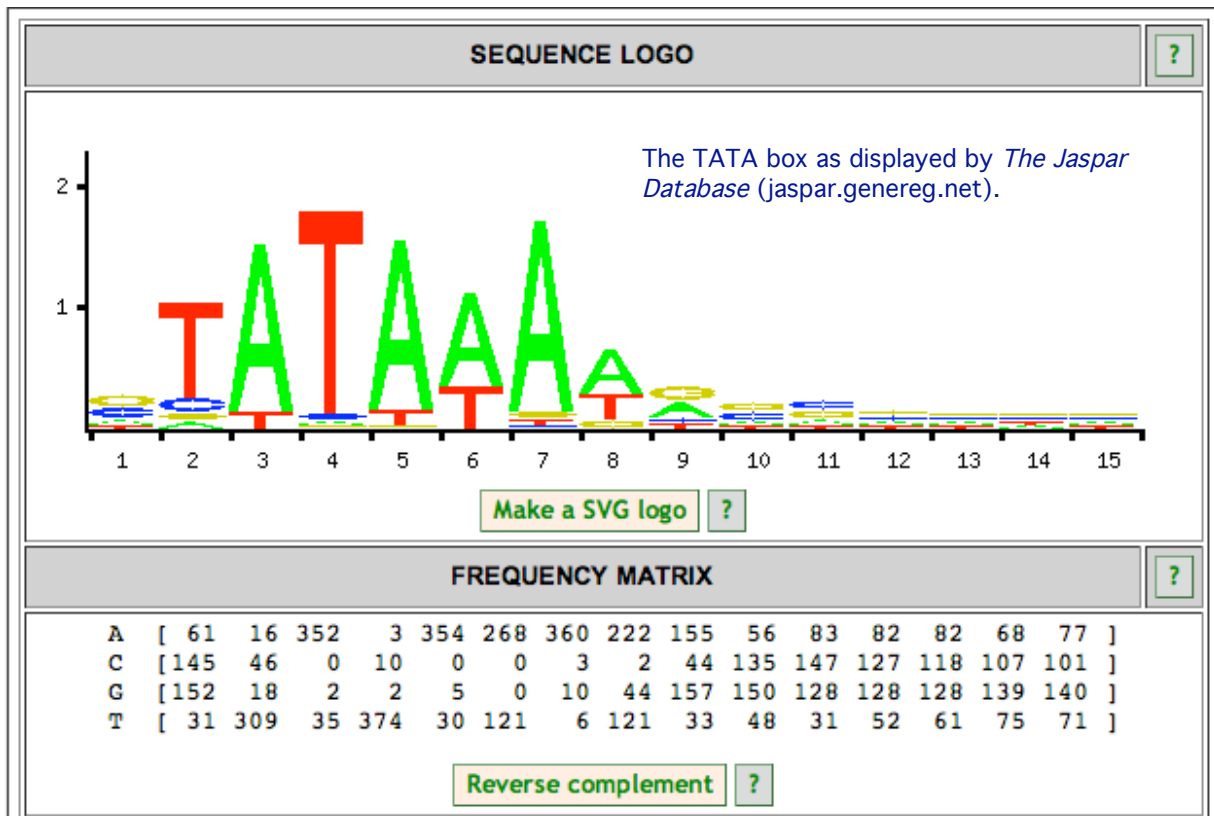
3.3.3 Local Alignment – Third Edition

This implementation was done in an attempt to solve the problem of efficiency. This edition is the same as the previous except for the fact that the whole strings are not saved (which could be a lot of memory if there are many matches and/or a long pattern). Instead, only the position of the text - where the alignment begins - and the score, are stored. In each position down the column (length of pattern), the source of the optimum determines which starting position is kept, in order to maintain an optimum score. Again, a limit is set, which saves all of the alignments that achieve above a certain score. When printing, a slice of the text (the same length as the pattern) is taken using the stored starting position from the saved list of tuples (containing starting position and optimum). This is faster, but does not align this part of the text with the pattern (because it cannot place the dashes (gaps)), but it does indicate which places can be aligned.

4. Methodology – Other ways of searching

4.1 Position Weight Matrices

This is a way of enabling scores to be given to how exact the match is. When looking at DNA motifs, it is very rare for an exact match to be found. For example, when proteins bind DNA, there are particular places that they will bind to; however, sometimes there will not be these exact matches due to mutations etc in a species, similarly, perhaps there is not an exact match required, and alternative matches are permitted. When these binding sites are discovered, it is noted how it is not always the same pattern that is present, and thus a matrix of the probabilities for each position are created in addition to the motif.



For example, this is the TATA box, which has previously been used as “TATAAA” in exact searching. In the image of the sequence, the size of the letter represents how probable it is to be found in a DNA sequence, or rather how many times it has been found when this pattern occurred in a sequence. The actual values of this frequency are shown in the matrix (often referred to as a Position Frequency Matrix (or PFM)), and these matrices were placed in a separate file, so that it could be easier to change the patterns used, without having to type the new values into the programme manually (instead, the programme was made to read the matrix and transform it into a format that could be used, by reading the file line by line, and creating a new matrix which can be used column by column). The PFM means – for example, the first column – out of the 389 (61+145+152+31) times that this sequence was found, the first position contained an “A” 61 times, a “C” 145 times, a “G” 152 times, and a “T” 31 times. Therefore, when searching for this pattern, the first position should be a “G”, with a high weighting for “C” as well, and with an acceptance for the other two letters too.

This matrix can then be used within the programme in order to search through the DNA sequence (text) and give scores, representing how closely that length of the text matches the patterns. This happens because the algorithm takes a character in the text string, and looks up the weighting for this character in the provided matrix. For example, if there is a “C” in the text at the current position, and this is to be matched to the first position of the pattern, then the probability of the “C” being part of the pattern is $152 \div (61 + 145 + 152 + 31)$, as looked up in the

matrix. Then the next position of the text is given a probability (for example, if it is a “G”, then the probability is $18 \div (16+46+18+309)$), etc, until each position of a section of the text (of same length as the pattern) has a probability. To find the overall probability of the pattern matching in that section, these individual probabilities must be multiplied together.

$$\text{Total probability of match} = \prod_{i=1}^n P_i$$

However, taking the product is impractical because if there is a long pattern (many probabilities), and each of the probabilities are low, then multiplying all of them together could easily result in a value that too small for the computer to handle, and thus produce a probability of zero. This is not a useful result, and deems comparison between different scores impossible. The product can be changed into a sum, by simply taking the log, and this also means that instead of a number such as e^{-500} , the score would actually be -500, which is much more comparable, and is not a gross estimation (where a small number is rounded to zero).

$$\log \left(\prod_{i=1}^n P_i \right) = \sum_{i=1}^n \log P_i$$

Therefore, the logarithm of each probability (it could be any base, here it was the natural log that was taken) must be taken. This is a relatively slow process, so it was pre-computed and placed into an equivalent matrix. Further, the natural logarithm of any number lower than one is negative, which could be a problem; it could be standardised by taking the highest possible score (H) for a match and the lowest possible (L), and using it with the actual score for the match (S): $((S-L) \div (H-L))$. Standardising like this means that the highest score is one (and the lowest is zero), so when getting the overall score, exact matches will score “1.0”. Additionally, a limit could be placed on the score, so that – when printing and storing (remembering) results – only the matches above a certain score would be counted.

4.1.1 Inversions with PWMs

Looking for a sequence can also be slightly more complicated than looking for the exact match, or with some variations. Every pattern will have an inversion on the other strand of the DNA, which goes in the other direction, and is complimentary (so “A”s on one side will be “T”s on the other, etc). The specific pattern could be on either strand, and one of the strands would be what is given as the nucleotide sequence. Therefore, the inversion of the pattern should also be searched.

This could be achieved by keeping the same algorithm as before, but using the matrix (containing the logarithms) in a different way, so that the row for “A”s is used as the row for “T”s and each row is read backwards for the score. Therefore, if the pattern is “ACCGT”, then an exact match could also be “ACGGT”.

4.2 Comparing position weight matrices and local alignment

Both programmes would have to be slightly adjusted in order to print the same results. The inversion version of alignment could not be used, because the alignment algorithms do not look for them either.

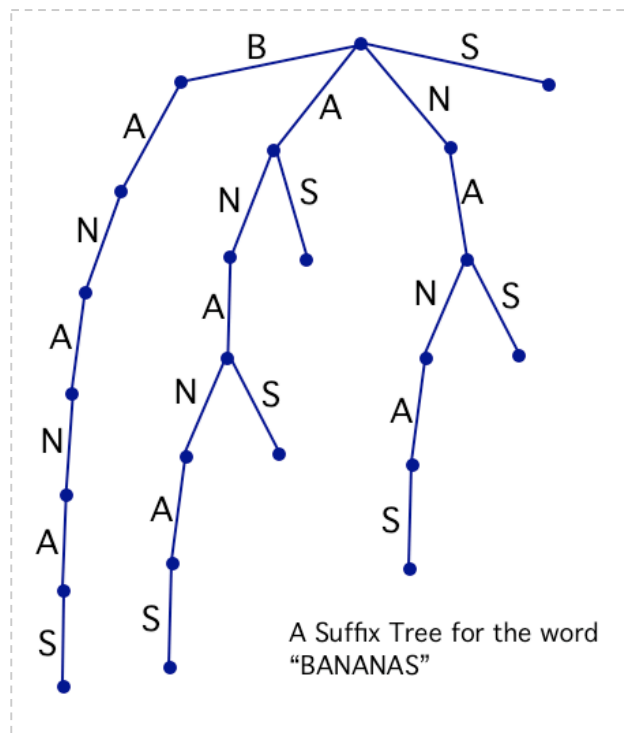
The MADs sequence that was used for exact matches was also used here because it is relatively long, quite distinctive, and had a frequency matrix available (from jaspar.genereg.net). Using this, the top scoring 30 matches of sequences produced by both could be compared, but for this the programmes had to have a dictionary that saved the sequence and the score of it (and the number of the individual sequences). In order to achieve 30 matches, the scores had to be

set to give only that many. Further, the scores for alignment had to be set (the scores for a gap, various substitutions, and matches), which could be quite an arbitrary aspect to this experiment. The scores were chosen to be 2 for a match, -2 and -3 for substitutions, and -2 for a gap. This was done so that they could be close to each other's scores, and so that when the allowance of the maximum substitution was given, this limit couldn't also be reached by giving many gaps, or minor substitutions. The time for this was also taken, using the same method as before (the "time" command).

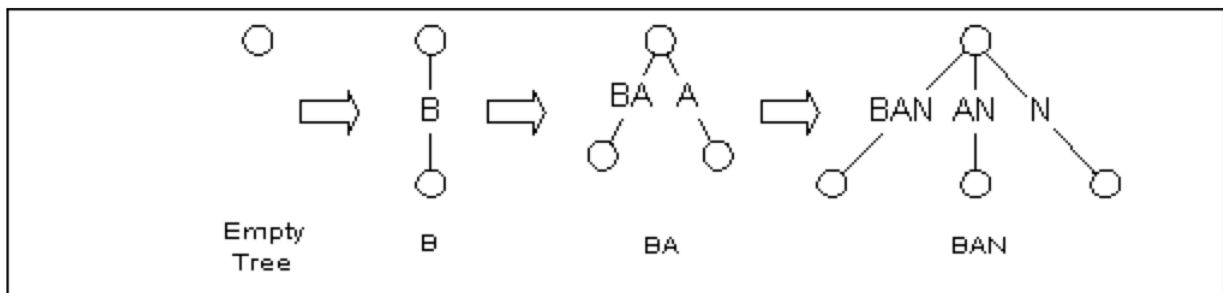
4.3 Suffix Trees

Suffix trees are another way of searching for a pattern in a text string, which could also potentially be quite fast and efficient. However, it is very different from the previous methods used to search.

Suffix trees (or sometimes, tries) can be visually represented with a tree (as shown in the diagram), with branches of strings and nodes stemming from a central point (the root), and ending in “leaf nodes” (the last nodes, that do not connect to anything further). The idea is that each stem from the main point contains a suffix of the word. Therefore, “BANANAS” is a suffix of the word “BANANAS”, as is “ANANAS”, “ANAS”, “AS” etc. Following upwards from each leaf node should represent each suffix, and the tree would be constructed by finding all of the suffixes that come particular letters (therefore the only letters coming from the root are “B”, “A”, “N”, and “S”, rather than having seven characters).



This method with the many nodes is a slightly basic and simpler version of a suffix tree. There are many algorithms for constructing (and using) suffix trees, and this is one of the more naïve approaches. An alternative is to condense the branches and reduce the number of nodes (and introduce things such as “implicit nodes”, which are nodes that become explicit when a new branch is required from it). This is the basis for algorithms such as Ukkonen’s algorithm, which is much more efficient than the more naïve method (but thus is also much more complicated, so the naïve was the implementation that was worked with in the different uses of suffix trees).



Building a suffix tree (with condensed branches, rather than a number of nodes)

Taken from: *Fast String Searching with Suffix Trees*; <http://marknelson.us/1996/08/01/suffix-trees/>

4.3.1 Suffix Trees – Exact Searching

The suffix tree is built with dictionaries where each node is a dictionary containing the subsequent nodes and strings. Therefore, there is one dictionary that is established first (as the root). The suffix tree is built by taking a slice of the text (initially the whole word), and if the first character is not already in the dictionary (which it won’t be, at the start), then it creates the character within the dictionary. If the character is already in the dictionary, then another branch needs to be added to the existing node. Each character of the slice is processed like this which would produce one branch. Then the next slice is also considered in the same way, until there is no more to take a slice from.

Subsequently, another function will use this tree (dictionary) to find the pattern. It works through the pattern, and tests to find if each successive character can be found in the dictionary. If yes, then it will check the next dictionary (branch) down to see if the next character follows from the first. If at any point the character cannot be found, then this function returns “False”, but if it manages to find the complete pattern in the tree, then “True” is returned. This means that a simple yes or no can be printed if there is or isn’t a match.

This is not particularly fast at all, and can require a lot of memory for long pieces of text (as each possible suffix must be stored). Using a large amount of memory can mean that it takes longer if the computer must start writing things to the hard drive. However, this implementation of suffix trees could also take a long time because building the tree requires $O(n^2)$ time. Though the actual searching is very quick because the pattern must only be looked at once, nonetheless, creating the suffix trees is the most limiting factor when considering this implementation (it is due to this problem that newer implementations were created).

4.3.2 Suffix Trees – Searching without a Pattern

An advantage of suffix trees is that they are able to search for patterns, rather than the occurrence of a particular one. The way this works is that the programme takes note of anything that it has seen before, when it is constructing the tree.

The previous algorithm can be used in the same way, so again slices of the text are taken (starting from the longest, whole text, and subsequently removing the first character), and are passed to a function which assess whether a new branch is required or not. If the character is already present, then the algorithm counts this (by incrementing a stored variable by one), then if the next character is present, then this is counted again (another increase of one), and this continues through the tree-building process. If this variable is higher than any previous counts of length, then it is stored along with the starting position of the first match, the starting position of another match, and the length.

There is another way of doing it, where the pattern can be found once the tree has already been constructed. This would be done by making a programme, which counts the number of endings (the leaf nodes). This method would be useful if the same text string is often being used, and the suffix tree can be built in advance, and accessed for the searching. However, there is also an advantage because this method would enable the specifying of how many endings are to be searched (and so how many matches). Nonetheless, it may be faster and easier to implement the method that finds the longest while constructing the tree.

In order to test this with chromosome 21, only very (comparatively) small sections could be used, to prevent the freezing of the computer. Nevertheless, the programme could be run by entering a file containing the short section of chromosome (not the whole chromosome, and taking a slice of it, but slicing it before-hand, in order to reduce the running time and ensure more reliable comparison) as a parameter of the function.

5. Results and Discussion

5.1 Exact Matching in Chromosome 21

| String | Repeat | Time (s) | | | | | Total Count |
|---------------------------------------|--------|----------|------------|-----------|-----------|-----------|-------------|
| | | KMP | Rabin-Karp | Naïve (1) | Naïve (2) | Naïve (3) | |
| ZN-Finger GAT (length 3) | 1 | 104.4 | 154.0 | 59.6 | 127.4 | 138.4 | 449868 |
| | 2 | 103.6 | 151.5 | 59.1 | 127.5 | 138.8 | |
| | 3 | 103.0 | 151.2 | 59.6 | 127.5 | 138.3 | |
| | Mean | 103.7 | 152.2 | 59.4 | 127.5 | 138.5 | |
| TATA box TATAAA (6) | 1 | 105.4 | 155.9 | 59.4 | 128.7 | 172.0 | 26122 |
| | 2 | 105.7 | 152.3 | 60.4 | 128.8 | 171.6 | |
| | 3 | 105.3 | 151.2 | 60.0 | 126.8 | 170.7 | |
| | Mean | 105.5 | 153.1 | 59.9 | 128.1 | 171.4 | |
| MADS box CTATTTATAG (10) | 1 | 105.1 | 155.7 | 59.8 | 127.8 | 215.0 | 64 |
| | 2 | 104.8 | 151.2 | 59.8 | 128.1 | 213.2 | |
| | 3 | 104.3 | 151.4 | 59.7 | 128.7 | 213.4 | |
| | Mean | 104.7 | 152.8 | 59.8 | 128.2 | 213.9 | |
| Nuclear Receptor GGTCAAAGGTCA (12) | 1 | 103.0 | 154.0 | 59.6 | 134.3 | 238.2 | 2 |
| | 2 | 103.1 | 151.1 | 59.6 | 133.9 | 235.7 | |
| | 3 | 105.3 | 150.9 | 59.5 | 133.5 | 237.2 | |
| | Mean | 103.8 | 152.0 | 59.6 | 133.9 | 237.0 | |

These results are perhaps not what would normally be expected. The purpose of algorithms such as Knuth-Morris-Pratt and Rabin-Karp is to shorten running times and thus be more efficient than the naïve method of searching. This is not at all conclusive from the data above. One of the naïve searches (Naïve 1) is almost twice as fast as Rabin-Karp, but if this were what is normally the case, there would be no point in developing alternative methods for string searching, or for calling the original method “naïve”. Naturally, the time required for the tests may vary depending on the CPU usage, and therefore what other things the computer is doing at any one time; however this should not vary the time too greatly, and would be unlikely to cause one algorithm to take significantly longer than another. Further, all of these tests were done at different times, so no algorithm was done in one go, but rather a mixture of different algorithms and pattern lengths followed one another. Similarly, the repeats should also mean that any irregularity caused by other activities in the background (which include those undertaken by the system, and which the user cannot control), would be either averaged or identifiable (if significant). In this respect, there do not appear to be any outliers or anomalies, which are significantly different to other values, except for – perhaps – the timing for the Nuclear Receptor, using the second version of the naïve test. The three tests were a few seconds higher than with the other strings, which is unusual and unexpected, but could be due to the fact that these were carried out on a different day than the some of the other tests, although there were other tests that were also carried out on this day, and there does not appear to be any difference. Nonetheless, there may have been some operations happening at the same time as when these were done, as it was a different day.

Another influence that could be the reason for the much longer time for Rabin-Karp could be memory usage. If the computer runs out of memory (RAM), then it must start writing to the hard drive in order to store information from the programme. This takes time, and therefore can slow a programme down. Taking note of the maximum memory usage during the operations can provide some information.

| | Max Memory Usage (MB) | | | | |
|---------|-----------------------|------------|-----------|-----------|-----------|
| | KMP | Rabin-Karp | Naïve (1) | Naïve (2) | Naïve (3) |
| Virtual | 810 | 994 | 810 | 810 | 810 |
| Real | 787 | 966 | 787 | 787 | 787 |

This table of memory usage shows how the Rabin-Karp programme requires more memory than the other programmes. The real memory usage is what could ultimately limit how fast the programme runs; if the real memory usage is too great, then the hard drive is written to.

Another factor that occurs when looking at the main table is that most of the values appear to be generally constant throughout the various lengths of pattern. This could also be unexpected, as the running times should be proportional to the length of the text multiplied by (or added to) the length of the pattern. Perhaps for KMP this would not apply, because if it is only addition of the two lengths, then perhaps the lengths of the patterns are not long enough to produce a noticeable increase in time (noticeable above the 'noise' possibly created by system operations). Nonetheless, the naïve methods and Rabin-Karp should probably show an increase in time, proportional to the increase in pattern length.

Perhaps the causation of the values being roughly constant could be because of some problems with the chosen strings. This is because, as the string length increases, the number of these motifs found decreases by much more. If the number of matches decreases (and it is assumed that the number of false matches also decreases), then possibly the time required will decrease too, because the algorithm – for example the naïve method – does not need to check any position more than once (because it does not match). This would mean that the running time could be expected to increase at the same time as decrease, and thus perhaps become more stable. Nevertheless, this may be not be a completely valid explanation, as the number of false matches may increase with increase in pattern length, because it may begin matching at the start of the pattern, and not find a mismatch until much later than it would with a short motif.

```
def naive1(T, P, s):
    return T[s:s+len(P)] == P

def naive2(T, P, s):
    for i in range(len(P)):
        if P[0+i] == T[s+i]:
            pass
        else:
            return False
    return True

def naive3(T, P, s):
    match = True
    for i in range(len(P)):
        match = match and (P[i] == T[s + i])
    return match
```

The Methods of Naïve Searching - The first method will return "True" if the slice of the text matches the pattern. The second will return "False" as soon as there is not a match (and stop the function), but otherwise return "True" (if it can go through the entire function). The third method assumes "True" (that there can be a match), and then will form either "True" or "False", on the basis of the next character matching or not. Even if there is not a match, it will maintain "False" and continue on throughout the pattern anyway, and return either "True" or "False" at the end (and only at then end).

Nonetheless, the time for the third naïve method obviously increases, with an increase in length of pattern, which is a very different trend than in the other algorithms. This could have something do to with the difference in the way that the algorithm works. The first naïve method is fast because the programme is doing the searching itself, and it is thought that perhaps the slicing is a fast internal process (though this still does not necessarily explain why it should be faster than Knuth-Morris-Pratt or Rabin-Karp). The second is probably slower because it is a manual process: the algorithm is instructing the computer to check each position for a match, but as

soon as a match is not found, then the function returns “False” (see diagram for algorithm). However, the third will never return (end) until it has completely checked through the whole pattern. Therefore, the time required for this algorithm could be expected to take longer - more than the other algorithms - with an increase in pattern length.

5.1.1 Further Analysis

In order to test these further, another experiment was carried out which could mean that some of the effects of different variables could be examined more closely. This meant running the same programmes, but on some different patterns.

| String | Repeat | Time (s) | | | | | Total Count |
|---|--------|----------|------------|-----------|-----------|-----------|-------------|
| | | KMP | Rabin-Karp | Naïve (1) | Naïve (2) | Naïve (3) | |
| Random 40 ATGACGTCCAGGACTT AGCAGGGACAGATCC AGCTGATGA | 1 | 106.8 | 156.9 | 61.1 | 152.6 | 552.8 | 0 |
| | 2 | 106.8 | 157.4 | 61.1 | 151.4 | 554.8 | |
| | 3 | 106.4 | 156.1 | 60.8 | 151.9 | 554.3 | |
| | Mean | 106.7 | 156.8 | 61.0 | 152.0 | 554.0 | |
| Twenty 'N's | 1 | 101.4 | 182.3 | 57.5 | 216.2 | 344.5 | 12774044 |
| | 2 | 101.6 | 182.1 | 57.5 | 215.3 | 347.2 | |
| | 3 | 101.9 | 182.4 | 57.5 | 225.0 | 347.0 | |
| | Mean | 101.6 | 182.3 | 57.5 | 218.8 | 346.2 | |
| Random 6 TGCATG | 1 | 107.7 | 152.1 | 59.1 | 130.3 | 176.2 | 10952 |
| | 2 | 105.6 | 154.2 | 59.2 | 131.3 | 176.1 | |
| | 3 | 105.6 | 152.7 | 59.3 | 132.3 | 177.1 | |
| | Mean | 106.3 | 153.0 | 59.2 | 131.3 | 176.5 | |

The algorithms were tested with these patterns because they each can show some significance. Testing for the random string of 40 was a way of seeing the effect of having a long pattern (where KMP and Rabin-Karp should take time proportional to the pattern length and the text length, while the naïve methods should multiply those values, and so be much slower for longer patterns). From these results it is possible to see that, with long patterns, there is indeed a considerable increase in the amount of time used for the naïve methods, while the KMP and Rabin-Karp algorithms remain quite similar to their previous times.

A random string of 40 would be able to test because of its length, but there are no matches. Therefore, using a pattern of twenty “N”s long would mean that many matches (from the long blocks of “N” in the chromosome) would occur, and so algorithms such as Rabin-Karp would take longer (having to test for matches, in a similar way to the naïve method). Further, there could be a more obvious difference in timing with the difference in length.

It is also possible, from these results to look at the statistical significance of them. The probability of finding a pattern in a random string (or, a random pattern in a string) can be calculated.

$$n \left(\frac{1}{4} \right)^m$$

Where n is the length of the text, m is the length of the pattern, and the four comes from the fact that there are four letters in the dictionary (A, C, G, T).

Using this equation for the probability of finding a pattern in a random string, it is possible to find estimates of the total count of a pattern.

| Length | 3 | 6 | 10 | 12 | 20 | 40 |
|-------------------|--------|------|----|----|----------|----------|
| Theoretical Count | 533908 | 8342 | 33 | 2 | 3.11E-05 | 2.83E-17 |

In some cases (for example the TATA box, the MADS box, and the twenty “N”s), there are significantly more of the actual motifs found, than would be expected by predicting in a random string. This strongly suggests that these motifs are not found by chance, but do actually have

importance within the context of a chromosome. This means that the number of occurrences of random strings should be much closer to the theoretical, for example the count of the random string of 6 is considerably less than that of TATAAA, and is much closer to the theoretical value.

Nonetheless, there are occasions where there are more expected by the theoretical count than actually occurred. For example, this happens for the pattern of length three; this could be due to the structure of the chromosome, and the little requirement for such patterns of three.

5.2 Local Alignment and Position Weight Matrices

These are really two completely different algorithms, which are designed for different operations. Alignment infers the evolutionary (phylogenetic) tree of the species: it can help compare animals, and find areas of similar function in different DNA sequences (for example where proteins bond) - it is not really a good method for searching for motifs (where gaps are unlikely), but for aligning the sequences and finding areas of similar function. However, position weight matrices are better for searching for motifs (while allowing some common 'errors' (gaps and mutations)).

| | Position Weight Matrices | Local Alignment |
|-------------|--------------------------|-----------------|
| Total Count | 3444 | 490 |
| Time (s) | 440 | 1590 |
| Matches | CTATTTATAG | CTATTTATAG |
| | CTATTTTATAG | CTCATTATAG |
| | CTATATATAG | CTTATTTATAG |
| | TTATTTATAG | CTAATTTATAG |
| | CTATTTATAA | CTATTTATAGA |
| | CTATTAATAG | CTATTGTATAG |
| | CTATATTTAG | CTATTTATTAG |
| | TTATTTTATAG | CTATTCTATAG |
| | CTATTTTAA | CTATTATATAG |
| | CTATTATTAG | CTATATTATAG |
| | CTATTTATAT | CTATTTCATAG |
| | CTAATTATAG | CTACTTTATAG |
| | CTATTTAAAG | CTATTTATGAG |
| | CTATTTATGG | CTATTTAGTAG |
| | TTATATATAG | CTATTTATAGG |
| | CTATTTTAT | CTATTTATAAG |
| | CTATATATAA | CTATTTATAGC |
| | CTATAAATAG | CTAGTTTATAG |
| | CTAATTTTATAG | CTATTTATATG |
| | CTATTTTGG | CTATTTATAGT |
| | CTATTTTAAG | CTATTTATCAG |
| | ATATTTATAG | CTATTTTATAG |
| | CTACTTATAG | CTGATTTATAG |
| | CTCTTTATAG | CTATCTTATAG |
| | TTATTTATAA | CGTATTTATAG |
| | TTATTAATAG | CTATTTAATAG |
| | TTATATTTAG | CTATTTACTAG |
| | CTATTAATAA | CATATTTATAG |
| | CTATATTTAA | CTATTTGATAG |
| | CTATAATTAG | CTATGTTATAG |

Nonetheless, it can be possible to compare the results that these two algorithms give, and look at the ways that they do similar actions. Both of these algorithms can be used to find variations of a pattern in a text, however position weight matrices do this in a more empirical way (where scoring the match comes from how often this type of match has been observed), and local alignment may be more arbitrary and uses scoring of individual adjustments to the pattern.

For local alignment, these results do not show gaps, but nonetheless the two sets of results appear to differ greatly from each other: all of the results for local alignment begin with a "C", whereas the search using position weight matrices does not show this same constancy. The results from local alignment tend to be a character longer than that found with position weight matrices. For example, some of the matches are the same pattern (match), but with an extra character on the end. This could suggest that the score for a gap is too small, and should be increased if this algorithm were to be repeated.

From these results, it can also be noticed that there were significantly more overall matches found by the PWM algorithm, than there were by local alignment. This is perhaps not expected, as the top 30 sequence

matches were taken in both cases, but local alignment found fewer occurrences of the sequences.

Furthermore, the local alignment method took a very long time, in comparison to the PWM method. This could be because of the fact that - in order to do the local alignment - all of the

“N”s had to be removed first. On the contrary, this observation may appear unusual, because the PWM method found many more matches, which would have to be stored, and thus require more memory usage, though perhaps there is enough memory available for these processes to not take much longer.

These two sets of data may not be completely comparable, because of the fundamental difference in their operation and purpose, but they should be able to act as a general indication towards their outcomes in comparison to each other. However, the fact that the scores had to be decided and guessed for alignment could mean that the results may not mean very much.

5.3 Suffix Trees

The exact matching method of implementing suffix trees was not tested on the chromosome because of the large amount of space that it would require, and it would only produce a “yes” anyway, rather than some of the detail that other methods would provide. Instead, finding the longest string that occurs at least twice would be far more interesting, and make more use of the benefits of suffix trees.

| Length of Text String | Time (s) | Max Real Memory Usage (MB) | Max Virtual Memory Usage (MB) | Longest Sequence |
|-----------------------|----------|----------------------------|-------------------------------|----------------------|
| 1000 | 2.6 | - | - | TTTTAGAGCAGGAATGAAAG |
| 1000 | 2.6 | - | - | GTTAGCTCAGC |
| 2000 | 24.0 | 360 | 380 | TTTTAGAGCAGGAATGAAAG |
| 3000 | 101.7 | 800 | 830 | TTTTAGAGCAGGAATGAAAG |
| 4000 | 310.6 | 1410 | 1430 | TTTTAGAGCAGGAATGAAAG |
| 4000 | 340.1 | 1420 | 1450 | TTTTTTTTTTTTTTTT |
| 5000 | >17mins | 1700 | 1810 | - |

It is interesting to see that the longest pattern (that occurs more than once) is of length 20. However, this sequence does not occur in different sections of the chromosome. It is possible to estimate a theoretical maximum length of a repeated string in the whole of the chromosome. The whole length of it is 46,944,323, but removing all of the “N”s (and therefore giving a length of text that can actually contain patterns) means that the total length is 34170106. An (simplified) equation can provide the approximate value, and - using this - it is possible to see that the longest pattern that could be expected from the whole chromosome would be about 24 or 25 characters long. It could be quite surprising, however that a section of 1000

$$\sqrt{\frac{\pi a}{2}} = N$$

$$a = \frac{2N^2}{\pi}$$

$$= \frac{2(34170106)^2}{\pi}$$

$$= 7.4 \times 10^{14}$$

$$a = 4^b$$

$$b = \log_4 a$$

$$= \log_4(7.4 \times 10^{14})$$

$$= 24.7$$

characters can contain a repeated string of length 20, because - using the same equation - the expected length would be more like 10 characters long. This suggests that this area is of significance (possibly has a function), and could be one of the main areas that would contain the longer strings.

Estimate of the length of longest pattern

where:

b is the maximum length of pattern

a = 4^b (due to the size of the dictionary: A, C, G, T)

N is the length of the text (the whole chromosome)

These results show the amount of memory required for the tests, and how this – as well as the time required – can grow very quickly. This means that the naïve method for suffix trees cannot be used on the whole chromosome. Even searching through only 5000 characters caused the computer to run out of memory, and a warning occurred, instructing the closure of programmes in order to prevent further problems (for this reason, it was not possible to get a time of the procedure, or a result – the result of the memory usage was the maximum until that point). At this point, the computer was completely frozen, and took a while to respond. Similarly, the computer seemed to be writing to the hard drive, as there was only about 20 MB of memory free (where the RAM on the computer was 2 GB, and the (real) memory usage during the test with 5000 was – at the maximum – 1.67 GB, and there were other operations happening in the background). The increase in time is beyond quadratic, which would be expected because the algorithm processes each of the characters in the text, then each of the following ones again, and so on. This could be tested by creating an empty list and iterating through the first 1000 numbers, then 2000, then 4000 and 8000. At each iteration, the

| Number | 1000 | 2000 | 4000 | 8000 |
|----------|------|------|------|------|
| Time (s) | 0.6 | 2.4 | 9.3 | 37.4 |

Testing for a quadratic increase in time (without storing information) - When the number that is iterated through is doubled, the time required (to append it to a list and delete it again) is multiplied by about four. This is a quadratic increase in time.

subsequence numbers were also iterated through, and appended to the list, then deleted from the list (so that nothing was stored), and so the process is very similar to the suffix tree programme. Testing the time required for this produced an almost quadratic increase (see diagram). Further, it also seems that the memory usage increases quadratically too, because when

doubling the length of the text (from 2000 to 4000), the real memory usage is multiplied by almost four (it was difficult to get a reading of memory usage for the string of length 1000, as it was too fast to be sure of the values).

Python is very inefficient when it comes to using suffix trees. Other languages, such as C, can provide much faster algorithm, because of the way that the language operates. Further, if a server (with much more memory, and a faster processor) is used, with this language, and a more modern algorithm (such as Ukkonen's), it can be possible to run the whole chromosome.

6. Conclusion

The aim of this project was to be able to search for patterns in chromosome 21, and explore the different methods of doing this, and hence learn more about how this process is carried out by the scientists (biologists, biochemists, mathematicians/statisticians and even their computer-related counterparts) in reality, as well as some of the problems they face.

From testing the different methods of exact matching, it was difficult to see that there could be some significant advantages of methods such as Knuth-Morris-Pratt and Rabin-Karp. However, the three methods were also compared on server, with a 3.0 GHz processor, 33 GB RAM (memory), and 8 core processors (compared to the computer that was normally used, which had a 1.83 GHz, dual core, processor and 2 GB RAM). With this (faster, and no writing to the hard drive because the memory was large enough to hold all of the information), it was possible to finish the naïve method in 23.3s, the fast Rabin-Karp in 32.3s (the slow version took 59.5s), and 21.9s for KMP. This meant that it could be possible for KMP to be slightly faster than the naïve method, however Rabin-Karp was still much longer, which is definitely not what should happen. Therefore, this experiment has either concluded the opposite to general consensus, or is inconclusive, and thus this information probably could not be used in situations where exact searching is required, without further investigation first. Nonetheless, exact matching would be useful to biologists when specific areas are to be researched, and therefore fast algorithms could be vital for saving time and producing results quickly.

The experiment comparing local alignment and position weight matrices could be very different to what would be expected, because the results were so different to each other. However, it could be thought that conclusions could be made regarding how long the two methods take, where local alignment appears to take considerably longer than the position weight matrix form of searching. It is quite peculiar, that the sequences found by the algorithms – where both are searching for the best matches – are so different than each other. From the problem of assigning scores in local alignment, it could also be thought that position weight matrices could be the better way of giving a score to a match, especially as it is less arbitrary (and more empirical), but also because it can give more precise scores (which could be represented as a percentage of the maximum score). This could be a sensible conclusion, as it is the purpose of PWMs to score matches like this, where as local alignment is an attempt to look at a larger picture, to more chromosomes.

From the suffix trees, it could be quite difficult to draw any particular conclusions from the investigation, as it was impossible to test very much of the chromosome. Nonetheless, it could be concluded that suffix tree algorithms would naturally be a good (fast) method of searching for patterns in a previously established suffix tree (such as where the same text – or chromosome – is being used repeatedly), but is greatly hindered by the construction of the suffix tree and the storage of it. However, suffix trees are still able to find the longest sequence that is repeated, which is not possible with the other algorithms used here. This would be useful in cases where similarities in the same chromosome are to be investigated (for example when looking for areas of similar functionality within one chromosome, such as repeated binding sites).

7. Evaluation

On the whole, many of the investigations appeared to go well, and produced interesting results. Generally, if there was something that could be done in an improved way, it was done anyway. However, there was the obvious issue of having a naïve exact searching method, which was faster than the later methods developed in order to make string searching more efficient. Nonetheless, it was difficult to see how there could be any more ways of finding the reason for this, though this could be something to look into further if there was more time; for example, other people's implementations of the methods could be taken and tested, or perhaps alternative programming languages could be experimented with, to see the effects of these with the algorithms (Python may have an efficient slicing function, etc). Other option could be to implement the Boyer-Moore algorithm, which could be another algorithm to compare, and be something that should be even faster than KMP. Therefore, a test could be made into whether this result is seen, and if not it would suggest something wrong with the methodology, which produced the unusual results.

Other problems that occurred when trying to take values and data, included finding the most efficient algorithm. Generally, this was something to try out and to work through, creating and imagining methods that could work best. Further, it was often difficult to visualise the process that the programme was using, and some algorithms required more imagination than others (such as KMP, dynamic programming, and suffix trees).

Another pursuit that could be linked to this project could be to do more with suffix trees, as these were not really explored much, despite their potential. It could be beneficial to be able to run a better suffix tree algorithm such as Ukkonen's (and perhaps create one too), and perhaps use the server more for this, in order to achieve results of the longest string that occurs more than once. From searching the whole chromosome, it could be interesting to compare the results against one of the more traditionally used methods for searching for motifs in DNA (usually websites or systems such as "motif scanner"). This could be particularly interesting if the methods used within the project could be used to build a database of promoters (sections of DNA that help with the making of (the transcription of) a particular gene), and finding new ones that have not already been found.

Further, more time could mean that the project could be enhanced by adjusting the current suffix tree algorithm in a way that could find the longest sequence that occurs a specified number of times, or perhaps counting the number of occurrences of this particular sequence. Another improvement to this implementation could be to also count the different sequences of the same length (for example, in "ababa" both "ab" and "ba" are of length two and occur twice, but the current algorithm would only give one of them as the result).

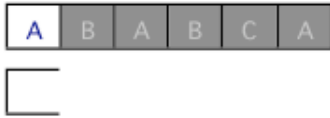
Slightly further beyond the scope of this project, more work could be done using alignment, and following that path. The algorithms could be used in conjunction with perhaps another species' DNA, in order to search for any areas of similarity.

It could also be preferable to also have done more research into the biological aspects of the information found from this project, such as learning more about the structure of DNA in the sense of the functionality of different regions, and the significance of the promoters that were found. Similarly, it could be enlightening to learn more about the importance of the work that was carried out, and perhaps learn more about the research that goes into DNA (for example how the nucleotide sequence is established, how promoters are identified, etc).

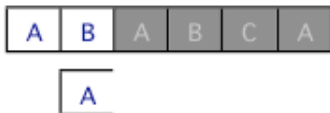
8. Appendix

Knuth-Morris Pratt Diagram

What the "pi" function does



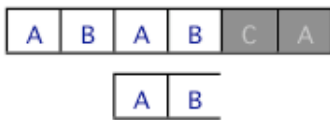
First iteration: the sequence (top) contains the sequence up to the current suffix. The one below represents the current prefix, which is - at first - empty (no prefix that is not a suffix). There is no match in position 0 (the first position of the prefix), so $\pi = [0]$, and the lower sequence shifts along.



This time, there is still no match of position 0 of the prefix with the main sequence. Therefore, now, $\pi = [0, 0]$ and the prefix shifts along again.



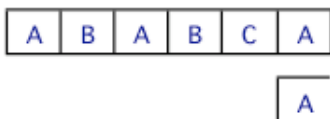
There is now a match in the sequence with the first position of the sequence, so the longest prefix that is also a suffix, is length one. Therefore, $\pi = [0, 0, 1]$.



Comparing the next positions also gives a match, so the longest prefix that is also a suffix is "AB". Now $\pi = [0, 0, 1, 2]$.



Now, there is no match. There is no prefix that is also a suffix. Therefore, $\pi = [0, 0, 1, 2, 0]$.



In the last position, the longest prefix that is also a suffix, is "A". Hence, $\pi = [0, 0, 1, 2, 0, 1]$.

8.1 Programmes

Naïve Searching

```
#!/usr/bin/env python

import sys

match_str = None

def match_at(T, P, s):
    global match_str
    match_str = "Occurrence with match_at"
    return T[s:s+len(P)] == P

def match_at2(T, P, s):
    global match_str
    match_str = "Occurrence with match_at2"
    for i in range(len(P)):
        if P[0+i] == T[s+i]:
            pass
        else:
            return False
    return True

def match_at3(T, P, s):
    global match_str
    match_str = "Occurrence with match_at3"
    match = True
    for i in range(len(P)):
        match = match and (P[i] == T[s + i])
    return match

f= open(sys.argv[1], "r")
T = f.readline()
f.close()
P = sys.argv[2]

def naive_str_match(T,P,match=match_at3):
    n = len(T)
    m = len(P)
    count = 0
    for s in range(n-m+1):
        if match(T, P, s):
            count = count + 1
    print "Number of matches:", count
naive_str_match(T, P)
```

Rabin-Karp

```
#!/usr/bin/env python

import sys
from math import pow

def match(string, pattern, x):
    return pattern == string[x:x+len(pattern)]

def dictradox(string, pattern):
    dict = {}
    for i in string+pattern:
        if i not in dict:
            dict[i] = len(dict)
    string = [dict[i] for i in string]
    pattern = [dict[i] for i in pattern]
    radix = len(dict)
    return radix, string, pattern

def Rabin_Karp(string, pattern, modular, match, dictradox ):
    radix, string, pattern = conversion(string, pattern)
    n = len(string)
    m = len(pattern)
    p=0
    for i in pattern:
        p = (radix * p + i) % modular
    s=0
    for i in string[:m]:
        s = (radix * s + i) % modular
    matches = []
    count = 0
    if s == p and match(string, pattern, 0):
        count = 1
    for i in range(1, n - m + 1):
        s = (radix * (s - pow(radix,m-1) * string[i-1]) +
            string[i+m-1]) % modular
        if s == p and match(string, pattern, i):
            count = count + 1
    print "count is", count

f=open(sys.argv[1],'r')
str=f.readline()
f.close()

Rabin_Karp(str, sys.argv[2], 16647133)
```

Knuth-Morris-Pratt

```
#!/usr/bin/env python

import sys

f= open(sys.argv[1], "r")
T = f.readline()
f.close()
P = sys.argv[2]

def find_pi(P):
    m = len(P)
    pi = [0]
    k = 0
    for q in range(1, m):
        while k > 0 and P[k] != P[q]:
            k = pi[k-1]
        if P[k] == P[q]:
            k = k + 1
        pi.append(k)
    return pi

n = len(T)
m = len(P)
pi = find_pi(P)
q = 0
count = 0
print pi
for i in range (0, n):
    while q > 0 and P[q] != T[i]:
        q = pi[q-1]
    if P[q] == T[i]:
        q = q+1
    if q == m:
        print "Pattern occurs with shift", i-(m-1)
        count = count + 1
        q = pi[q-1]

print count
```

Local Alignment (to be compared against PWM)

```
#!/usr/bin/env python
```

```
import sys
```

```
f= open(sys.argv[1], "r")
```

```
T = f.readline()
```

```
f.close()
```

```
P = sys.argv[2]
```

```
T = T.replace("N", "")
```

```
def process(T, P, gap = -2):
```

```
    matrix = [[2, -3, -2, -3], [-3, 2, -3, -2], [-2, -3, 2, -3], [-3, -2, -3, 2]]
```

```
    dict = {'A':0, 'C':1, 'G':2, 'T':3}
```

```
    matches = []
```

```
    def matrix_score(T, P):
```

```
        a = dict[T]
```

```
        b = dict[P]
```

```
        return matrix[a][b]
```

```
    previous_score = [""]*(len(P)+1)
```

```
    current_score = [""]*(len(P)+1)
```

```
    for j in range(len(T)+1):
```

```
        current_score[0] = (j, 0)
```

```
        for i in range(1, len(P)+1):
```

```
            if j == 0:
```

```
                current_score[i] = (0, i*gap)
```

```
            else:
```

```
                optimum = previous_score[i][1]+gap
```

```
                value = current_score[i-1][1]+gap
```

```
                if optimum > value:
```

```
                    source = 0
```

```
                else:
```

```
                    optimum = value
```

```
                    source = 2
```

```
                value = previous_score[i-1][1]+ matrix_score(T[j-1], P[i-1])
```

```
                if value>optimum:
```

```
                    optimum = value
```

```
                    source = 1
```

```
                if source == 0:
```

```
                    current_score[i] = (previous_score[i][0], optimum)
```

```
                elif source == 2:
```

```
                    current_score[i] = (current_score[i-1][0], optimum)
```

```
                else:
```

```
                    current_score[i] = (previous_score[i-1][0], optimum)
```

```
                if i==len(P) and optimum>=(len(P))*matrix[0][0]-3:
```

```
                    matches.append((current_score[i][0], j, current_score[i][1]))
```

```
            (current_score, previous_score) = (previous_score, current_score)
```

```
    return matches
```

```
result = process(T, P)
```

```
dict = {}
```

```
for y in result:
```

```
    string = T[y[0]:y[1]]
```

```
    if string not in dict:
```

```
        dict[string] = (1, y[2])
```

```
    else:
```

```
        dict[string] = (dict[string][0] +1, dict[string][1])
```

```
for x in range(len(dict)):
```

```
    print "Sequence:", dict.keys()[x], "Count:", dict.values()[x][0], "Score:", dict.values()[x][1]
```

```
print "Number of sequences:", len(dict)
```

```
print "Total number of matches:", len(result)
```

PWM (to be compared against Local Alignment)

```
#!/usr/bin/env python

from math import *
import re
import sys
dict = {'A': 0, 'C': 1, 'G':2, 'T':3}

def PWM(T, limit, Pmatrix):
    logmatrix = []
    smallnumber = 0.000001
    minimum = 0
    maximum = 0
    for i in range(len(Pmatrix)):
        logmatrix.append([""]*len(dict))
        total_rate = Pmatrix[i][0]+Pmatrix[i][1]+Pmatrix[i][2]+Pmatrix[i][3]
        for j in range(4):
            logmatrix[i][j] = log(float(Pmatrix[i][j])/total_rate+smallnumber)
        minimum += min(logmatrix[i])
        maximum += max(logmatrix[i])
    def signum(x):
        if x>0:
            return 1
        if x<0: return -1
        return 0
    result = []
    for s in range(len(T)-len(Pmatrix)+1):
        Tprobability = []
        for x in range(len(Pmatrix)):
            a = dict[T[s+x]]
            Tprobability.append(logmatrix[x][a])
        score = (sum(Tprobability)- minimum)/(maximum-minimum)
        if score>limit:
            result.append((s, T[s:s+len(Pmatrix)], score))
    result.sort(lambda x, y: signum(y[2] - x[2]))
    return result

f= open(sys.argv[1], "r")
T = f.readline()
f.close()
T = T.replace("N", "")
g = open(sys.argv[2], 'r')
h = g.readlines()
PmatrixRead = []
for y in range(len(h)):
    list = [int(j) for j in re.split(" +", h[y].strip(" "))]
    if len(PmatrixRead) == 0:
        for x in range(len(list)):
            PmatrixRead.append([""]*len(dict))
    for x in range(len(list)):
        PmatrixRead[x][y] = list[x]
answer = PWM(T, 0.967, PmatrixRead)

dict = {}
for y in answer:
    if y[1] not in dict: dict[y[1]] = (1, y[2])
    else: dict[y[1]] = (dict[y[1]][0] +1, dict[y[1]][1])
for x in range(len(dict)):
    print "Sequence:", dict.keys()[x], "Count:", dict.values()[x][0], "Score:", dict.values()[x][1]
print "Number of sequences:", len(dict)
print "Total number of matches:", len(answer)
```

Suffix Trees (Longest Pattern)

```
#!/usr/bin/env python

import sys
f= open(sys.argv[1], "r")
T = f.readline()
f.close()
T = T.replace("\n", "")

dict = {}
maxl = (0, 0, 0)

def suffixtree(S, j):
    point = dict
    length = 0
    global maxl
    position = j

    for i in range(len(S)):
        if S[i] in point:
            length += 1
            if length>maxl[2]:
                maxl = (point[S[i]][0], position, length)
            point = point[S[i]][1]
        else:
            point[S[i]] = (j, {})
            point = point[S[i]][1]

    return maxl

for j in range(len(T)-1):
    suffixtree(T[j:len(T)], j)

print maxl, T[maxl[0]:maxl[0]+maxl[2]]
```

9. References

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest (1994); *Introduction to Algorithms (first edition)*; The MIT Press.

This book provided a diagram of the Rabin-Karp process.

<http://genome.ucsc.edu/>

This website provided the DNA that we used.

<http://jaspar.genereg.net/>

This website provided some diagrams, and the motifs that were searched for in the DNA.

Understanding Science; http://undsci.berkeley.edu/article/0_0_0/dna_02

This website provided a diagram of the shape and structure of the different nucleotides.

http://mila.cs.technion.ac.il/~yona/suffix_tree/

This website gave Ukkonen's implementation of suffix trees in C.

10. Bibliography

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest (1994); *Introduction to Algorithms (first edition)*; The MIT Press.

Michael Dawson; *Python Programming*; Premier Press (2003)

<http://docs.python.org/tutorial/index.html>

<http://marknelson.us/1996/08/01/suffix-trees/>

http://en.wikipedia.org/wiki/Ordinal_number

11. Acknowledgements

Appreciation must go towards the Oxford Centre for Genome Function, within which this project was undertaken. Moreover, thanks is also particularly due to Professor Jotun Hein, Mr Adam Novak, and Dr Rune Lyngsø who co-wrote the project, and generally helped a lot. Their seemingly endless patience and helpful hints, suggestions, and direction contributed considerably to the completion of this project. Furthermore, Michelle Parker (who also participated in a project) was responsible for implementing the original Rabin-Karp algorithm that was used, which originally came from pseudo-code in "Introduction to Algorithms" (along with the KMP code too). Otherwise, programmes were generally created by planning, trial-and-improvement, and thinking about the goal, with help, suggestions and feedback from supervisors.

The project was useful, and certainly provided much information about how scientists operate. It was interesting to be able to see the type of environment in which research occurs on a daily basis, and to learn more about what is involved in research.