Motif Discovery

Michael Charleston
michael.charleston@sydney.edu.au

CS4HS
April, 2012

This material is adapted from a 3rd year University course on *bioinformatics*, the discipline of solving biologically motivated problems that require clever algorithms to solve.

One of the main goals in the course is to get students to come up with new algorithms to solve such problems. The later exercises in this part of the workshop are to give you a taste of this.

What is DNA made of?

- DNA is *deoxyribonucleic acid*.
- It is arranged in a double helix of base-pairs.
- Each strand of the double helix is a long chain of *nucleotides* joined by peptides.

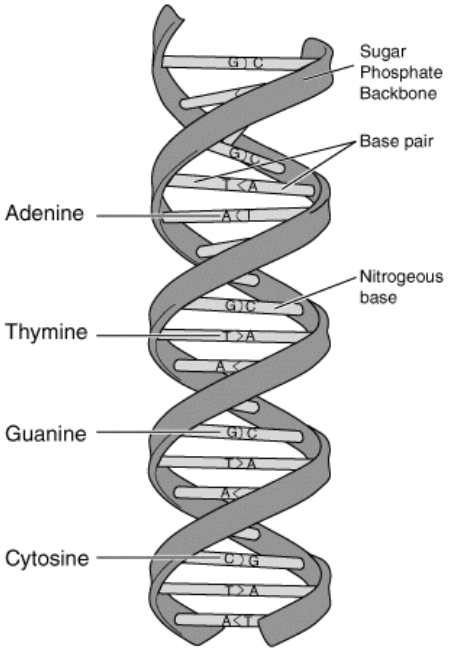
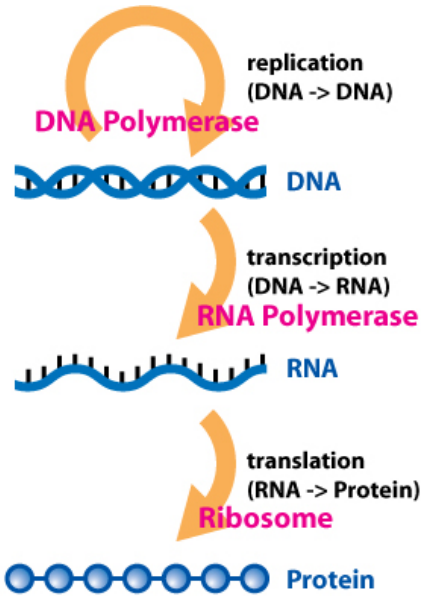


image courtesy Wikimedia commons

The Central Dogma

The "central dogma of molecular biology" is that:



- Genes in DNA are
- *transcribed*[1] into RNA,
- which is then *translated*[2] to proteins

Background

A certain experiment shows that if some gene X is "knocked out" then another 20 genes are no longer expressed.

How can this happen?

Transcription Factors

- Gene X encodes a *regulatory protein*, a.k.a. a *transcription factor* (TF)
- The 20 unexpressed genes rely on gene X's TF to induce transcription
- A single TF may regulate *multiple* genes

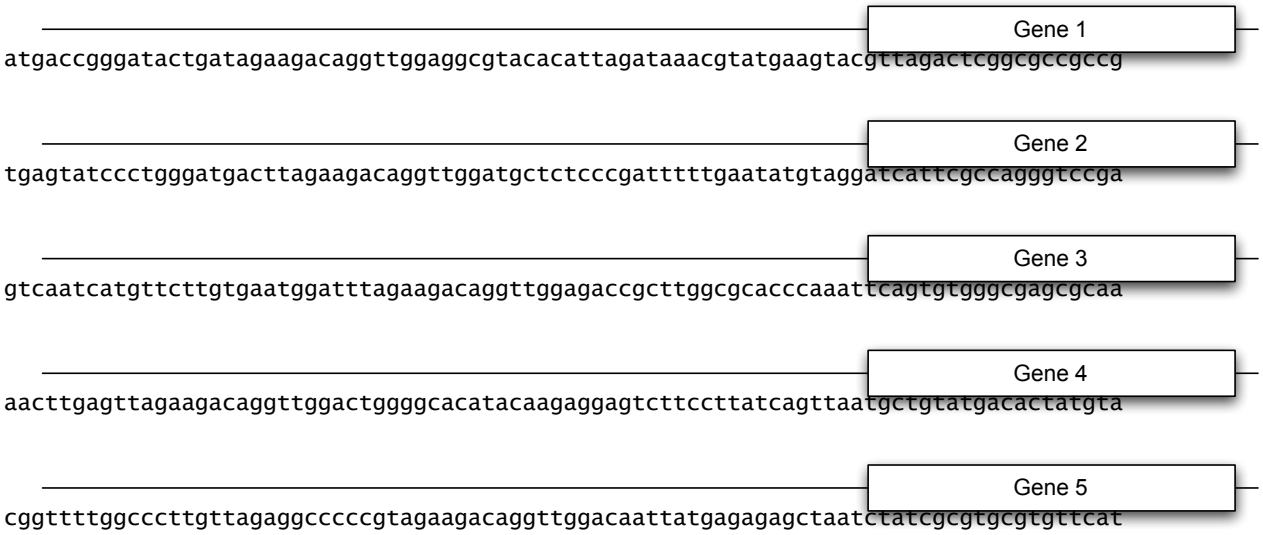
Regulatory Regions

- Every gene contains a *regulatory region* (RR), typically stretching 100-1000bp[3] upstream of the transcriptional start site.
- Located within the RR are the *Transcription Factor Binding Sites* (TFBS), also known simply as *motifs*, which are specific for a given transcription factor.
- TFs influence gene expression by binding to a specific location in the respective gene's regulatory region.

Transcription Factor Binding Sites

- A TFBS can be found anywhere in the regulatory region.
- TFBSs may vary slightly across different regulatory regions, since some (non-essential) bases[4] could mutate. We won't worry about mutation today!

[3] base pairs
[4] = nucleotides



We will be trying to identify strings that are common across all these sequences.

Summary

- Motifs are just over-represented patterns in DNA sequences: they occur more often than we'd expect by chance.
- The ones we are interested in are "upstream" of genes
- They're of interest because transcription factors bind upstream of genes in the *regulatory region* in order to start transcription, which turns the genes "on".
- Knowing where the transcription factors bind and to what small sequence of nucleotides enables us to understand which genes regulate which genes.

Complications

1. We do not know the motif sequence
2. We do not know where it is located relative to the genes' start
3. Motifs can differ slightly from one gene to the next (but not today)
4. How to discern it from a "random" motif?

On the other hand, in practice we know that TFBSs are not very long: we might only be looking for a match of about 10 bases, and we may even know something about the motif itself, in which case we're trying to solve the *site search problem*. Here we do not have such information, so we're trying to solve what's known as the *sequence motif discovery problem*.

How can we find them?

Let's begin with just two strings.

We are lucky that the alphabet of characters available has just four letters, A C G and T, but really we just care about matches and mismatches.

Matching Two Strings

First let's ask some simple questions to see if we can get our heads around this problem enough to begin solving it.

- How long does it take to show two strings are identical?
- How long does it take to show two strings are *not* identical? *Not* the same thing at all.
- How long does it take to find the longest common substring in two strings? That is, the longest perfectly matching string that occurs in both sequences. As mentioend, we're not concerning ourselves today with the major challenge introduced by the presence of variation in the motifs.
- ... in k strings?

Showing they're the same

Algorithm 1: SAMESAME(s, t)

```
1   given s, t, sequences of length n, m respectively
2   for (i = 0 up to n − 1) do {                          // i is a starting position
3   ·   for (j = 0 up to m − 1) do {
4   ·   ·   x ← 0                                        // x gets the value 0
5   ·   ·   while (s_{i+x} = t_{j+x}) do {
6   ·   ·   ·   x ← x + 1                                // increase the count
7   ·   ·   }
8   ·   ·   if (x > max_match) then {
9   ·   ·   ·   max_match ← x                           // we've found a new max. length
10  ·   ·   }
11  ·   }
12  }
13  return(max_match)                                     // Or we could return the positions in s, t
```

How long will this take?

Testing SAMESAME

Suppose $s = t = AAAAA\ldots A$, and $n = m$ gets really big. Here "really big" means "as big as we like". In practical terms n is likely only to be about 1000, but the number, k, of sequences could increase to hundreds of sequences.

Then we check about n^2 pairs of positions (i, j), even though the maximum match is found first.

Checking each match in this case takes about n operations[5] too.

Overall this is an n^3 algorithm. If n is 1000, this will take about 1000,000,000 operations...

If we were to use the same algorithm for k sequences, this case would give n^{k+1} time complexity[6]

Of course things won't always be this bad. We could look at a different sort of case, which is the *expected* time taken. To keep things simple we will have two sequences s and t as before, and they will have the same length, n. The "expected" part will come from the two sequences being completely unrelated to each other. This isn't exactly what we'd expect in real life, when we'd probably have somewhat related sequences, as we'd be comparing them only if we had a good reason to do so. However, the arguments will follow.

We will work out how many times we expect to increment x in the algorithm above.

For two random sequences of four nucleotides (A, C, G, T) we expect them to mismatch more than match: in fact the expected length, m, of a matching string between them is given by

$$m = \frac{1}{4}\frac{3}{4} \times 1 + \frac{1}{4^2}\frac{3}{4} \times 2 + \ldots + \frac{1}{4^k}\frac{3}{4} \times k + \ldots = \frac{3}{4}\sum_{i \geq 1}$$

[5] The amount of time required grows linearly with n

[6] Roughly how the time taken will grow with n.

which some mathematical manipulation shows comes to about $\frac{1}{3}$. (The first term is the probability that a sequence matches in the first position and then mis-matches in the next, times the match length (1); the next term is the probability that the first two characters match and the next one does not, times the length (2), etc.)

Exercise 1 (approx. 15 minutes) : Find a way to show that m is 1/3. (If you can program, write a program for this: if you can't, then you could use a spreadsheet or *maths* ☺.)

Exercise 2 (approx. 15 minutes) : In many cases the base frequencies are not uniform: Cs and Gs might be relatively high (but usually about the same as each other), such as where the sequence is 30% C, 30% G, 20% A and 20% T. Extend your method above to deal with unequal base (A, C, G, T) frequencies.

LESSINSANESAME

SLIGHTLYBETTERSAMESAME(s, t)

```
1   given s, t, sequences of length n, m respectively
2   max_match ← 0
3   for (i = 0 up to n − max_match − 1) do {
                                        // we can't get a match longer than what's left in s
4   ·   for (j = 0 up to m − max_match − 1) do {                    // same for t
5   ·   ·   x ← 0
6   ·   ·   while (s_{i+x} = t_{j+x}) do {
7   ·   ·   ·   x ← x + 1
8   ·   ·   }
9   ·   ·   if (x > max_match) then {
10  ·   ·   ·   max_match ← x
11  ·   ·   }
12  ·   }
13  }
14  return(max_match)
```

Dealing with multiple sequences

This seems bad enough so we could stop... unfortunately we usually need to deal with more than just two sequences. If we have k, then the above method would require k nested loops: really not efficient!

So now we have a more general problem. Given a set of sequences, how long is the longest substring that occurs in all of them?

Exercise 3 : (**This is a really hard problem. In my class I don't expect students to necessarily be able to solve it!**) (*i*) In groups, come up with a method that you *don't* expect to take a very long time, to solve this problem for multiple sequences.

(*ii*) Then nominate one person to explain your algorithm to everyone. Groups will be chosen at random to explain their solutions, based on available time.

Enjoy!