Bioinformatics

Contents

01.03.2023 Lesson 1

What is bioinformatic?

Interdisciplinary field that develops methods and software tools for understanding biological data, in particular when the data sets are large and complex.

Biological sequence analysis is what we are going to study.

Computational part is relatively easy.

Functional annotation.

Comparing sequences is useful for:

▪ Finding functional relationships

▪ Identifying gene and protein families

▪ Building phylogenetic trees

▪ Identifying protein characteristics, such as their organization in domains, functional sites, regulatory sites

▪ Building protein three-dimensional models

▪ …and many more

Compare sequence is important now that we have sequencing platforms that can give us a very large amount of data.

If I look at human and a mouse gene, they are similar if they originated form the same gene and same protein present in the same ancestor.

2 sequences: ACGATCGAGCC…

If I align 2 sequence what I am implicitly saying is that these As for example are also the ones found in the ancestors.

Substitution: the case in which there are one G and one T, means that they are evolving.

Gaps: one sequence has extra characters -> deletion/insertion (acquire/lost one character)

Different kind of event can happen during evolution.

These comparisons are based on an *inference of the evolution*, that can be much more complex that what we can infer just by looking at alignment; we should have the ancestor sequence.

So, we need to compare sequences to identify evolutionary relationships.

- **Homology**: applied to biological entities that originated from the same sequence
- **Divergence**: accumulation of differences – subject to selective pression

NON COMPLETO

Sequence comparison

02.03.2023 Lesson 2

Our objective is to compare modern sequences considering that they have common ancestor, reconstruct evolutionary history.

Compare sequences such that given their alignment there's a relationship of homology.
Goal of comparing: find relationship between two sequences:

- compare pattern of mutations
- check conserved regions
  conserved regions are important features, some might not be perfectly similar but their mutations might not have impact on the function.

Each form of homology can have different type of conservation.
If two sequence are similar enough we can say that there's a relationship, but there is also **convergence** so they could not come from the same ancestor anyway. The sequence in this case would be considered **analogous**, similarity but with different origin.
Usually does not regard entire proteins/genes, but just a part. i.e. catalytic site exactly the same

Synonym mutation instead, do not change amino acid sequence, so they're not under selective pressure (neutral mutations). These usually happen in ncDNA, but they could also happen in cDNA.
Since genetic code is redundant (different codons encoding for the same amino acids), if we have a mutation for G in alanine we will obtain alanine anyway, etc.

If you compare sequence of homologs, you can count different synonym and not, and checking for different mutations family, you can see that mutation in some proteins are easier to occur.

What we see has gone through selective pressure, so that feature survived selective pressure. Why some mutation survives selective pressure and other not?

Chance to survive depends on what is mutated in amino acids, because they can have different characteristic from the original amino acid.
(Some amino acids are small, some are aromatic, some are hydrophobic etc)

Varying the amino acids but remaining in the same group should not have big changes (from lysine to arginine it is ok because they're in the same group, higher chance to survive)

Chance to survive also depends on where the mutations happen.
If the mutation happen in the core of the protein is less likely to survive (maybe affecting folding stability, affect the packing, etc)
On the surface, the mutation is more likely to survive – also higher chance to see deletion or insertions.

General rules:

- if 2 sequences are sufficiently similar this implies that they have the same evolutionary origin.
  It is not true in the other sense: it is possible that something accumulate so many difference that you can't recognize its origin and might become similar to another.
- If 2 sequences are sufficiently similar same 3d structure: sometimes protein can fold in similar way even if they don't have common origin (analogous).
- 2 protein that fold the same way usually have same functions.

Anyway, there are many exemptions to these rules.

Assuming that accumulation of differences occurs linearly in time, then if we report the number of amino acids substitution the rate increases in the time.
For algorithm is important to align with biological meaning, they are strings, but not just strings!

Strings

Offsets is the difference between an index position and a position that character occupies in the context
There might be a difference in the relative position of residues in a biological context with the index position

Operations on strings:

- Concatenation
- Substring
- Prefix substring (slice) starting at index 0 and ending somewhere within the string.
- Suffix substring ending at the end of the string and starting somewhere.
- String matching find the occurrence of P in T EXACT match, but we are looking at similar string.
- Naïve (brute force) algorithm algorithm that try to solve a problem by trying all the possible solutions and see the best one.

08.03.2023 Lesson 3 (from slide 7 in Lesson 2)

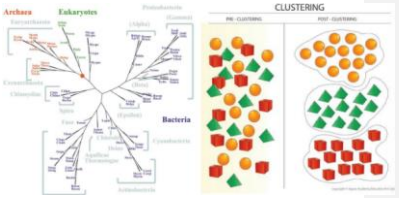
Similarity

Similarity counts how many characters are identical between two strings.

It is useful to understand if there's biological relation between two sequences. It depends on the length, so it's also sensitive to the statistical evaluation.
It is intuitive and easy to get a statistical evaluation, others are less intuitive.
Using distance, in the example with these two cases, it returns always 0, regardless on the length of the identical sequences. (Distance is not statistically meaningful)

Hamming distance

It's important to do clustering, you need to from classes with similarities to differentiate organisms.
Hamming distance is the simplest measure of string distance,
it defines the number of positions at which the two strings differ.



The problem with the hamming distance is that it considers only the number of substitutions and is not able to detect deletions or insertions.
It is then important to introduce the concept of gaps and mismatches to our algorithm.

Edit distance

Edit distance between two strings is defined as the minimum number of single-character operations required to transform one string into the other.

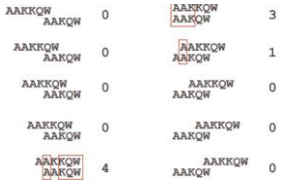
Edit distance can handle substitutions, deletions, and insertions of bases.
These three distances together are needed to consider the shortest path to make the two compared sequences, similar.

On virtuale there is a video to compute the distance of two sequence over time

While computing the hamming distance is trivial and can be done in linear time, computing edit distance is very difficult, and must be done in polynomial time, usually in O(mn).

Distance can be a measure to compute also as **similarity score**:

1. Compare the different alignments of the sequences and find the best one (one with more similarity)
   In this case we evaluated 10 alignments, compared 30 characters, it is NOT EFFICIENT enough.



To generalize, to compare two strings of length *m* and *n*, respectively, with no gaps, the number of possible alignments is:

$$N_{all} = m + n - 1$$

While the number of ch

As another example, this is the best alignment that can be obtained by scanning one string with the other, in all possible overlaps

```
IPLMTRWDQEQAAAAAAADD
||||||||||| 11
IPLMTRWDQEQDD
```

By deleting some characters (i.e. introducing *gaps*) we can get a higher number of matches

```
IPLMTRWDQEQAAAAAAADD
||||||||||  ||
IPLMTRWDQEQ-------DD   13
```

2. Use **gaps** in order to maximize the best score possible.
   When using a gap, you will generate also a negative score, it has cost -1 for each deleted character.
   This is because we're not considering the evolutionary events explaining the differences between the strands if we only consider the similarity.

Problem is: How well the scoring function describe evolutionary distance.

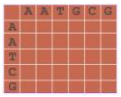
The optimal alignment between two sequences, given a similarity scoring function?
Usually it's always required to scale the problem down until the solution is immediate.

Dot matrices or dot plots

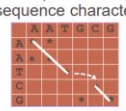
It's a simple visual way to see how two sequences merge.

First, for two sequences of length *m* and *n*, let's build a matrix having dimensions *n* x *m*

Seq1 = AATGCG Seq2 = AATCG



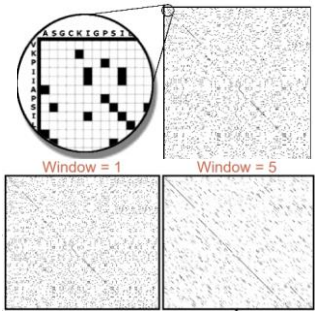
Then, for each cell in position *i* and *j*, if the corresponding sequence characters are identical, let's put a dot



AATTCG
AAT-CG

The tracts in the two sequences that are identical appear in the matrix as a contiguous series of dots in a diagonal. By connecting the diagonals with a *jump* we get the alignment

It is useful to visually interpret the identity between two sequences, since identical tracts are easily visible as diagonal tracts, and connecting them by jumps gives an idea of the two sequences alignment. Each jump corresponds to a gap, i.e. insertion or deletion.

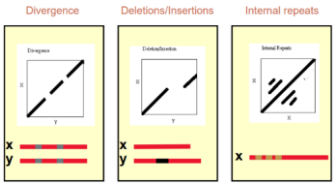


This process will generate much **background noise**, since most of the dots in the matrix are due to chance identities, especially for strings composed by a limited alphabet.

To limit this phenomenon, we can consider windows of fixed length.
i.e., window size = 5, so we will draw a dot only if 5 contiguous characters of one sequence are identical to 5 continuous residues of the other.

This reduces the number of highlighted cells outside the diagonal.

What we look for in this case is divergence, deletions/insertions, and internal repeats.



To plot the dot matrix on py, you need to use lists or list of lists (for cubic matrix)

Dynamic programming

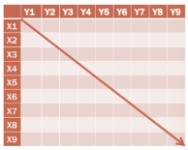
A class of algorithms generally called dynamic programming are used to compute the optimal alignment:

- Needleman & Wunsch 1970
- Smith & Waterman 1981

They work based on two assumptions:

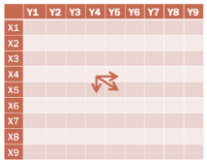
- Each column of the alignment is independent from the other columns (not always true)
- Basically, split the problem into smaller subproblems to solve in a more easy way, this holds true since it can be proved that the alignment problem has an optimal substructure.
  If one can extend in the optimal way a partial alignment of two subsequences that were already aligned in the optimal way, the resulting alignment is optimal.

The algorithm must be able to do what we saw in the dot plots -> finding the best path in the matrix without having to generate all of them.
Each cell in this matrix, instead of just a dot, contains the score of the optimal alignment up to that point, and we just need to extend it further one cell at a time until the whole matrix is filled.



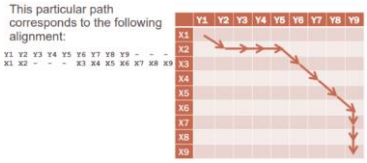
Each path in this matrix correspond to a different alignment between the two sequences. They usually have to start from the first cell and end in the last one so all the residues will be included. (global alignment, we'll define it later)

The rule is that there are only three possible movements:



- From cell [i,j] to cell [i+1, j+1]
- From cell [i,j] to cell [i, j+1]
- From cell [i,j] to cell [i+1, j]

The first correspond to the addition of two residues, one for each sequence, piled one on top of the other, to the growing alignment. The other two movements correspond to the addition of one residue from one of the two sequences, and one gap for the other (step correspond to a mismatching in the sequences).



Hence, each alignment has its score, depending on the path itself, we need to find the best one.

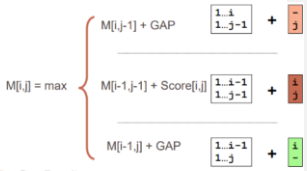
The number of paths in this matrix is almost exponential. I don't want to go through all of them and analyse all of them. We need a way to make it go the optimal direction.

It can be proved that, given two sequences having length m and n, and a scoring function to be maximized, the optimal alignment, the one producing the highest score, can be computed from:

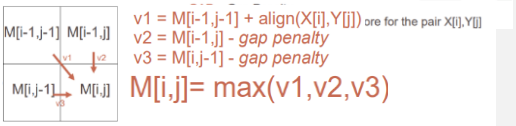
1. The optimal alignment of the first m-1 characters of the first sequence with the n-1 of the second;
2. The optimal alignment of all the m characters of the first sequence with the n-1 of the second;
3. The optimal alignment of the m-1 characters of the first sequence with all the n characters of the second

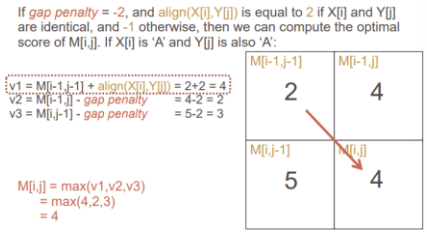
This might not seem particularly useful, since at the beginning we don't know any of these three values. Nevertheless, the same property holds true for any pair of characters i and j of the two sequences, independently on the position of i and j.

When we have an optimal alignment for two subsequences, we just need to be able to extend this alignment in the optimal way. It is possible to understand which is the next optimal alignment by knowing all the three possible extension result. The score is computed from the score of a match, the cost of a mismatch and the cost of a gap.
So it is important that the previous best scoring cell keeps track of the score value in order to be able to pass it to the next one for the computation of its score.

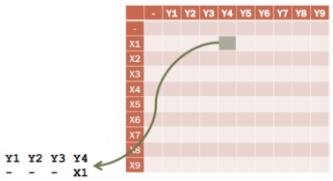


These are the simplified equations to get the score:





How to start from [1, 1]?
It has no neighbouring cells, so we need to add a column and a row.



Now if we want to start from the previous green cell, you'll consider that they must be gaps.

If the characters are the first of X and the fourth of Y, this means that we need to create a three-character gap to fill the alignment.

Basically, progress expands the problem from the one which has obvious solution.

09.03.2023 Lesson 4

Each cell in the matrix can be reached from 3 neighbour cells. Each one of these three neighbour already contain the score of best possible alignment since then. To get the value of the cell [i,j] we need to calculate the score in the case of the 3 events that lead to that cell, get the maximum between them and give it to the cell. This is **true for each cell**. This means that the last one will contain the maximized scoring function.
To compute the optimal value of the first cell instead, since we are not having any cells with a value in its neighbour, we have to imagine that:
in case of the cell [X1, Y4], if we consider that one, we will have that no diagonal can reach it so all the previous values Y will be considered as gap comparted to X string.

This leads to the introduction of **progressive gap penalty**.

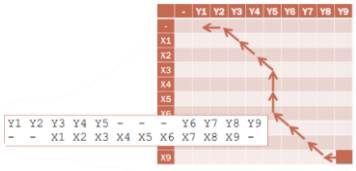
Once these additional first row and column are filled (**matrix initialization**), we can proceed with the rest of the matrix.

- If the gap penalty is -2, the values of the cells in the additional row and column is the progressive sum of gap penalties

	-	Y1	Y2	Y3	Y4	Y5	Y6	Y7	Y8	Y9
-	0	-2	-4	-6	-8	-10	-12	-14	-16	-18
X1	-2									
X2	-4									
X3	-6									
X4	-8									
X5	-10									
X6	-12									
X7	-14									
X8	-16									
X9	-18									

Traceback

Since we also want to see which residues are similar, and which are not, in order to understand which part are under selective pressure; we have to start from the last cell and reconstruct how the value in this last cell was computed, so we can reach again the path that led to the value.



The optimal alignment must pass through one of the three neighbours, understanding whether it is in the diagonal or in a gap.

Stages of the algorithm

- *Initialization*: adding of the two columns with progressive gap score
- *Get score*: get the optimal score going from the top left to the bottom right
- *Traceback*: construct the alignment that led to the optimal score going from the bottom right to the top left.

This algorithm is an **exact algorithm** since doesn't matter how many times you run it, it would get you the same result.

Usually, we work on 2 equal matrixes, one to compute the best path, the other to trace the path.

It might happen that two direction have the same score:

- random choose the solution, not very correct, it might exclude the optimal path.
- check the full pathway of the two equally valued scores and then choose the right one.

This algorithm is the **Needleman & Wunsch** (NW), and it is of **global alignment**. Alignment between two strings that include all the characters of the two sequences, it is useful especially from quite divergent sequences.

Smith & Waterman (SW) works as a **local alignment**, which retrieve the optimal substring of the two strings, the part of the sequences in which they're more similar.
This is needed because biological sequences might be considered **MODULAR**, each of which could have had a different evolutionary history. It is useful because it's possible to find better the similarity between them.

Compute a local algorithm

In a global alignment we always start from the bottom cell.
For a local alignment we start from the cell containing the maximum score which can be anywhere in the matrix, and it correspond to the end of the substring of the two strings which is most similar.

Needleman & Wunsch (NW) = GLOBAL alignment
Smith &Waterman (SW) = LOCAL alignment

Anytime we try to compute the optimal value of the cell, and it is based on a simple observation:
negative scores within the matrix that are included in the optimal path indicate strong divergence, while positive scores indicate a likely evolutionary link.
In case of a strong divergency (*lazy negative scores*), it could be useful to reset the matrix in some way, such as removing these regions of strong divergency so that only the ones that are similar can be compared without affecting the quality score.

This process is done by adding one more condition:
If the three other scores are negative, we will take 0 as the value in the [X, Y] cell. This allow resetting the matrix each time there's divergence.

So, in this case the initialization is done by adding the first column and the first line and fill them with only 0. This allows the score not to be affected by the initial gap that could be present between the sequences.

The result will be a matrix without a negative value.
There will be at least a value with the maximum score, and from it we can start traceback the best path. It is easy to trace because it will always follow the higher neighbour to follow the path, and it will always lead to the best local alignment.

The other difference is also the way to stop the traceback. It happens when all the three neighbours of the cell are 0. Because it means that up to that point the two strings are different.

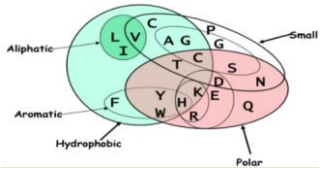
To summarize, we described three approaches:
- Dot plot:
  - Visual exploration of sequence similarities
  - Visual identification of insertion/deletions and repeats
- Global Alignment:
  - Comparison of full-length sequences
  - Identification of extended evolutionary relationships
  - Works better for close species and/or conserved sequences
- Local Alignment:
  - Identification of the subsequences with the highest similarity
  - Identification of localized evolutionary relationships (e.g. a domain)

It's also important to mention that the local alignment take in consideration only the highest rated alignment, but there could be also other meaningful ones. In order to retrieve them it's just necessary to delete the cells of the first best alignment and calculate again.

Scoring function

It's important to maximize it in order to improve the biological meaning of the program.
Since this moment we used +1 for the diagonal, -1 mismatch, -2 gap. Changing these parameters will change the score function.
To evaluate differences and identities in sequencing with scores, it's important to reflect in some way the chance that a mutation change in one amino acid into another one is not discarded by selective pressure.

We can then divide amino acids into different groups based on some classification to attribute different values if the difference between the two sequence is between amino acids of the same group or not.



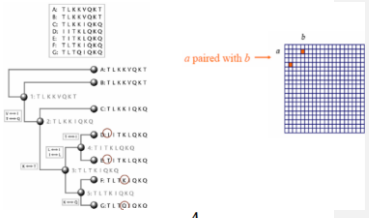
Also, the alignment of the same amino acid on the two sequences, has to be evaluated depending on the amino acid. For example, if Try is aligned, it need to have a different score than Ala. Just because the Try molecule is a rare and specific amino acid which functions cannot be substituted by other amino acids, so it is more likely to be conserved.

Substitution matrix

What we need is the cost of substituting an amino acid with another, by counting its **substitution frequencies**, i.e. the number of times that, during evolution, a given amino acid was mutated into another, and this mutation was tolerated by selection.

The substitution frequencies compose the **substitution matrix** (20x20 matrix) in which each column and row correspond to one different amino acid. The values in each cell correspond to the substitution rate between the two amino acid.
The matrix is than integrated in the scoring function.

The first substitution matrix had been computed by **Margaret Dayhoff**, the first bioinformatician.
She and her co-workers manually built protein pair-wise alignments of clear homologs, and counted the number of times each amino acid was changed into another, and how many times it didn't.
It is possible to create a substitution matrix because the differences found, since they're from homologs, have mutations which are related to the differences in evolution.



The target matrix had then been converted into frequencies by calculating the ration between the number of times a substitution Aij was observed, normalized by the sum of all substitution involving amino acid i.

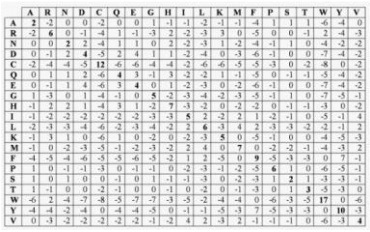
$$p_{ij} = \frac{A_{ij}}{\sum_n A_{in}}$$

$$s_{ij} = \log \frac{p_{ij}}{p_i * pj}$$

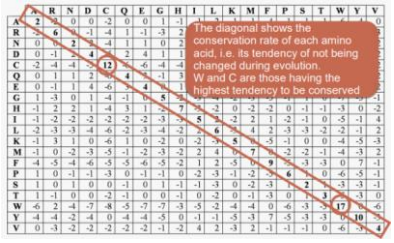
To convert this value in a more comprehensible one, they did then the log of this probability over the product of the frequency of amino acid i multiplied by the frequency of amino acid j.

The interperetation is that this ratio between the obsterved number of substitution of that type over the expected number of substitution i and j.

- If **Sij > 0** it means that the observed number of substitutions are more frequent that what we would expect by chance. So, they are **more likely to be tolerated during evolution**.
- If **Sij < 0** it means that observed number of substitutions are less frequent that what we would expect by chance. **Strongly counter selected**.



10.03.2023 Lesson 5



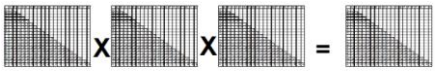
Taking in consideration the diagonal of the matrix, we are able to retrieve conservation scores, which means how many time an amino acid remain the same (it is conserved). It is not the same for all amino acids since some of them tend to tolerate more change because they don't have unique specific functions, so it might be ok to substitute them with a similar one.

If you want to compute the score of any given alignment you need to scan the alignment position by position, check which are the pair and find the corresponding value, then just sum them.

Matrix product

The way a protein evolves compared to a different protein under a different selective pressure evolve, might be different. And substitution rates that we observe in very conserved proteins might be different from other protein that can change more freely.
Since the original substitution matrix was computed on a set of very similar protein sequences, with very clear evolutionary relationships, the matrix scores are not suitable for aligning more divergent sequences.

To solve this issue, they used a mathematical shortcut to an higher degree of divergence.
Starting from a substitution matrix computed on the very conserved protein family identical at 99% of their amino acid sequence. This matrix had been called **PAM1** = 1 Percent Accepted Mutations. The one stands for the rate of divergence of the protein used for building that matrix.
The values correspond to one evolutionary step.



Multiply matrices N times to obtain PAM 'N'

Starting from the original substitution matrix (identical 99%), they multiplied for itself to get the PAM2, PAM4, etc. In this way they developed evolutionary matrix which should be similar to what we would expect from an alignment of more divergent sequences. Simulating further evolutionary steps.

If I could compute the values of alignments of very divergent sequences, we would expect many variance. So, conservation should decrease, along with the rate of substitution.

PAM	0	1	30	80	120	200	250
% identity	100	99	75	60	50	25	20

So, the observed frequency of each change should be more similar to the expected frequency of change.

For example, if you have sequences with 20% identities the best matrix is PAM250.

In general terms the substitution scores of the original matrix computed are pretty much universal.

A criticism of the PAM matrices is that the substitution rates are computed pulling together data from entire sequence alignments, which implies that these rates are constant regardless of where in the proteins they are found.
This does not reflect how evolution operates, since the selective pressure is different in different protein parts depending on their functional and/or structural role.

Moreover, by selecting very close sequences and looking at the most variable positions, it is likely that the model is biased towards more tolerant positions, while evolutionary rates should be better computed on truly neutral mutations.

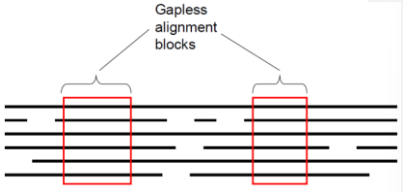
Finally, they were originally computed on a relatively small dataset, using those protein sequences that were available in 1978, and preferring globular proteins, therefore it was not clear how general are these scores.

BLOSUM

It's the substitute of the matrixes PAM.
In 1992, Henikoff S & Henikoff JG proposed that better matrices could be computed from parts of divergent protein sequences that can be still be aligned without the need of gaps

The blocks should correspond to the protein parts under stronger evolutionary pressure. From these blocks, substitution rates can be then computed in a way similar to that used for the PAM matrices.

These matrices are called **BLOSUM** = Block Substitution Matrix



If you see part of the alignment in which there are many gaps, it most probably is a domain in the surface, which is less conserved.
The ones without gaps instead, are probably the sequences corresponding to the ones of the core, structurally important part of the protein.

So, the idea is to compute the substitution rates only within the chore of the protein, but we cannot do that because we don't know the 3D structure. The solution is understanding it by sequence analysis, finding blocks of very conserved regions.
From these blocks, which might contain substitutions, we are going to compute substitution rate.

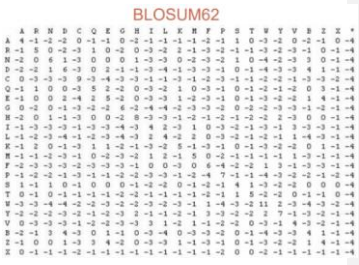
To recap

Get sequences from a family of organisms, align them (tricky part because you need to use a PAM matrix) and they should be different, have some gaps (not too many).

You can then compute the frequencies of changes q(i,j), undergo some sort of normalization similar to how PAM matrixes are computed.

In BLOSOM matrices, they wanted to create matrices to better compute sequences with some degree of divergence.
To compute those matrices they did not use mathematic tricks.
Their idea had been to start from blocks of protein families in which there are no gaps. From these they counted distances and the percentage of identity.
If I can classify each block I can classify each degree of identity or difference. So then it will be possible to compute substitution score only for blocks having some degree of divergency.

i.e. if I compute substitution score for a block with 50% identity, I will get a BLOSOM matrix which could be used to align sequences with 50% identity.



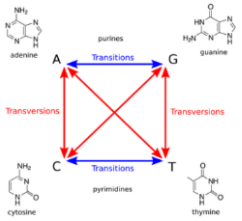
The value of the BLOSOM matrix is the percent of identity of blocks used for computing that matrix.

You can directly compute different matrices with different level of divergency.

The concept of a substitution matrix does not apply to nucleotides.

Tolerance or rejection of a mutation changing one nucleotide into another usually does not depend on the physical or chemical nucleotide characteristics, but on different kinds of selective constraints.

There are exceptions, for example related to the flexibility or rigidity of the DNA double helix, which can have functional roles and that depends on particular nucleotide compositions or related to DNA denaturation.
(i.e. separation of the two strands of the double helix, which depend on the content in G and C)



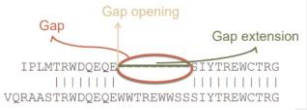
If you attempt to understand how organism evolve, it's possible to see a sort of pattern. It is possible to see that some variances are more frequent than others (transitions), but it's more difficult to convert it to score.
The problem is that not in all organisms transition is more frequent, so the score might be not appropriate.
Nowadays, we still haven't found a solution to calculate the score of the nucleotides because of these problems, so no substitution matrix.
For the moment we rely on arbitrary sequencing scheme: positive score means identical residues, negative for differences.

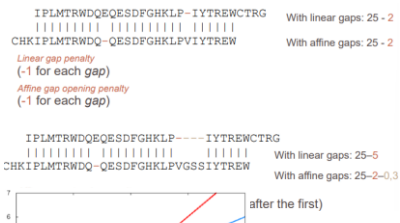
- Proteins -> substitution matrix.
- Nucleotides -> arbitrary score schemes based on degree of divergence.

Gap penalty

There are different ways for penalise gaps.
Up to this point we used **linear gap penalty**: each gap character has some negative score, so the overall penalty for a gap of n size increase linearly depending on n.

It is possible to have alternative gap penalty, but this require a more complex align algorithm.
Affine gap penalties, takes in consideration 2 things: each time there's a gap there's one *gap extension penalty* assigned regardless the length, the other is the *gap extension penalty* which depends on the length.



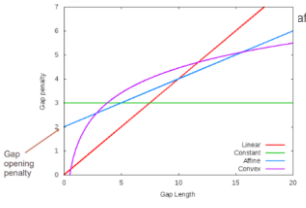


The affine gap penalties are better because they're sensitive to the number of gaps / length of the gap. The parsimony principle is applied to represent better the reality of evolution.

```
AAAGAATTCA
A-A-A-T-CA

AAAGAATTCA
AA----TTCA
```

With a simple gap penalty, these two cases would receive the same alignment score, but we need to consider that each gap correspond to a different evolutionary event, therefore 4 separate gaps indicate a stronger divergence that one single gap, even if it is larger.

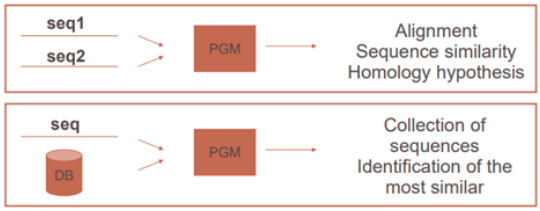


The **convex gap penalty** is a convex function.
GO stands for Opening Gap, GE for Gap Extension

$$Gap(L) = GO + GE \log(L)$$

It rates the intermediate length gap higher and less the ones which are longer, with respect to the other functions.

To take into account these kinds of penalties we need an algorithm that must know if you're adding a gap to an already existing one or it's a new one.
To solve this they thought about a cubic matrix, in which the other axis takes into account the gap. But with a cubic matrix complexity increase very much.
One with polynomial quadratic development is Gotoh, which is very nice, but we don't have to study.



Up to this point we studied pair-wise alignment. From the score of the alignment you can understand if they're homologous or not; indeed you want to verify that there might be some kind of evolutionary relationship which might also implies functional relationship.

You start by looking at two sequences, from which you already have some kind of evidence that they might be related.

But what you usually have is just one sequence, so we align this sequence with another sequence (you need evidence that they might be related) contained in a database. Indeed, you have a collection of sequences, and you want to identify in this database which sequences are the most similar to the query sequence, so you can get all possible information.

How can we approach this? Compute the optimal pair-wise alignment between my sequence and all the sequences in the database. However, these databases are often huge, so it will require a lot of time and the majority of these alignments will be quite useless. So, we don't want to align all the sequences present in the databases, but we want to select from the database sequences that might be related in some way to the sequence of interest, and then discard the ones that are less likely to be homologs of our sample.

This selection must be done based on some reasonable assumptions in order to get the homologues sequences that could be similar to the one of interest, but it risks not to get all the possible homologues, but this is the cost to pay to speed up this procedure.

A consequence is that this algorithm cannot be considered exact anymore. Optimal result is not guaranteed.

These kinds of algorithms are called heuristic. A heuristic is any approach to problem solving that employs a practical method, not guaranteed to be optimal, perfect, logical, or rational, but instead sufficient for reaching an immediate goal. Where finding an optimal solution is impossible or impractical, heuristic methods can be used to speed up the process of finding a satisfactory solution. A good heuristic usually works well, but can lead to systematically wrong outcomes, sometimes called biases (if the assumptions are reasonable then the solution might be useful, otherwise the solution might be biased in some way).

In bioinformatic heuristic algorithms are often used, because for many different problems there is not an exact algorithm to solve them. To have an idea of how far our result might be from an optimal solution, often the solution provided by these algorithms are supported by statistical evaluation.

Not for all heuristics that are used in bioinformatics you can do statistical tests to evaluate the solution, but at least for sequence database resources we have a solid way to evaluate the result.

The most commonly used heuristic algorithms for sequence similarity search in sequence databases are:

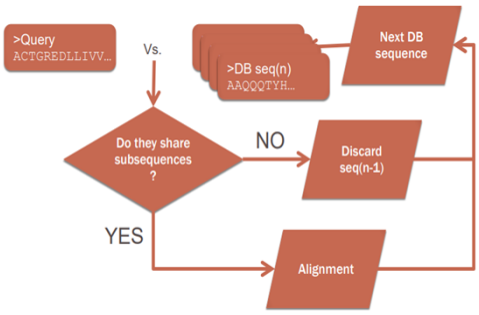
- FASTA
- BLAST

FASTA was publish in 1985, it is still used mainly for nucleic acid sequences, for which it seems to perform a little bit better than BLAST, which is more often used for protein sequences. FASTA is slower than BLAST, and it is based on assumption, which are performed during the filtering step at the beginning of the procedure and that are stronger than the ones in BLAST. The result is that we could have stronger biases.

FASTA stands for **FAST** for **A**ll kind of sequences, it is a general purpose alignment, a database search algorithm for any kind of sequences (protein, RNA, DNA…). The idea is to have a heuristic based on filtering procedure with the goal of finding promising leads, that is sequences in the DB that are more likely to be homologs to the query. After this initial selection, all sequences which don't satisfy this initial filter are discarded, and only the more promising leads will be

aligned to the query. A very small fraction of leads survive this filtering procedure, and the alignment, that is the computationally heavy step, is done only for some targets that satisfy some requirements, saving the time needed to align the query against targets that are not likely homologs.

The gain in execution speed is then only due to the fact that the optimal alignment is not computed with all DB sequences, but only with a subset, more or less large, of them.



What is the assumption on which the selection step is based? The assumption is that is two sequences are truly homologs, no matter how distant these two sequences are in evolutionary terms, we would still be able to see some identical substrings.

I take a sequence from the DB, I decompose the sequence in substrings of size k (called k-mers). If the database sequence shares some of these k-mers with my previous sequence, then we are going to perform the alignment and then pass onto the next DB sequence, otherwise you discard the sequence and go to the next one.

There might be some sequences in the DB that by chance have some substrings of length k identical to the query, but just by chance only. Therefore, this filtering selection might include many false positives, many sequences that are not related to the query. But it is not important, because when we perform the alignment, it will show that a part from this identical substring the rest is different.

How often I am going to perform the alignment depends on many different factors. One of them is obviously how many homologs of the query sequence are present in DB, but it also depends on the length of the identical substring that I am looking for. The value of k can usually be set up by the user.

If k is large (so if I require that the two sequences share a large substring) then I am going to possibly avoid the substrings share by chance only, this means that this filtering step is going to discard many sequences and I am going to perform the alignment in very few cases, speeding up the procedure. However, if k is too large, the risk is to lose many of the true homologs. On the other end, if k is too small, I am going to identify all the true homologs present in the DB for sure, but also I am going to perform the alignment for many sequences that are not related to my query sequence and share that substring just by chance. I will have an input that is more complete, closer to the optimal one, but at the cost of having a much longer run. Hence, balancing the value of k is not easy. There are some default values that are fine in most cases, while sometimes you have to manipulate them.

How to identify if two strings share a common substring is relatively fast, but this process must be repeated for a huge collection of strings. So, this is usually done efficiently using hash table.

Hash Table

We should imagine hash table like a python dictionary (pairs of keys and values); a way of storing variables associated with an identifier so that it is possible to retrieve the value quickly. In this way it doesn't matter where the key is, it would take the same time as any other key to retrieve, it works in **constant time**.

How can I fetch the value associated to a key? One way would be just to store in a list a pair of tuple, and then, to fetch the value of a given key I just have to iterate the list of tuples until I find a key corresponding to the one I am looking for and then I fetch the corresponding value. This process is not so slow, it is done in linear time depending on the length of the list/array. Linear time is good if n (size of the list) is small, but if it is applied to a huge collection of key:value pairs it still requires a lot of time.

With hash table instead this process is done in constant time, so it does not depend on the size of the dictionary.

When we access FASTA, database is already prepared into data structures, such that the filtering initial step of looking which sequences in the database are promising leads is done quickly (don't need to reprocess database every time).

For every value of k, a hash table is created so that it is easier to reach it and retrieve the information.

T = "GTGCGTGTGGGGG"

3-mers of T	Offset of 3-mer
GTG	0,4,6
TGC	1
GCG	2
CGT	3
TGT	5
TGG	7
GGG	8,9,10

Suppose we have this sequence T, these are all possible 3-mers, all possible substrings of length 3 in which this sequence can be decomposed. For each substring, we want also to store the relative position of the 3-mer with respect to T.

3-mers could be found many time times withing the string as we can see in the case of GTG.

The keys are the 3-mers and the values associated to them are their relative positions, offsets.

So, if I am given a query 3-mer, I can scan this dictionary to retrieve if the query 3-mer is found within the string T and where.

This is not the best way of fetching a value associated to a key. We are iterating through all the possible key:value couples until we have found what were we looking for, and this would be done in linear time, while we want to do this in constant time, regardless the size of the dictionary.

Therefore, here it comes the hash table. It's the internal organization of the dictionary. Can be imagined as an array of values. When you want to retrieve the value associated to the key, you get the index position of the value directly. You don't have to iterate through this array of values, you can just know in constant time where is the value for that key.

How can we do that?
Consider the following function:

```
>>> def hashString(st):
...     s = 0
...     for c in st:
...             s = (128*s + ord(c)) % (2**120+451)
...     return s**2 % (2**120+451)
...
>>> _
```

This function takes a string as its input and returns an integer that is computed from that string, by doing some calculation on the characters of the input string one by one. The details are not that crucial. What is important here is the fact that this function maps any string to a number.

Python's built-in function ord is used to get the Unicode value of a given character. Each single character has an integer representing it. For example, "a" is 97, "A" is 65 and " " (space) is 32.

This function scans for each character in the string passed as argument, for each character get the Unicode and return a number. If we run this sample, for A we get 4225.

```
>>> hashString("A")
4225
>>> hashString("ATGC")
18961370141484681
>>> hashString("CGGTAGCATGG")
336467204693082933939964371009647291
```

The function hashString then maps infinitely many strings to a range of integers. More generally, a hash function maps a large (possible infinite) set of elements (numbers, strings, tuples) into a much smaller (and finite) range of integers (that can be also negative).

How is it used? An empty vector is built any time you create an empty dictionary. This list is going to be used for storing values. So, I have a key:value pair, which key is processed by hash function. From it I obtain an integer, and it is then used to set the position at which the value is going to be stored.

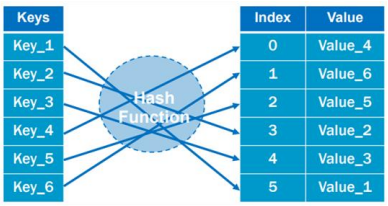
Example: I have a pair, and the key is a string. The value obtained with the hash function is 5. Therefore, 5 is the index position at which the value associated to that key is going to be placed. This procedure is repeated for all the other key:value pairs that I want to place in the dictionary.

Hash function is also used in the opposite case. Once the dictionary is created, so all the values are now stored in it, I can retrieve the value associated to a key. The key is again processed by hash function, and it is going to tell me the location of the value associated to that key. In this way you can fetch it, without scanning all the values in the dictionary. The time required for this procedure is independent from the size of the array – constant time.

Also, we are not sure that the hash function is going to give back different values for different characters (different substrings can point to the same position in the array). Since a hash function maps a large universe of elements to a smaller one, hash functions cannot be one to one – there must be different elements that are mapped to the same number. Such a scenario is known as a collision.

Collisions are unavoidable in our context, due to the well-known pigeonhole principle – when n pigeons are placed into m holes, and n > m, there will surely be a hole with at least two pigeons. For now, let's assume the collection of elements we hash is small enough that no collisions occur.

Database sequences are organized in hash tables in order to quickly fetch some sequences from it. A **hash table** is an **unordered collection of elements consisting of key:value pairs**. A hash table is implemented using a hashing function. Rather than sorting the data according to a key, it computes the location of the data from the key. In simple terms, it computes an index value from the key of the desired record. At that index value, the record (i.e., the value of the key:value pair) is stored/retrieved.



Let's imagine that we have 3 key:value pairs: 1:'a', 2:'b', 3:'c'. We want to store these pairs in an array of size 3.

We create an empty list (in this case we are also specifying the size of the array) and we define the hash function as h(k), that is equal to k % n, where k is a key, n is the size of the array, and % is the modulus operator, which returns the remainder of the division k/n.

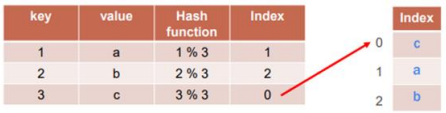
Now, let's start populating our arrays placing each value in the position indicated by their key.
In the first case key is 1. If I pass 1 in the hash function h(k) I obtain 1. So, the value 'a' associated to the key 1 is placed at index position 1 of the empty array.

For the second key:value pair, the key is 2. The number returned by the hash function is 2, so I'm going to place the value 'b' associated to the key 2 at the index position 2 of the array.

For the last pair, the kay is 3 and the number returned by the hash function is 0. Therefore, I'm going to place the value 'c' associated to the key 3 at the index position 0 of the array.

Now the array is populated with all the values of the key:value pairs at some specific position, as determined by the hash function (that uses the key to return index position).

Once I created this data structure, if I want to retrieve value associated to any key, I apply again the hash function on the key, and I will obtain the index position in which the value associated to that key is stored.



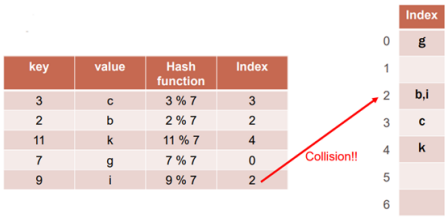
key	value	Hash function	Index
1	a	1 % 3	1
2	b	2 % 3	2
3	c	3 % 3	0

Index	
0	c
1	a
2	b

In this way I don't have to iterate through the array. As we said before, the time required for fetching values associated to a key does not depend on the size of the array. Complexity does not depend on the size.

Let's see another example.

Let's imagine now that we have 5 key:value pairs: 3:'c', 2:'b', 11:'k', 7:'g', 9:'i'. To store these values, we have an array of size 7. Let's use the same hash function h(k) = k % n as before, where k is a key, n is the size of the array, and % is the modulus operator.

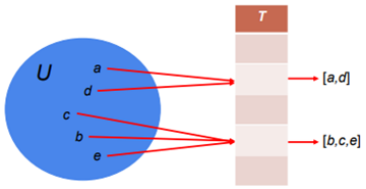


key	value	Hash function	Index
3	c	3 % 7	3
2	b	2 % 7	2
11	k	11 % 7	4
7	g	7 % 7	0
9	i	9 % 7	2

	Index
0	g
1	
2	b,i
3	c
4	k
5	
6	

In this case, while computing our table, we can see that the index position for the keys 2 and 9 is 2. Yet, the hash can be identical for different keys, implying that different values can be assigned to the same index: in this case we have a collision.

Collision can be avoided or minimized, sometimes increasing the complexity of the hash function (but a perfect hash function cannot exist!). Some strategies (e.g. chaining, probing, multidimensional hashing) can be used to minimize the cost of retrieving values from the array in the presence of collisions.



A common way to handle collisions is called chaining: all the values associated to different keys for which we have collision are put together in an ordered container (bucket = container for all elements having the same hash). Therefore, if my hash function returns the same values for different keys, as we can see for the keys a and b in the example, I'm going to place the values associated to those keys in a list.

So, we start from a set of key:value pairs. I apply the hash function, which is going to tell me where to place the values. Anytime I have to place the values somewhere in this empty array, in which each unit is going to contain a empty container/list, each value is going to be appended into the corresponding cell. Once the dictionary has been created, each cell of this array contains a list (sometimes with just one value, sometimes with more).

Now let's fetch in the dictionary the values associated to a given key. As we already know, the key is processed by the hash function, which returns the index position in the array where the corresponding value is. Then, in that index position I'm going to find a container which can contain one single value or more (if contains more values, I have to scan the list to fetch the right value – I have to iterate in the list!)

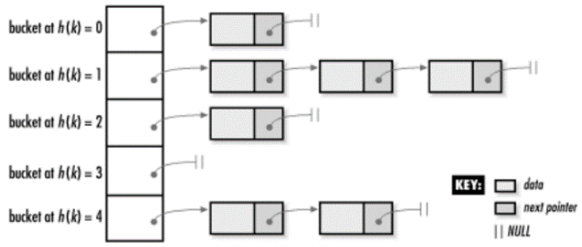
Complexity it is not constant anymore; it depends on the number of collisions. For each collision indeed, we have a container with multiple values through which I must iterate. Therefore, hash tables have an average time complexity of O(1) in the best-case scenario (each key is assigned a different index), while the worst-case time complexity is O(n). The worst-case scenario occurs when all keys generate the same hash index, and we have to resolve the collisions for each key.

Time required to do this process depends also on how efficient these operations are:

- Insert(x) – append x to the chain at index h(x) (initially, the chain is empty).
- Search(x) – go over the chain at index h(x) and look for x there. If the search failed, x is not stored in the table.
- Delete(x) – search for x. If it is found, remove it. If not, do nothing.

Python's lists are fine but some operations are not efficient. Append is always done at the end of the list, so this is reasonably fast. But if I want to put a value at the beginning of a list, it usually means shifting all the other values in order to create an empty space at the beginning in which I can place my value, so this process is very slow.

For this reason, hash tables usually don't use "normal" lists, but a more complex data structure. Often in fact these containers are linked lists, that are a series of values and pointers toward the next item in the containers (linked lists offer some advantages in terms of efficiency compared to standard lists, especially with long lists).



A linked list is a sort of list, formed by sequences of nodes, each one storing a value and pointing to the next node in the list. A node is an element of a linked list, and they are tiny container containing Data and Next:

- Data: contains the object to be stored in the node.
- Next: contains a link (pointer) to the next node on the list.

The first node is called the head, and it's used as the starting point for any iteration through the list. The last node must have its next reference pointing to None to determine the end of the list.

The main advantage of linked lists comes when inserting or deleting elements. If I want to place a value in a standard list, so if, for example, I have to put a value in position 5, all the items from the 6^{th} on, basically have to be shifted, in order to create an empty space to place my value. In a linked list, if I want to place one item, I just have to change the pointer. I have to add a new node and I need to change the pointer to the new node and the one from the new node back to the rest of the chain.

Inserting elements at the end of a list by using the methods .append() or .insert() is done in constant time O(1), but when you try inserting an element closer to or at the beginning of the list, the average time complexity will grow along with the size of the list O(n), since all or most elements need to be shifted.

(Python doesn't have a data type for linked lists, they are just lists.)

Linked lists, on the other hand, are much more efficient when it comes to insertion and deletion of elements at the beginning or end of a list, where their time complexity is always constant: O(1), since only the links of the elements at the insertion or deletion site need to be changed.

Problem of collision is solved using chaining. We ignore the fact that two different keys point in the same position and we add all the values in that position in a chain. This chain can be a linked list, such that, each time you have a new key pointing to an already existing chain, we can insert a new value in some position in this linked list efficiently.

Searching in the linked list require the same time and still depends on the length of the chain. But deletion and insertion can be done efficiently.

One way to handle collision using Python is the open addressing. In this approach each cell in the table contains at most one element. When a new element tries to enter an already occupied cell, the new element will be moved to another cell, according to some predefined rule.

I start scanning through the array. Position 1 is already occupied, then I look at position 2, that is as well occupied ecc until I find the 4th place that is empty. Therefore, I will put my value associated to my key in that position, because is the first empty position that I see. This approach creates a lot of problems.

IO Lesson 7 16.03.2023

The Needleman & Wunsch and the Smith & Waterman, are algorithms with polynomial quadratic complexity. This means that they're reasonably fast, but when I have a sequence that has to be compared to million other sequences, they're too much slow.
The problem of finding among a large set of sequences which are more similar to the sequence of interest, has to be solved in a different way.

For this reason, some new algorithms had been developed and their peculiarity is that they're **based on assumptions** to speed up the research of homologues.
These are **Heuristic algorithms** which means that they align a query sequence with only a subset of the database sequences satisfying a selection criteria (assumptions), then select scores of alignments higher than a certain value.

The first two algorithms that we saw are instead **optimal algorithms** because they compare the query sequence with all the sequences in the database. This makes them more precise but much less efficient.

It is possible to get the optimal result using the heuristic solution only if the assumptions are good.
So, usually if you base the research on assumptions, it will give you the suboptimal solution which depends on the correctness of the assumption given.

Going back to FASTA:
Two sequence which are evolutionary related (have a common ancestor), still maintain some substring that are kept identical. Regardless the distance between the two sequences.

Instead of explicitly aligning the query sequence to all the ones present in the database, a preliminary step occurs in which only the sequences sharing at least some identical substrings with the query are moved forward to the next procedures.
This leads to the discard of most of the database sequences so the alignment with polynomial quadratic complexity is done on less sequences, decreasing the total complexity of the algorithm.

The gain in terms of how much time I'm going to dee compared to the time required to align each sequence of the database against my query sequence depends on how efficient this step is, so finding whether two sequences share a substring or not.
The substrings must be identical, in order to find them there are many algorithms that can be applied but they are mostly not efficient enough.
The solution adopted by FASTA is to convert all the sequences in the database into data structures called Hash tables.

Hash Tables

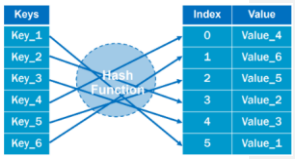
Hash tables are like a Python dictionary (which is a sort of hash table), there are keys and values. The keys are the substring of a given sequence in the database, the value is where that key is found (position).

All the sequences in the database are pre-processed: divided into substring of length k and then stored in hash tables.

The advantage is that if the sequence is already decomposed in hash tables, the retrieval of the key happens in constant time because it does not depend on the size of the hash table or the database itself but just how fast is the memory access (pretty fast).

A hash table is implemented using a **hashing function**. It can take as argument a large and different set of datatype that can be str, numbers, tuples, or any for of immutable data type, and give back an integer number.

Given an ordered container (has to be indexed), a key (ex. Key_1), a value (ex. Val_1), the hash function takes the key and returns an integer number (ex. 5) which correspond to the index position in which the value will be stored.

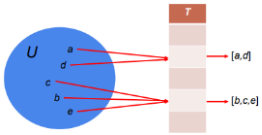


If I want to search in this data structure, I must need a key in order to look up for the corresponding value.

Since the only thing that I'm doing is applying a function to a key, the time to do that depends only on how fast the function is in returning the index position for the key, then the access is direct.

Three kinds of operations can be done on the hash table:

- Insert
- Search
- Delete

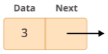


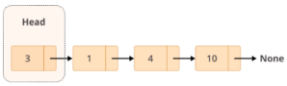
Collisions can happen. It is when the hash function returns the same index for a different key.

To handle collisions without increasing too much the complexity of the algorithm, **chaining** is used.

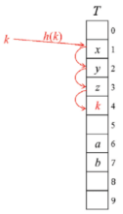
In chaining, a container used for storing all values associated to keys having the same hash are **linked lists**.

A linked list is a sort of list, formed by sequences of nodes, each one storing a value and pointing to the next node in the list. The first node is called head and it is used as the starting point for any iteration through the list. The last node must have its next reference pointing to None to determine the end of the list.





Linked lists are special kind of arrays, they are formed by nodes formed by two halves: one that store the value and one that points to the next node.
Its advantage is that it is easier to **insert nodes** at the beginning and in the middle of the linked list. It is because it's just necessary to change the pointer of the node, and this operation's cost do not depend on the length of the array.

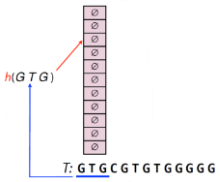


In Python dictionaries and sets collisions are instead handled by using a different approach, called **open addressing**.
If the index position is already occupied the new position will be the first free one.
The problem now is to retrieve the position because it is not directly linked to the hash function anymore.

How a hash table can be used for k-mers of a biological sequence:

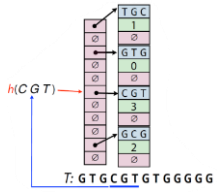
- We need to decide the value of k for generating all k-mers in the sequence, and we want to store the offset for each one of them.
- The key:value pairs that we have are therefore of the kind k-mer: **offset.**
- Then we need to use a hash function on each k-mer for obtaining its index.

- Then we need an array of chains (buckets), and we need to place the value (offset) associated to a key (k-mer) in the right bucket.



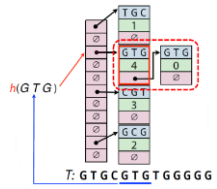
For example, let's take in consideration the string GTGCGTGTGGGG.
Let's say that k is set as 3. The first 3- mer of the sequence T is GTG; we apply the hash function h to GTG, we get an index (2 in this example), and we place the offset of GTG (which is 0) in the bucket corresponding to the index.

For handling collisions, the bucket is created as a list, or a linked list for which this is the first node (the head), and we place in the bucket corresponding to the index not only the offset of the k-mer but also the k-mer it corresponds to (GTG), for example in the form of a tuple (k-mer, offset).



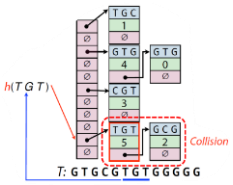
We then move to the second k-mer (TGC), apply the hash function to it, and place its offset in the array at the resulting index position (0 in this example).

And we repeat the process, moving over all possible 3-mers of T.



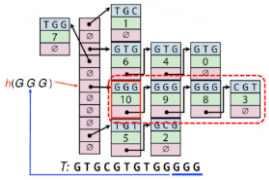
When a k-mer that was already placed in the array is encountered, it is necessarily placed in the same bucket of the previous one, since its hash is identical. A new unit is added to the container (e.g. to a list or linked list) storing the values at that bucket.

Dictionaries in python must have only one key, not duplicated ones. So, when we work on hashing sequences, every item in the array must be a chain.
Being a chain is helpful to not have collision and for handling multiple instances of the same substring in the same string (which happens a lot especially for nucleotide sequences).



There might be cases of collisions, when a k-mer has the same hash of a different one. In this example, the hash of TGT is the same of that of GCG, hence they are placed in the same bucket.

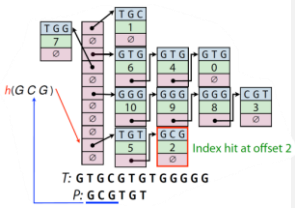
We then go forward with the process placing GTG again, TGG, and 3 GGG which create another collision.



We are finally done with hashing the sequence to the hash table.
This process can be repeated for any sequence present in the database that can be condensed in only one hash table or one for each sequence.

If we want to hash all the sequences in one hash table we have also to add the name of the sequence to which the k-mer refers to in order to retrieve which sequence is which.

Now we consider our query sequence P (GCGTGT), I want to know if P and T share some identical substrings.
I need to divide P in substrings of the same length I used to construct the hash table (3).
Then starting from the first k-mer (GCG) from P, apply the same hash function that will lead me to the same k-mer of the previous sequence.
Basically, I scan through the linked list checking whether the substring is the same as the one we are looking for, finding in this case GCG with the offset 2.



Then I move to the next substring of P (CGT), apply the hash function, scan through the linked list, and retrieve the offset number.
The process is repeated for every k-mer in P with the objective to find how many k-mers the two sequences have in common.

If we exclude the additional complexity of iterating through the linked list, the access time is constant.

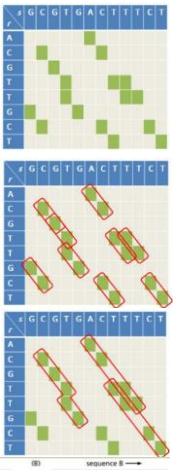
In this subset of the database sequences' collection, there might be not only true homologues but at least it allowed to discard the vast majority of the ones which are not related at all.

In this subset of the database, there are some additional steps for deciding whether or not a sequence has to go through the alignment. The user can also impose that at least m k-mers of q must be found in a sequence i from the database in order to let sequence i proceed to the next step of the algorithm.

Further steps are required to lower the amount of non-homologues sequences left in the subset.

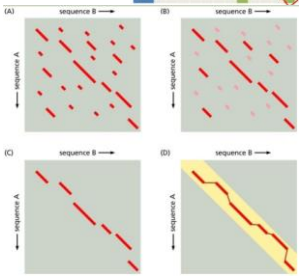
In order to justify the presence of the sequences in the subset, similarity can be checked in another way:

The two sequences, decomposed in substrings and stored in hash tables, are then analysed by for each substring to check how many are identical.
In this example, the matrix has green squares in the position in which the two 2-mers are identical.



From this matrix we're able to understand whether it is worth it to align them or not, but there are some additional checks that have to be performed:

- merge consecutive matches
- extend the matches — extend the matches that are not exactly consecutive. This means allowing mismatches.
- Allow gaps



Basically:

- Find region of perfect identity in the hash tables.
- Join the consecutive ones. (A)
- Consider the longer ones. (B)
- Look up for segments on the same diagonal. (C)
- Join them by a small number of gaps. (D)

It is important to avoid computing all the cells in the matrix by excluding the ones which do not have an alignment. This allows us to focus on the part of the matrix including all the segments that I actually need to compute (yellow band).

Which is the score of each one based on how many mismatches and how easy is to connect them with gaps. Only at the end an alignment is performed but only in the band in the matrix including all the segments that I selected up to this point that should be part of the optimal alignment. The segments can be chosen because each one has to be scored based on the values on the substitution matrix.

In FASTA and BLAST the identical substrings are called **words**. The word size is called k-tup.
For protein sequences, the default word length is k-tup=2, for nucleic acid the default is k-tup=6.

k-tup determines the balance between execution speed and search sensitivity.
Raising the k-tup leads to:

- Lower background noise;
- Shorter running time;
- Higher risk of missing distant homologs.

Therefore, if the substring length is long, it is less likely that you're going to carry forward with the procedure, sequences that have some identity with the query sequence just by chance, but the risk is that you're going to discard homologues of your query just having higher divergence. This could be a problem or not depending on how many you lose.

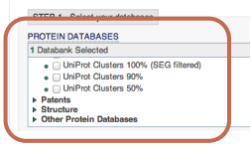
If the substring length is too short, it is more likely to retrieve more homologues but also many false positives (sequences having random identity with your query sequences) which means more noise.

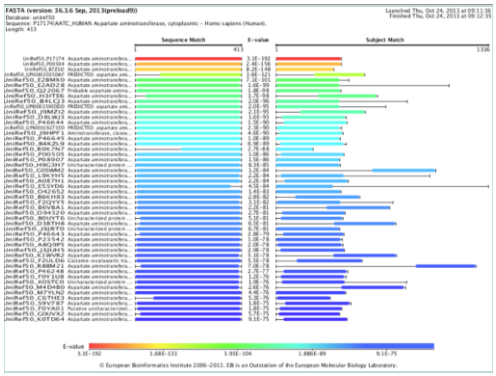
Parameters that need to be defined:

- The **word length** (= word, or k-tup) for the initial matching
- The **cut-off** for the partial alignment scores
- The **number of partial alignments** that are carried on
- The **band width** for the final alignment

FASTA is not as used now because BLAST is better in most cases, specifically for proteins.
The advantage of FASTA is that BLAST doesn't actually compute the true alignment. But this cost having higher running time.

FASTA and BLAST can be used both with online interfaces which database is limited to the one given by the developer, but if you download it, it's possible to use your own database.
Some database used are the standard and the most complete ones, for example UniProt (protein sequences).
Redundancy happens because all biological databases might contain different entries which are basically the same thing, similar or identical because researcher around the world can add their own sequences.
Redundancy can be selected from those datasets to choose the ones that fit properly for your investigation. These databases are prepared by clustering in order to differentiate the sequence that are similar to each other's and put them in the same cluster so then only one of them is picked and not all of them.
100% clusters means that the group contain only sequences which are identical.





Each line corresponds to one alignment.
Different colour correspond to different E-value, which is a statistical evaluation of the significance of the match: how likely is to get an alignment with the same score from two unrelated sequences. (red is high E-val)

The smaller is the E-value the more significant is the alignment.

There is a clear limit to FASTA, which is that it requires perfect identity between substrings from two sequences. Basing the algorithm to the hypothesis that homologues no matter how far they are, they must have maintained some identical substrings. But there might be cases in which is there are not identical substrings, even tho the sequence looks similar.

In the example to the right, this could happen if we take k = 4.

The sequences
ALERDESTGGVLIM
ALCRDECTGGCLIM

do not share any identical substrings of length at least 4, therefore, if 4 is the chosen word size, even though their sequence identity is 79% overall and they are likely homologs, FASTA would not find this match

FASTA requires identity, BLAST require similarity!

Problem with FASTA that do not happen in BLAST:

EMI Lesson 8 17.03.2023

Fasta is kinda old, slower than blast but it's still valid and used for large scale sequencing. It has a limit: since the preliminary step is the search of the perfect match in substrings could happen that two strings are really similar but they do not share any subsequence of k length, so they are discarded even though these sequences are true homologous.

BLAST

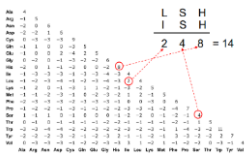
In order to avoid this problem Blast was introduced. The idea is to move from the perfect identity to just similarity. This search must be efficient and carried out in reasonable time. Hash tables are perfect for this kind of situation since you can perform a search in all the DB in almost constant time (ignore collision).
The idea behind blast is to still use hash tables but, instead of just searching for all the subsequences in the query sequence, it first generates variants (simulates how the query sequence may have evolved) of the query sequence and then perform the search in the hash table.

In steps:

1. I start from my query sequence and I select the length W in which I want to decompose my query sequence (word or W-mers)
2. From each word I generate some variants changing characters obtaining a list of neighbors(Neighbors: collection of variants of each substring of length W in the query sequence).

To understand which substitutions are more probable to occur I use a substitution matrix. Substitution matrix contains substitution rates computed over the alignment just by counting the differences that we see and then converted into some numerical score.

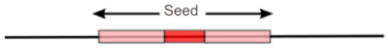
3. To each similar/homologous sequence that I find in the DB I assign a score based on the substitution matrix
4. I select a threshold and accept all the sequences that have a score greater than that.



Computing similarity during the search space it's time consuming.
The great idea behind Blast: generate variants that contain small differences and then use this variants for fetch things

in a hash table in constant time. The neighbors at the end are reasonable variants of the sequence that I could find. Each query sequence is seen as a collection of variants.

Blast then extends the initial match (seed). Blast looks the residues before and after it, evaluates each pair of residues

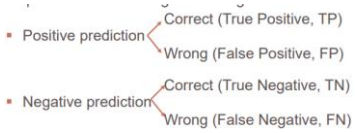


and if they have a good conservation score they are added to the sequence(seed extension); this extension continues until you find a negative substitution score.
Once extension stops, a score for this extended alignment is calculated (by summing all the substitution score of the residues that have been aligned) and, if this is above some threshold we previously fixed, we obtain the HSP(high-scoring Segment Pair).

Limit: there cannot be gaps in the sequences. Anyway often a local alignment is enough to find some homology with a well annotated gene, so blast it's still very powerful.
In the recent version of blast (Blast 2) we can perform alignment with local gaps.

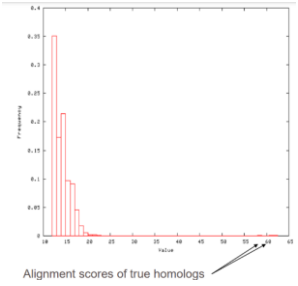
All the matches can be evaluated statistically to compute the probability of finding similarities by chance only (False Positive).

When we are making a prediction, it can be positive or negative. Positive when I classify the sequences as homologous, negative is the opposite. Obviously, I can be right or wrong, a True Positive (TP) is when I'm right, False Positive (FP) when I'm wrong; same for the negative prediction (TN and FN).



Having several FN might not be that harmful, since we just need a few of them to identify the origin and function of the query. Instead, if we have several FP (unrelated sequences that we accept as homologs to the query), the output of the analysis will be much more difficult to interpret, since our query might seem related to sequences with different origin and functions.
There are 3 important parameters that govern BLAST: the word size W, the word similarity threshold T and the minimum HSP substitution cut-off score X.
W: The choose of the W-mers is really important because if W is too long, then I speed up the procedure avoiding many FP at the cost of having FN (true homologous of the query sequence that I discard).
If W is too small the search is slower, I might have more FP but I could catch all the TP of the query.
T: If T is higher, the algorithm will be faster, but you may miss sequences that are more evolutionarily distant. If comparing two related species, you can probably set a higher T since you expect to find more matches between sequences that are quite similar.
X: Its influence is quite similar to T in that both will control the sensitivity of the algorithm.

In the end: if in the output I don't see any match or just few I need to relax these parameters.



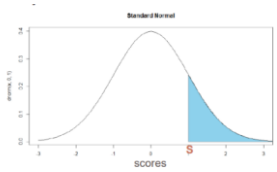
This is a typical distribution of the BLAST output alignment scores, in which we can easily assume that the true homologs are those at the far end of the tail, while the main peak contains mostly false positives.
Nevertheless, we would like to estimate if this is true and how much likely is to get a high alignment score by chance only.
We would like to reduce the number of errors that we might make, especially false positive errors; in order to do that, we need to evaluate the statistical significance of an alignment, which means testing which is the likelihood of having an alignment with a given score between unrelated sequences, whose similarity occurred by chance.

Statistical significance of an alignment

In hypothesis testing we test the probability that a hypothesis is true.
Usually, the null hypothesis is that two sequences with high similarity align just for chance → we want to find FP.
We need to obtain a test statistic and its distribution.
p value: significance level (the smaller, the better), it's usually scheduled on the size of the DB

Shuffle the characters in each sequence, keeping the percentage of nucleotides, and you create a distribution of scores. Define a threshold of significance to compute the probability that the sequence is or is not a false positive.

Which is the score distribution? How can I compute the p value?
We might assume that these scores would follow a normal distribution, symmetric around the mean

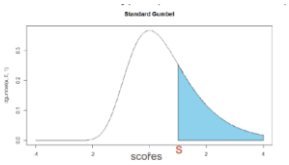


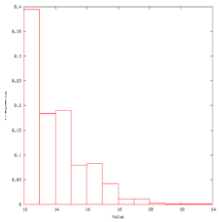
If we could know the parameters of this distribution, we could use them to compute the p-value of an alignment A with score S as the probability P(x ≥ S) of having an alignment with score equal or higher than S by chance. The blue part represents the probability to find a real alignment

But this is not what BLAST does, since in the BLAST output we don't get the score of all possible alignments, but only those surviving all the different thresholds and cut-offs.

In the end we will get the best alignments and the score distribution will be something like this:

The distribution of the maximum scores, i.e. the scores of the HSPs in the BLAST output given a set of parameters, is an extreme value distribution of type I (Gumbel distribution).





This is the distribution of BLAST scores that one can obtain by comparing a sequence against a database of proteins in which the sequences were randomly shuffled (change the order of the in each sequence)

If we can predict the distribution of the score over a random DB without really compute it, then I can use that value as real score.

Check how far is the score from the average score. More distant we are from the background noise, the better. We also need to take in account how big is the distribution, the variance. It's important to calculate the z-score to asses the significance of the score over a distribution.

Z-score = 0 → mean
Z-score > 0 → alignment score is higher than the mean
Z-score > 5 → considered significant

$$\text{Z-score} = \frac{\text{Score S} - \text{average score}}{\text{standard deviation}}$$

If I select a substitution matrix for conserved proteins I will have a higher, narrow distribution
If I use a matrix for divergent proteins this will be shorter

Two important values I can compute with mathematical operations: K and lambda. They can describe the noise distribution without actually compute the distribution.
What I got in the end is an approximation that actually works.

For an alignment *A* with score *S*, we would like to know the probability $P(x \geq S)$ of having an alignment with score equal or higher than *S* by chance (i.e. between the query and a target that has no relationship with the query)
This probability, that is the p-value, is a function of K and λ, and is formulated as:

$$P(x \geq S) \approx Ke^{-\lambda S}$$

The E-value is the number of alignments between the query and the database sequences having by chance score equal or higher than S (in a random DB of the same measure, it's a better estimator than p-value)

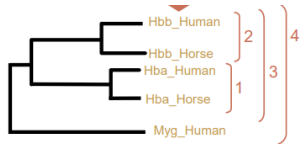
Multiple sequence alignment

We can scan multiple sequences and it can suggest that a residue present in both is conserved and it's important for something. Conservation in two sequences can be misleading: many evidence, found in different sequences, can provide more convincing evidences.
Many algorithms based on a multiple sequence alignment are able to predict many structural features of proteins in a quite accurate way. More conserved structures usually correspond to the core of the protein.
Multiple alignment is also important for understanding evolutionary relationship.
Complexity: n^k k is the number of seq aligned simultaneously

I start aligning the two more similar sequences, then on the top I will add the most close to each other sequences.

To do it I first need to align each pair, at the end I need to compute k(k-1)/2 pairwise alignments, if I start with k sequences. In the matrix I will put the score of each alignment. the matrix will have as dimension the number of sequences I want to align. Then I can look at the matrix and I select the greatest score. All the other steps are the addition of new sequences at this alignment.

		A	B	C	D	E
Hbb_Human	A	-				
Hbb_Horse	B	0.17	-			
Hba_Human	C	0.59	0.60	-		
Hba_Horse	D	0.59	0.59	0.13	-	
Myg_Human	E	0.77	0.77	0.75	0.75	-



I construct a sort of phylogenetic tree reporting the relationship between the sequences.

The overall complexity of this progressive algorithm is O(kn^2) since I compute first the k(k-1)/2 pairwise alignments, then I need to add the k-1 remaining alignments.
To keep the complexity polynomial quadratic each alignment (seq-seq, alignment- seq, alignment-alignment) in a bidimensional matrix (polynomial quadratic matrix). The implementation can be then relative fast.

Isaac Lesson 10 24.03.2023

One of the most important features of this kind of alignment is that it can show conservation patterns and if some sequences are under selective pressure.
Furthermore, it is possible to use multiple alignment as a tool for an artificial intelligence that allow us to find 3D structures of proteins.

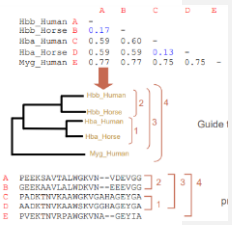
This concept of multiple sequence alignment has the problem that a new dimension of the matrix is needed for each new sequence we want to align at the same time. This makes the algorithm difficult to implement and the number of the sequences increases exponentially the calculations that have to be done.
In addition, for each number of sequences in input for the multiple sequence alignment, a different algorithm has to be implemented. Since the number of neighbour is different, the way the scores are computed is different, etc.

We need an algorithm that can be run for any number of input sequences without having to change the algorithm itself, and that has low complexity.

Progressive algorithm recap

It is an algorithm which builds the alignment progressively by adding each sequence one at the time.

The idea is to start building the first pairwise alignment of the two most similar input sequences. So, first is to check all possible alignment of the n input sequences.



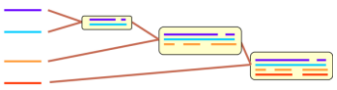
Add new sequences on this initial alignment following an order depending on the similarity of all the remaining sequences. It is made by building a tree structure similar to a phylogenetic tree (Guide tree), which guides the algorithm specifying the order of addition of sequences to this progressively built alignment.

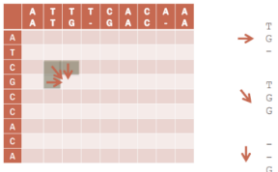
This implies that each alignment that needs to be compute is going to be always pairwise aligned, regardless of whether it is the alignment two sequences one against the other or if it's the alignment an alignment computed in a previous step to another sequence, or if aligning two alignment one against the other. This leads to the creation of a matrix every time there is an alignment.

Such an algorithm needs to compute first k(k-1)/2 pairwise alignments, each one having complexity $O(n^2)$, and then compute k-1 alignments between clusters, each alignment having again complexity $O(n^2)$.

The total number of alignments to compute is then k(k-1)/2 + (k-1). If we approximate the number of pairwise alignments to be performed to k, then the overall complexity is $O(kn^2)$. Which means that the algorithm can be considered relatively fast.

How do we align an alignment with a sequence?





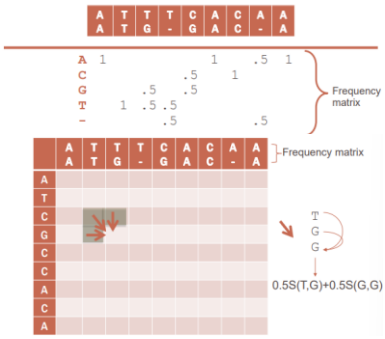
Since we know how to align two sequences, we would like to solve this problem in a similar way. This implies that all the previous relations that we learnt are still valid (each cell has three neighbours, there are three directions to reach a cell, etc.).

Instead of having one sequence for each of the matrix dimensions, we report on one dimension the columns of an alignment.

The question now is how to score each of the three possible movements.

The solution is to convert the alignment which is already computed in the previous step of the procedure in a frequency. A **frequency matrix** is a matrix in which the frequency of the characters that can be found in each column of the previously computed alignment is reported.

We can convert the alignment into a frequency matrix having a number of columns equal to the alignment length, and a number of rows equal to the number of characters in the alphabet plus the gap character.
In each cell of this matrix, the relative frequency of each residue in each alignment column is reported.



Use this relative frequency matrix as weight for computing the score of every movement in the alignment matrix.

The scoring function doesn't need to be different from the Needleman Wunsch. Just the weight is different.

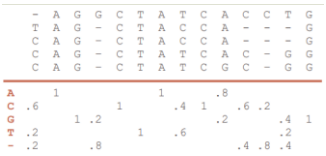
Consider each character at the time for each alignment of the sequence. So, in this case you compare T with G and G with G and sum them.

The values that have to be sum are retrieved from the scoring function, which consider the relative frequency of T, multiply it by the score of the alignment T-G; then summed with the relative frequency of G multiplied by the score alignment of G-G.

This method does not give the ability of integrating a new nucleotide which is not already present in the first two sequence alignment, because the frequency of the nucleotide would be 0 and when computing the score, it will not be considered, and passed.

In order not to lose this information it is possible to use pseudo-counts that are small numbers which is possible to assign to each possible character when computing its relative frequency in order not to have any character scored as 0, but something close like 0.01.

After this process, there will be the usual following steps as in a Needleman Wunsch algorithm, the last value of the matrix is the score of the alignment and then the traceback occurs. The only thing that changes is that there's a system of weights computed on the progressively built alignment which is used to weight the single pair of different cases.



This algorithm maintains the complexity low and extend the case of Needleman Wunsch for multiple alignment.

With the growth of the progressively built alignment, the frequency matrix needs to be recomputed after each addition of a new sequence, but regardless on the number of sequencies already included, the strategy remains the same.

Progressive multiple alignment algorithms are many, one of the most popular tool is Clustal which has many different versions. Right now, ClustalW and Clustal Omega are the most used ones.

ClustalW

Invented by Toby Gibson and Des Higgins, works with a progressive alignment strategy similar to that we just described, with adjustments.
The W in ClustalW stands for weighted, meaning that the algorithm uses a system of weights for different parts of the alignment, "**freezing**" some parts (usually corresponding to large gaps) and modifying more freely others.

If it is possible to detect a conserved sequence after the first few sequences aligned, it is possible to "freeze" it in order to impede substitutions or gaps, because that part of the sequence correspond to a region aligning under high selective pressure. In this way the algorithm is forced to focus the substitutions and gaps in the less conserved regions in order to have a more coherent alignment.

Gap penalties can moreover be adjusted based on specific amino acid residues, regions of hydrophobicity, proximity to other gaps, or secondary structure. This is because differences of amino acids in chore or in outside regions has different weight for example.

The complexity of the algorithm is polynomial quadratic time some constant factors, so it is relatively fast and seems accurate.

ClustalX is the version that can be installed on all the major operating systems. It has a graphic interface with histograms, etc. not important.

Clustal Omega

It's a bit different because it is based on hidden Markov models which are statistical descriptive sequences (we'll see them in next lessons).
It can be considered a progressive multiple alignment algorithm since the alignment is built step by step adding new sequences from a starting point.

Clustal Omega can change substitution matrices in different part of the alignment while running. Once aligning the first two sequences, that alignment cannot be changed anymore. So, it is fundamental that this first alignment is good, otherwise the next alignments will be meaningless. The addition of new sequences to the first alignment is done by weighting in a specific way in order to align the new sequence characters based on the context in which the character is found.
It is not enough to overcome the problem of the initial alignment but at least it should minimize the propagation of errors from that point on.

Iterative algorithm

It's another general class of multiple sequence aligners.
They work similarly to progressive alignment algorithms, but they can realign the first two sequences and at every point in the process of multiple alignment.
The realignment is very complexity expensive, because they re-evaluate the alignment that had been done up to that point at specific moments.

They perform better at the level of biological meaning at the cost of higher complexity.

- T-Coffee seems biologically more meaningful;
- MUSCLE which is an iterative multiple sequence aligner, it is quite fast;
- ProbCons is based on the Needleman Wunsch chains, it seems to be the most accurate for the biological meaning but it's very slow.

ClustalW: Progressive; most widely used
http://www.ebi.ac.uk/clustalw

T-Coffee: Progressive; better but slower
http://www.ch.embnet.org/software/TCoffee.html

MUSCLE: Iterative; can align very large number of sequences
http://phylogenomics.berkeley.edu/cgi-bin/muscle

ProbCons: Hidden Markov chain; most accurate
http://probcons.stanford.edu

One of the reasons why I might want to do a multiple alignment is for **finding conservation patterns**: where and how much different residues are conserved, indicating a differential selective pressure.

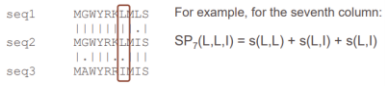
In order to get this information, we need to look at these evaluations:

- Number of matches conserved residues in the alignment columns.
  It is related to the frequency, the symbols are:
  - ***** conserved.
  - **:** not perfectly conserved but composed of residues with similar characteristics in terms of hydropathy and size.
  - **.** not perfectly conserved but composed by residues with similar characteristics in terms of hydropathy or size.

- Sum of pairs score (**SP-Score**)

This score is a way for computing the conservation score of each column of the multiple alignment and an overall score of the total alignment. It allows us to have a number describing how well all the sequences were aligned up to that point.
Once computing the multiple alignment, all pairwise alignment are extracted.

Follow the example:
Given the alignment of these three sequences, to compute the SP-score we need to consider the three pairs of sequences and check the score of the alignments.



Considering the fourth character from the right, its SP-Score is the sum of the score of the characters in that position in the three cases.
i.e. the score of the couples: L-L, L-I, L-I.
These scores are retrieved from a standard substitution matrix.
Taking scores from a substitution matrix means that some residues will have a lower score independently form the level of conservation in the multiple sequence alignment, for example in the case of alanine which has low scoring because it is very versatile.

- Entropy of the alignment

It refers to the entropy concept of information theory, so it refers to how clear is a message.

Px is the frequency of a character x in a column of a multiple alignment, which is multiplied by the logarithm of the frequency of Px changing its sign to get a positive value.

With base-2 logarithm, the entropy ranges from 0 (perfect conservation, minimum entropy) and 2 for nucleic acid, 4.3 for proteins (low conservation, maximum entropy).

The total entropy of the alignment is the sum of the entropy of each alignment column:

$$\Sigma_{\text{Over all columns}} \Sigma_{X=A,T,G,C} p_X \log p_X$$

A	A	A
A	C	C
A	C	G
A	C	T

- Column 1 = -[1*log(1) + 0*log0 + 0*log0 +0*log0] = 0
- Column 2 = -[(1/4)*log(1/4) + (3/4)*log(3/4) + 0*log0 + 0*log0] = -[(1/4)*(-2) + (3/4)*(-.415)] = 0.811
- Column 3 = -[(1/4)*log(1/4)+(1/4)*log(1/4)+(1/4)*log(1/4) +(1/4)*log(1/4)] = 4* -[(1/4)*(-2)] = 2.0
- Total Entropy = 0 + 0.811 + 2.0 = 2.811

Sometimes the average entropy is used, and it is calculated like the total entropy over the number of columns.

Computational Genomics

Sequences coming from sequences platforms.

We're still working with strings, but there are differences.

- Goals it is still string comparison but done with different methods.
- Scale sequencing data are much many, so the scale is very different. It is therefore important to handle these data.

Even if we still work with string, we will need specific data structures and algorithms.

Depending on the biological problem and the type of sequencing that is performed to help solve it, we can classify sequencing problems from a computational point of view in two classes:

- **Assembly**
  How to combine all these short sequencing reads in one whole genome or transcript. To assemble sequencing reads we need to compare them to check whether one is similar to the other in some regions.
- **Mapping**
  Determination of locus of origin of a sequence in the genome used as reference. It is similar to an alignment problem, but it has to be much faster. It is a problem which sometimes is solved by hashing the reference genome.

Once a genome is reconstructed, many methods are available for the characterization of its sequence, including:

- **Genome annotation**
  the association of a role and/or function to ideally each gene in the sequence.
- **Comparative genomics**
  the comparison of genome sequences from different species or different members of a species.

We will not go into details in every comparative genomics sectors, but we'll underline some features of the data that we could work with, because the algorithms are specific for some conditions.

Errors

FASTQ file is the standard format to report sequencing data, it has the sequence and an estimation of correctness for each one of them.

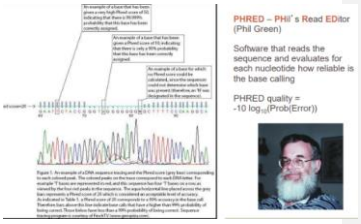
First protocols for sequence analysis of nucleic acids:

- Ray Wu (1970) quite similar idea to the one later implemented by Sanger (only one not getting Nobel).
- Frederick Sanger (1975) is sequencing by synthesis (very similar to Ray Wu's method).

- Maxam & Gilbert (1977) is sequencing by degradation.

In Sanger sequencing dideoxy nucleotides are used to stop extension. It's important to remember to understand where the errors could happen.

- go look slides 58 – 61 (pdf 10) if you don't remember the process -



Nowadays it is possible to incorporate all the ddNTPs in the same tube because they fluorescent with different wavelength (colour).

The machine that is used has a thin acryl capillary (instead of gel) for the fragment separation.
While the fragments generated by the replication reactions are separated by migration in the capillary, a laser scanner induce fluorescence for each fragment and, by reading the emission frequency, determines the sequence of the DNA template. This sequence is called read. The output is an electropherogram.

To calculate the quality of each read is used the PHRED score. The PHRED score is calculated by a software that reads the sequence and evaluates for each nucleotide how reliable is the base calling.
PHRED quality = $-10\ log_{10}(Prob(Error))$

Every software that deals with data having a quality score has to be aware of how to treat them.

Emi Lesson 11 30.03.2023

Computational Genomics

We can classify sequencing problems from a computational point of view into two classes:

- assembly: reconstruction of a whole sequence from a collection of overlapping fragments
- mapping: determination of the locus of origin of a sequence in the genome, used as a reference

When we have assembled the genome sequence we need to extract information. There are:

- genome annotation: the association of a role and/or function to ideally each nucleotide in the sequence
- comparative genome: the comparison of genome sequences from different species or different members of a species

Sequencing data are big and usually contains many errors. Some algorithms to work with data are perfect on ideal data (no errors) but then there are a lot of errors when working on real data.

Sequencing started in the '70s, the Sanger method was developed in 1975. This is a sequencing by synthesis, it uses one strand to synthesize a complementary strand and recognize the modified nucleotides added. We will get many strings of different length differing just for one deoxynuclotide. They needed to prepare two reactions in four different tubes, then they started to use radiolabelled nucleotides → fluorescence.
Since it's an automatic software we should be able to recognize errors and deal with that.
The data from this software come out as a electropherogram. The software it's able to tell us which nucleotides we can trust more, which less. The probability of having a wrong nucleotide was converted into an integer number (Phred score).

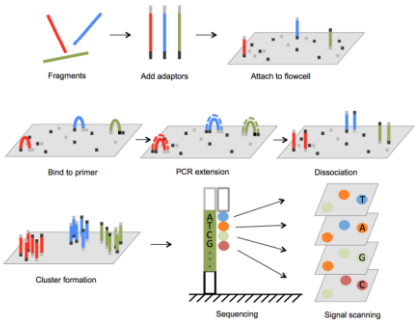
The FASTQ format

```
@SEQUENCE1
GCCCGGCGGGTTCATGCTGAAGAAAGGCGAAGTGTTCGGTTGGGCGGC
+
ffffffefe^eeceedffdcd^dXecffbeed`Reebe`db\]XWSS
```

A FASTQ file is divided into blocks of four rows for each read:
The first row starts with a @ character followed by a unique identifier of the read;
The second row reports the read sequence
The third row is a spacer, starting with a + and sometimes repeating the read identifier
The fourth row reports the quality score for each nucleotide, encoded as character conversion from the ASCII code (Q score)

NEXT GENERATION SEQUENCING METHOD

Next generation sequence: Illumina is the most used



Illumina produces huge data
everything happens on a flow cells divided in 8 lanes.
There are 2 types of adapters needed for amplification of the fragments. What we get are high resolution images in which there are dots, each colour depends on the wavelength, that correspond to the particular nucleotide.
Each spot is multiple copies of the same fragment (cluster).
A DNA polymerase is added to the mixture, along with four types of modified nucleotides, with the following characteristics:

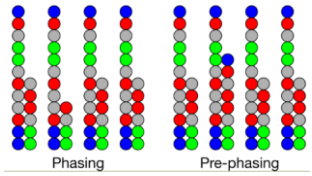
- Light emission at different wavelengths when excited by a laser
- A reversible replication block, which allows each nucleotide added for the elongation of the complementary strand to be read by its emission wavelength.

Both the fluorophore and the replication extension block are removed at the end of each cycle, allowing the next nucleotide to be added and read.
In each sequencing cycle:
1. A mixture of labelled nucleotides with reversible terminators is added
2. The complementary nucleotide is added to extend the growing strand
3. Fluorescence is acquired, and the terminator is removed
Steps 1-3 are repeated for a fixed number of iterations

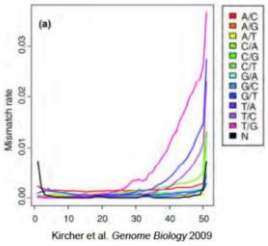
For assembly it's important that fragmentation is random in a way in which reads overlap.

In sanger there is a block at the 3' position that make impossible extension after the nucleotide modified has been added.
In illumina the block at the 3' position can be cut and the extension can happen only after the fluorescence is read and cut away.

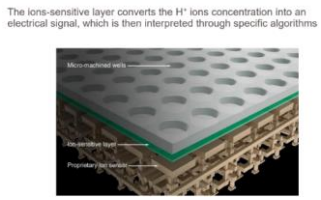
Accuracy of illumina is good, the most frequent error is a substitution with a rate of 0.1 %. Some errors are more common, no one know why.
Many people tried to correct illumina reads but it didn't work well.



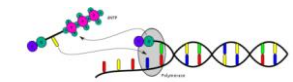
Errors depends on the clusters themselves; another source of error is phasing.
This events are rare but they actually happen and accumulate over each cycle.



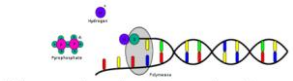
Ion torrent platform



They're faster than illumina sequencing.
We start from ssDna and fragment it. Then the Dna is amplified and it follows the synthesis of the complementary strand. The formation of the phosphodiester bond releases 2 phosphates and a hydrogen ion. The ion torrent platforms detect the hydrogen ion release thanks to a change in ph.



Polymerase integrates a nucleotide.

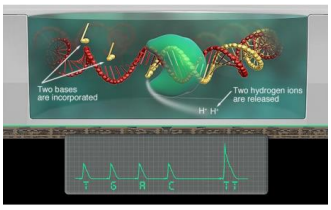


Hydrogen and pyrophosphate are released.

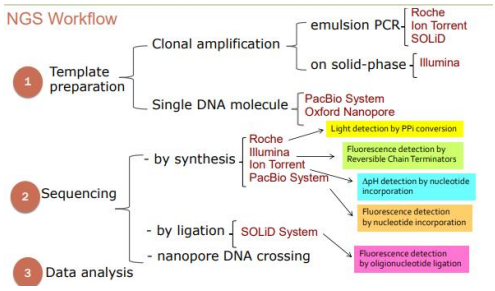
We divide each cycle into 4 subcycles (each one corresponding to a nucleotide) in which we add a nucleotide per time and then wash it away. When the nucleotide is added and the change of ph is detected, it means that in the strand I'm sequencing there is the complementary nucleotides to the one added. In this way we can use standard nucleotide (no fluorescence).

We don't need to block the elongation. If there are consecutively nucleotides that are equal, these are added all together and the change of ph is proportional to the nucleotides added.
The problems arise when we have long homopolymeric stretches. The most common errors are deletion and sometimes insertions.

The error rate is estimated around 1.7%, for homopolymeric stretches the error rate can reach 60%.



Some sequencing platforms of the so called 'third generation' employ protocols that do not require amplification, and that can produce reads having length in the order of tens of kilobases. Some examples are PacBio and Oxford Nanopore technologies.
PacBio can perform single molecule real-time (SMRT) sequencing. This means that the sequencing is faster and does not need amplification. You don't need to block the synthesis in cycle but just follow the polymerase. In PacBio fluorescence is still used but signals are weak because there is not amplification. Error rate is quite high, estimated at around 19%
Oxford nanopore technologies are different. They follow passage of ssDna molecules through ports, which can change the passage of current through the same ports depending on which nucleotides is passing. The passage of the nucleotides through pre is really fast and lead in difficulties in detecting the electric signal. For this reason sequencing error rate is high, being 20% on average. Error correction algorithms can be applied to reduce the error rates, lowering it at around 12%



Sofia Lesson 12 31.03.2023

Given a genome or transcriptome fragment, the sequence can be determined:

- from one end of the fragment (single end sequencing)
- from both ends of the fragments (paired-end sequencing)

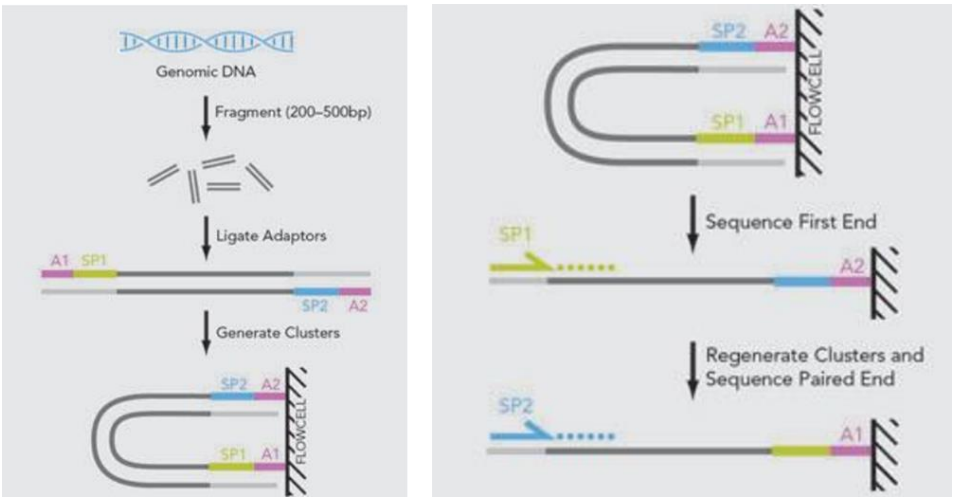
The paired reads that are generated through these protocols usually are still not long enough to cover the entire fragment sequence, but they are still useful to connect sequences at relatively long distance. For each fragment you obtain a pair of reads, separated by an unknown sequence, but for which we know the size.

The central part of a DNA fragment that is not sequenced is called insert. The insert size is equal to the fragment length, minus the sum of the paired reads (example: if a fragment has length 400bp, and from both ends a 100bp read is read, the insert size is 400-2*100 = 200bp).

With Illumina, each cluster after each amplification retrieves a mixed population of strands, half of them have a sequence which is identical to the original genome fragment, while the other half is the complementary sequence. Before starting the sequencing, you have to remove one of this two types of strands by cutting the adapter of the fragments we want to remove, so that a cluster contains strands all identical to each others (bunch of strands having identical sequence) (we obtain the sequence of one end).

Then you sequence them using single-end sequencing.

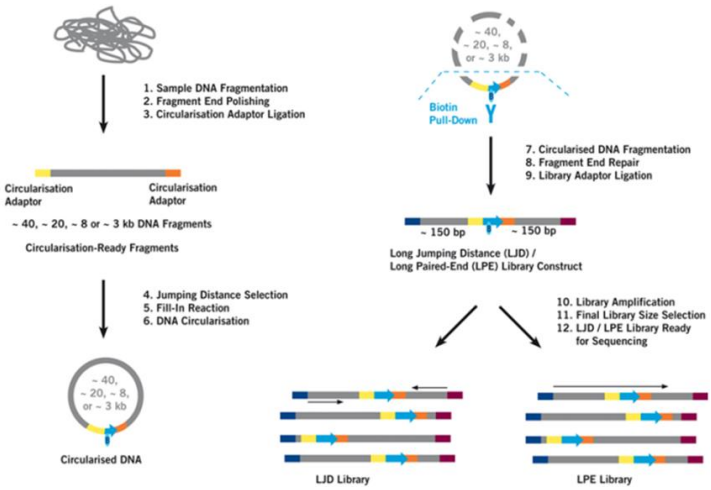
After that, you can regenerate the cluster and remove the other type of fragments, in order to obtain a cluster with the other kind of strands (obtain the sequence of the other end) by using the paired-end sequencing.



Paired-end sequencing provides long-range connectivity information that can help in read mapping and read assembly. Moreover, paired-end reads can help in determining which splicing variants (isoforms) are transcribed from a gene in the sample cells.

A limit of paired-end sequencing is that pair of reads that you get comes from fragments that are usually short, just few hundred nucleotides. In order to get longer range connections some protocols can be introduced.

Mate pairs sequencing (or jump libraries)



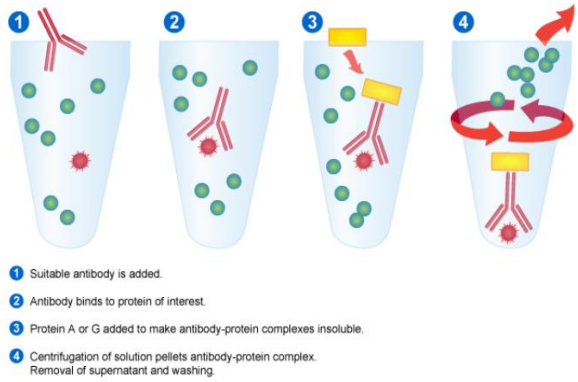
You first fragment DNA using sonication or enzymatic digestion, and then select long fragments (not fragments of size that machine sequencers like, but also 10Kbp long). Then these fragments are circularized joining the end of each linear fragment (ligation – covalent bond), and the end junction is labelled in some way, usually using biotin (which can strongly bind to a protein called avidin).

You will obtain a double-stranded DNA circular molecule in which the joining point is marked by biotin.

This circular molecule can be further fragmented (using sonication), and one of these fragments will contain the joining point. Isolate it and then sequence it from both ends (paired-end).

Important: what we are sequencing is not part of the genome but a chimeric structure – it doesn't exist in the genome, is created through circularization (the yellow and the orange parts in the picture are not linked in the real genome).

Immunoprecipitation



DNA regions which are bound in vivo by a protein can be isolated from the rest of the genome by immunoprecipitation, and then sequenced. Immunoprecipitation get its name for the usage of antibodies that bind with high specificity and affinity a protein target, for example a transcription factor, histones etc.

To strengthen the binding between the protein and its DNA target sites, covalent bonds can be induced by cross-linking, this way facilitating their isolation. Cross-linking is usually reversible, so that the protein can be detached from DNA before sequencing it.

Exome sequencing

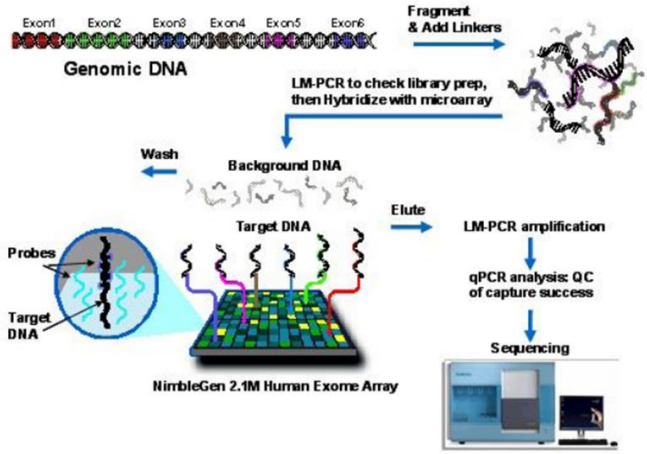
To focus on coding sequences within a genome, one can sequence only genomic fragments containing exons; this approach is called exome sequencing and is usually aimed at finding mutations responsible for diseases, which often affect protein sequences.

When we isolate exons, we can sequence them at high coverage obtaining many different reads only from a small subset of the genome, everything without wasting time and resources in sequencing parts of the genome which are unlike to host pathological mutations.

How can we find exons? Exons are not marked, so they are difficult to capture. To select fragments containing exons, capture arrays are used, which are chips onto which DNA probes are spotted that are complementary to parts of all known exons in a species genome. We design a set of oligonucleotide probes each one having sequence complementary to one specific part of a exon/gene. Fragments of the strands that are complementary to the probes will bind to them so captured by the array, while all the other fragments will be removed.

Fragments are then sequenced.

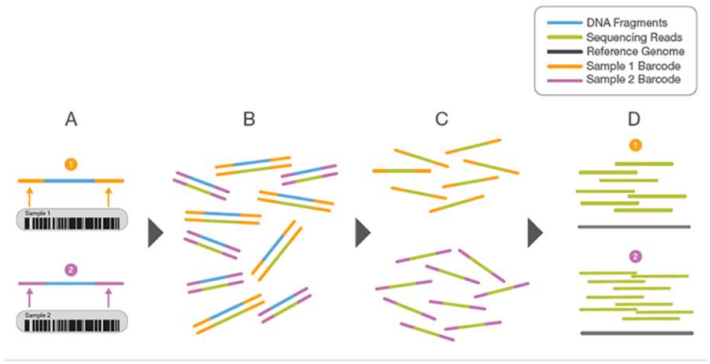
Commented [Ss1]: single-stranded sequence of DNA or RNA used to search for its complementary sequence in a sample genome



Multiplexing sequencing

Last variant of sequencing is called multiplexing or barcoding. To save time and reduce costs, it is possible to sequence multiple samples together, for example on the same lane of an Illumina flow cell.

To distinguish the sample of origin of each obtained read, short sequences, called barcodes, are attached to the genomic fragments together with the adapters. Barcodes are chosen as to have a different and characteristic sequence for each sample.



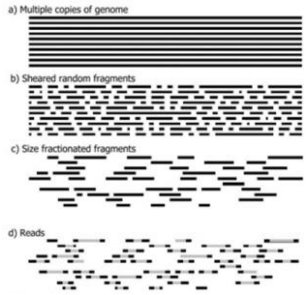
A. Two representative DNA fragments from two unique samples, each attached to a specific barcode sequence that identifies the sample from which it originated.
B. Libraries for each sample are pooled and sequenced in parallel. Each new read contains both the fragment sequence and its sample-identifying barcode.
C. Barcode sequences are used to de-multiplex, or differentiate reads from each sample.
D. Each set of reads is aligned to the reference sequence.

Genome assembly

In assemble you try to combine all reads reconstructing genome fragments or transcript (possibly up to entire chromosomes or entire transcripts).

In mapping instead, you have reads but you also have to use a reference genome. Therefore, you already know the original sequence, the problem here is to place reads in the right place.

Both can be approached using sequence comparison, but usually we have a large number of reads that require something more efficient (and faster) than sequence alignment.



You start from multiple copies of a genome, extracted from different cells. So, each sample contains different cells, each one carrying the same identical genome, you extract DNA from each one of these cells and then you fragment the genomes (random process).

You select randomly or by size a subset of fragments between the ones we have generated, and you sequence them using single-end sequencing or mate pairs sequencing.

Then, we will look for overlaps between reads because they come from the same part of the genome fragmented in a slightly different way.

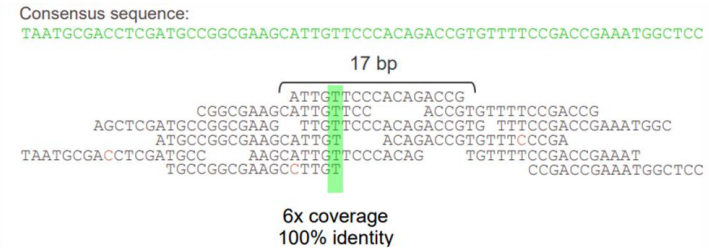
How to do this? We have to organize reads in the right order such that at the end we obtain reads organized by their overlap and we can go through this organization column by column deriving the representative sequence.

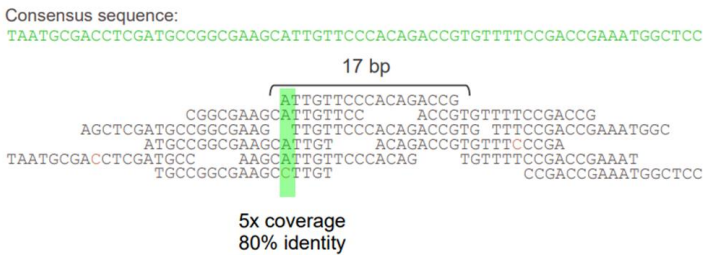
Therefore, in a **whole genome shotgun sequencing**, the starting point is multiple copies of the genome (e.g. extracted from different cells in a sample) which are then randomly fragmented, and each fragment (or a subset of them) is sequenced.

- 3 stages:
- overlap detection
- reads organization based on the overlap we have found
- identification of the representative sequence

More reads you have more the final output will be similar to the original one.

Coverage is the number of reads covering one particular position over the genome representing that position.
You can compute a local coverage position by position or an average coverage.





Coverage: average # of reads supporting each base of the consensus

Length of the assembled contig: ***L***
Number of reads: ***n***
Average read length: ***l***

Coverage ***C = n l / L***

Lander-Waterman model:
Assuming a uniform read distribution along the genome, having *C*=10 means having a gap each 1,000,000 nucleotides

In our example:

13 reads of length 17 nt = 221 nt used to build the 66 nt of the consensus.

Coverage is 221 nt / 66 nt = 3.3 (usually written 3.3X)

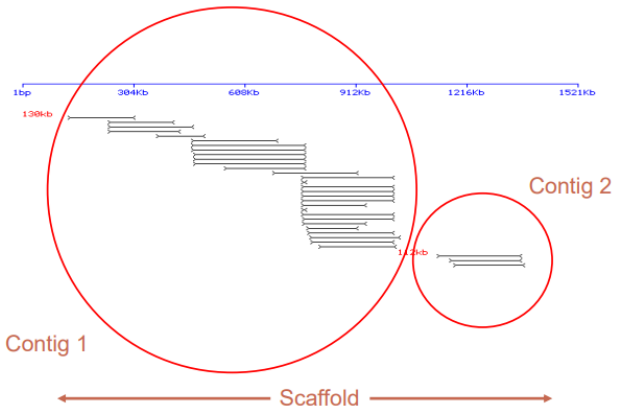
The coverage is an average value, and it is usually not uniform along the consensus sequence. This implies that some parts of an assembly (those having high coverage) can be considered more reliable, since the assembly there is strongly supported, and errors can be corrected.

When compute the overlaps, you might expect that reads that come from same part of the genome but fragmented in a slightly different way are identical at least in their suffix and prefix. But it is not always the case.
Some differences can depend on:

- the quality of the genome of the species we want to reconstruct,
- number of chromosomes of the same kind, so the number of homologous chromosomes (species can be haploid, diploid or polyploids – homologous are almost identical but there can be heterozygosity – reads coming from same chromosome might come from the father of from the mother and there might be differences)
- sequencing errors

(if there's a difference in the genomes of around 50% we are probably dealing with a diploid organism, while if one genome differs from the other by the 10% it is probably due to sequencing errors)

Genome assembly is something that changes every year. Some parts of the genome are not reconstructed (too many errors, wrong fragmentation (ex. sonication resistant), repeated sequences).
So, the problem is not difficult to describe, what is difficult is to do it efficiently and taking into account that overlaps might be affected by several aspects.



The result of an assembly procedure is never the whole sequence of all the genome chromosomes. Usually, the assembly is composed just by fragments of different size (contigs), for which the genome location might be unknown.

Contig: a contiguous set of overlapping reads

If contigs can be placed at specific genome locations (e.g. by the identification of genetic markers), or if the reciprocal position of contigs can be organized, we get scaffolds.

Genome assembly

It's one of the most important tasks in computational genomics and currently there are algorithms solving it but that still need a lot of improvement.

The basic idea is that starting from fragmented DNA that had been extracted from multiple. The genomes, after sequencing, are reconstructed based on the overlaps between reads (consider that sometimes there could be sequencing errors).
Read length depend on the sequencing method used, i.e., Illumina gives a large number of short reads.

Assembly starts from extension, which is basically overlapping reads to get a longer fragment.
A problem raises when different nt reads are produced from the same sequence in the genome or even when you're considering a diploid organism which carry 2 copies of the same gene.
Coverage is a general term to define how many reads are covering that piece of DNA. In principle, to discriminate between sequencing errors and heterozygosity sites, high coverage is needed.

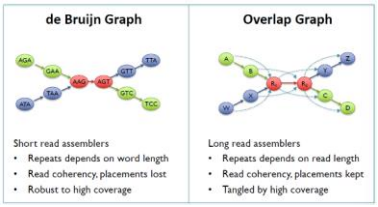
Heterozygosity sites are identified when the reads of a nucleotide are 50% of one kind and 50% of another. For example, considering this image, G and A have more or less the same occurrence in that position so that could be considered a heterozygosity site but it's difficult to be sure because the coverage is low.

Contigs appear when there are reads which are not overlapping with the rest of them, this leads to fragmentation. It could be possible to put together this contigs into scaffolds by pair end sequencing.
Large scaffolds represent chromosomes, generally no chromosome is complete 100%.

There are many different ways to assemble a genome, which leads to many different algorithms. These methods also depends on the characteristics of the sequencing data that are available.

General genome assembly steps:

1. Shear & sequence DNA
2. Construct assembly graph from reads (**de Bruijn** / **overlap graph**)
   In the overlap graph, each sequencing read is a node which are connected by an edge if they overlap (useful for small number of reads that are length enough).
   For large number of small reads, a de Bruijn graph.



3. Simplify assembly graph.
   The finding overlaps process has to be aware that there could be sequencing errors and know how to find them and how to behave with them.

4. Detangle graph with long reads, mates, and other links.

With this process we want to find the shortest superstring containing all the sequencing data, but the shortest is not always the real one. Anyway, we're going to consider this procedure first.

Shortest common superstring (SCS) problem

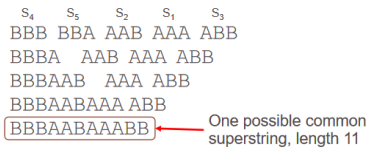
Given a set of strings S {s_1, s_2, …, s_n}, find the minimum length string T such that every string s_i is a substring of T.

T is a superstring because it is a string in which all the reads are included. In computer science, finding the shortest common superstring has been the beginning for genome alignment.

Brute force approach

A brute force approach could be the solution for the problem:

We can assemble strings by organizing them in any order and following two simple rules: if they have a prefix-suffix overlap, merge them; if not, concatenate them.

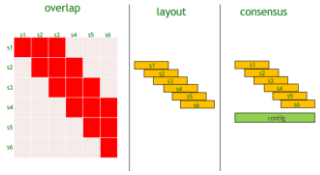


In the example, this could be a solution, but not necessarily the best one. So, the algorithm has to go through all the others and just compare the length keeping the shortest.

The problem is that brute force approach cannot be applied in reality because of their complexity: if S contains n string, the number of different ways in which they can be combined is n!.

This problem is np hard, which means that it is necessary to look for alternative algorithms, like a *heuristic* approach.

Overlap-Layout-Consensus (heuristic approach)



This algorithm is based on three steps:

- Overlap
- Layout
- Consensus (just draw the string)

This approach is one of the oldest one, but it is now appearing again among the newest software.

Overlap Matrix

The overlap matrix is a matrix in which every row and column correspond to different reads.
This matrix is not symmetric because we're looking for overlapping in suffixes, so the suffix of s1 might overlap with the prefix of s2 but s2 prefix might not overlap with s1' suffix.
i.e., if the four nts of the suffix of S1 are the same of the prefix of S2, the score in the matrix will be four.

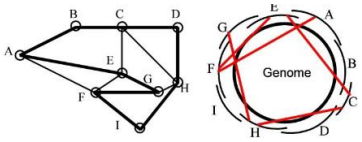
It is therefore possible to build a matrix in which each row and column correspond to a different string, each cell corresponds to a different couple of string where it is stored the score of the alignment of the suffix and prefix of the overlap.

Layout

Now the objective is finding the best way to combine the reads to get the superstring with the precomputed scores.

The approaches to solve this problem are some greedy algorithms and some based on clustering.

To better understand this problem, an overlap graph is used, which is basically a greedy strategy to find the best path in the graph.
It is a graph in which nodes are the reads, and edges connect nodes if the corresponding reads have a significant overlap. With this representation, genome assembly can be converted into the problem of finding a path in the graph passing through each node once, maximizing the overlaps.
This means that all available reads will be included in the assembly.
The path dictates the ordering of the reads, which is their layout, and there could be many. From the layout and the read overlaps, a consensus sequence can be drawn.



A walk is a series of alternating nodes. But our objective is to find a path, which is a walk without repeated vertices. The nodes v_0 and v_n are called path extremes.
When dealing with a circular genome, the objective would be to find a cycle in which $v_0=v_n$.

With this approach the problem still results np-hard because if S contains n strings, the number of different ways in which they can be combined in a path is again n!. And even if we want to find the path with maximum weight passing through each node, only once per node, the problem will still result np-hard (Hamiltonian path).

Find at each vertex of the path which is the best next one to follow. This is a **greedy approach**, where local best solutions should result in global best solutions. It is easy to find the wrong path!

How to setup a greedy algorithm

Given a set S of strings, let's first compute all possible suffix-prefix overlaps for each possible pair of strings in S, scored as the number of identical characters, to obtain the overlap matrix.
Note that suffix-prefix overlaps are not symmetric, in the sense that the suffix of string S_i can overlap with the prefix of S_j, but not necessarily the other way around.

	S_1	S_2	S_3	S_4	S_5
S_1	-				
S_2		-			
S_3			-		
S_4				-	
S_5					-

EXERCISE

Given a set of string, compute all the possible overlaps and then combine them in a layout dictated by this greedy approach.

First, we need to consider each S_i string and check the overlaps with all the others.

S_1=ATCGGA S_3=AGCTAC

S_1=ATCGGA S_4=CGGATT

	S_1	S_2	S_3	S_4	S_5
S_1	-	0	1	4	0
S_2		-			
S_3			-		
S_4				-	
S_5					-

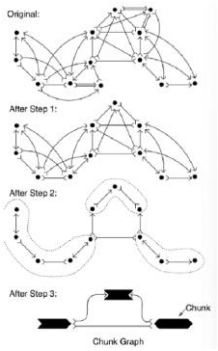
Let's imagine that we are given the following strings:
S_1=ATCGGA
S_2=TACCCA
S_3=AGCTAC
S_4=CGGATT
S_5=TTGCTA

When the overlap matrix is complete, we can start working with the "overlap graph" even tho we're not going to build it in our code because it's just a simplifiaction we use to understand visually what the overlap matrix means.

The alignment of the reads begins with the connection that has higher score. This leads to a shortening of the overlap matrix, since now two strings are connected together it's necessary to remove the columns and rows corresponding to those two sequences.
Now it is necessary to recompute the overlaps for the newly merged string, and go on with this process until one superstring is left.

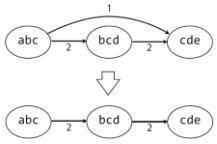
A problem raises when at one step there are two or more equivalent solutions, it is not possible to determine which one is the best one to follow. So, the result will return two or more solutions without having the possibility to find the correct one.

In order to reduce the probability of finding a wrong solution, we could **simplify the graph** such that every greedy algorithm can perform well.



Overlap graphs tend to be redundant in many ways, so three things can be done to simplify it:

1. Contained reads removal
   Short reads that are substrings of longer reads don't advance the assembly, remove those nodes and all of the edges.
   It is very important to identify them correctly, otherwise if deleted they would bring loss of information.
2. Transitive (inferable) edge removal



   One idea for simplifying the graph is that in many edges between nodes can be removed since these edges are redundant, not providing any additional information.
   Some edges can be transitively inferred from others, and as such they can be safely removed from the graph.
   The overlap graph simplification can also help in identifying repeated regions and more precisely determine their boundaries.

   If we remove edges the number of nodes remain the same, but removing nodes is deleting part of the sequence, which reduces the coverage. So do not remove nodes but edges because there's no loss of information for them if they are transitive. Coverage is needed for checking for sequencing errors among all the other things, so it is important not to delete nodes.
   This step simplify a lot the solution's calculation.

3. "Chunkification"
   Linear subgraphs define segments that can be easily assembled (unitigs).

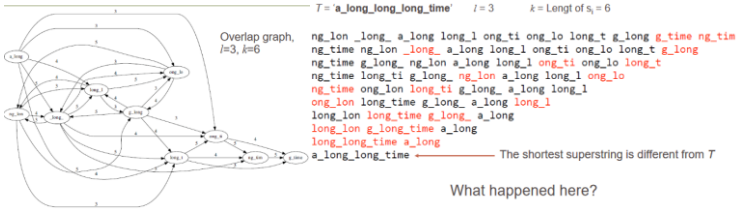
Generally, after simplification, it is possible to reach unity (linear graph where no decision need to be taken) where the complexity is very low.

Sometimes messy subgraphs in the middle can raise, in which simplification is not able to fully solve them. These can just be solved by greedy algorithms and the complexity is still definitely lower than before the simplification.

Another problem that could be encountered when looking for the shortest possible superstring:

T = '**a_long_long_long_time**' $l = 3$ k = Lengt of $s_i = 6$

```
ng_lon _long_ a_long long_l ong_ti ong_lo long_t g_long g_time ng_tim
```



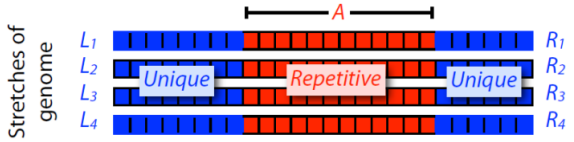
What happened here?

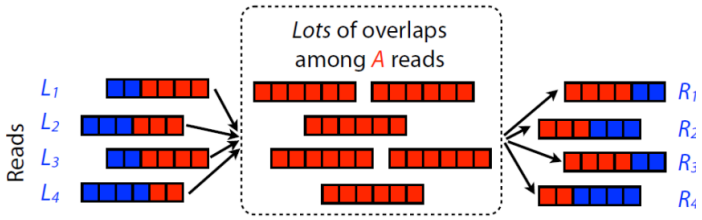
Sometimes even if we're able to find the shortest superstring, this might not be the correct one because the original one contained repeats. These repeats collapsed during the assembly one onto the other, loosing information.

Often this problem cannot be solved, only making the reads longer could help (using NGS).

SOFI Lesson 14 17.04.2023

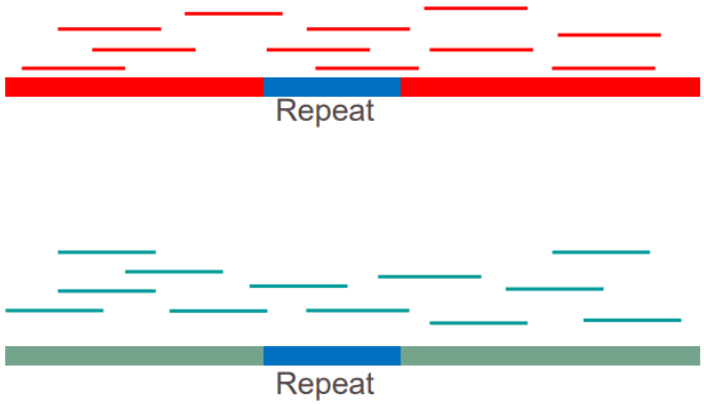
Repeats can also create problems to assembly algorithms when they are not repeated in tandem but spread all over the genome This kind of repeats can lead to misassembly since it's difficult or impossible to correctly connect the sequences upstream and downstream the repeated element, unless the read length is higher than that of the repeat.



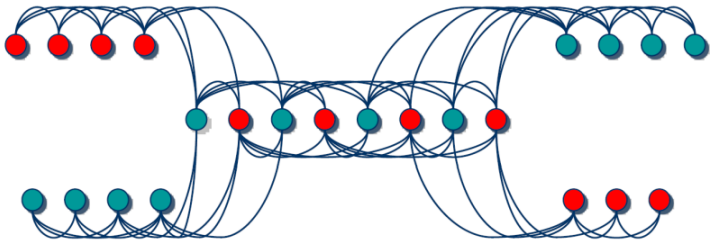


Let's consider the case in which a repeat A is present in different points of the genome. After fragmentation, we can't know which paths in the repeat correspond to which paths out of the repeat. Here, for example, we can't know which left segment (Li) should be joined with which right segment (Ri).

Let's consider these two different genome fragments, that share the same repeated element. In a whole genome shotgun experiment, reads from both fragments are mixed together.

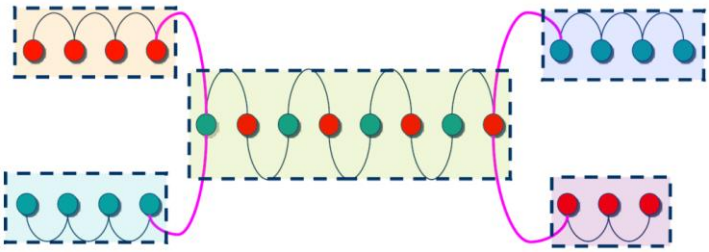


Checking the overlaps among all these reads would lead to an overlap graph with the following structure:

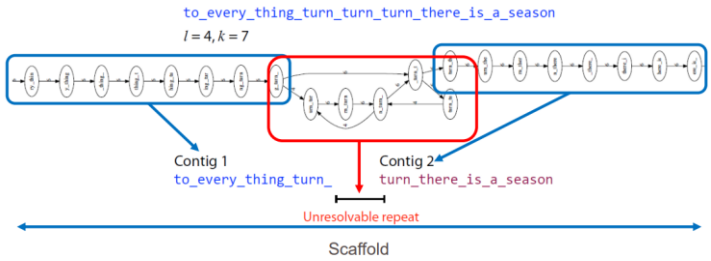


Five discrete regions can be identified, two for each genome fragment, and the central one where reads carrying the repeat sequence from both fragments are mixed together, since their sequence is similar.

With graph simplification, one can end up with discrete subgraphs for which, in the best case, there is no need to look for Hamiltonian paths because the path is clear (nodes have only one incoming and one outgoing edge).



In this case, the boundaries of the repeats can be easily identified, since the repeat starts with a node with two (or more) incoming edges, and ends with a node with two (or more) outgoing edges.



Even if we are not able to fully resolve the central repeat, we still know that contig 1 and 2 are part of the same supersting (e.g. a chromosome), are close to each other, and are separated by a repeat composed by an unknow number of elements. So we still get a scaffold.
Sometimes this is not possible, for example when we have multiple repeats present in a genome region. In those cases, it might not be possible to understand the right order of the surrounding non-repetitive regions.

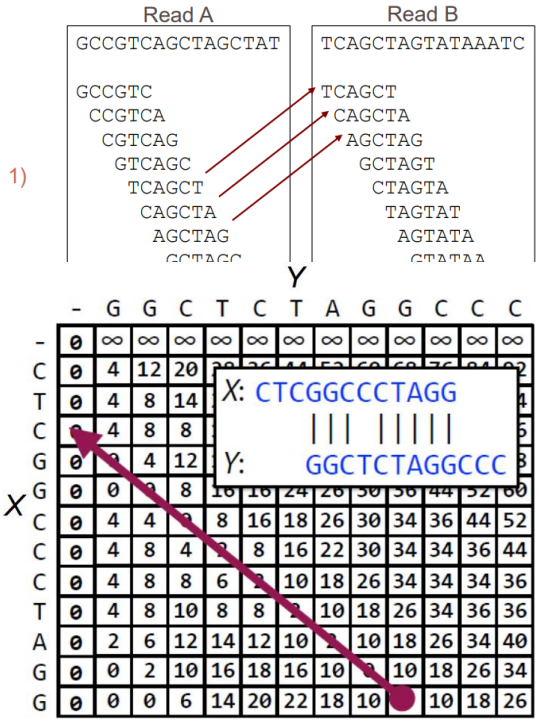
The second way we can improve the Overlap-Layout-Consensus approach is to improve the Overlap step, by reducing the time required for detecting suffix/prefix overlaps that, when the number of reads is large, can be a severely limiting factor.
Two broad categories of methods are commonly employed:

1. Methods based on hashing
2. Methods based of graph representation of strings

In both cases, the idea is to speed up the finding of prefix/suffix matches between pairs of reads. Then the initial match can be kept or improved by a true dynamic programming alignment.

Hashing-based method

In hashing-based methods, the idea is similar in principle to that used by FASTA or BLAST, that is dividing each read in overlapping substrings of length k (kmers), then looking for pairs of reads sharing one or more k-mers. Therefore, reads are split into k-mers: too frequent k-mers are discarded, while the remaining ones are hashed.



1. Reads sharing a certain minimum number of identical k-mers are selected

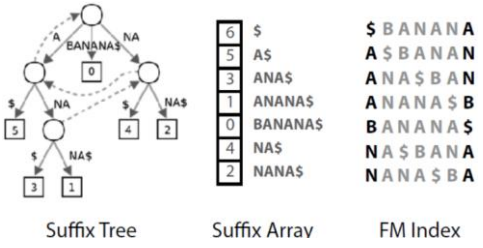
2. Shared contiguous k-mers are merged and the resulting initial alignment is evaluated 3. Finally, reads are aligned using a modified dynamic programming alignment algorithm

How do we force a NW algorithm to find prefix/suffix matches?

We need to change initialization, such as setting the first column as all 0s (any suffix of X is possible), and the first row as $-\infty$ (must be a prefix of Y). Then, traceback from the maximum value in the last row until a 0 in the first column. By starting the traceback from the maximum value in the last row, we ensure that the optimal alignment includes the last character of the string X (i.e. contains the suffix of X). Moreover, the traceback will never reach the first row, because the penalty for it is too high, and instead will reach the first column, therefore the optimal alignment is going to start from the first character of the string Y (i.e. the alignment contains a prefix of Y) and reach the last character of the string X (i.e. it contains the suffix of X)

Graph-based methods

Other assemblers instead employ complex constructs with which performing string matching operations is nevertheless fast, such as suffix trees, suffix arrays or other forms of string indexes.



Suffix tree

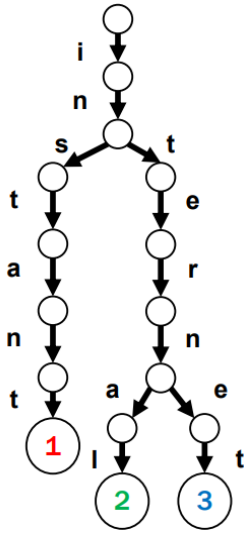
A suffix tree is a compressed structure containing all suffixes of a string. Suffix trees allow fast implementations of many important string operations.

The suffix tree for a string S of length n is defined as a tree such that:

- The tree has exactly n leaves, numbered from 1 to n
- Except for the root, every internal node has at least two children
- Each edge is labelled with a non-empty substring of S
- No two edges outgoing from a node can have string-labels beginning with the same character.
- The string obtained by concatenating all the string-labels found on the path from the root to leaf i corresponds to the suffix S[i..n], for i from 1 to n.

Keys: *instant*, *internal*, *internet*

Key	value
Instant	1
Internal	2
Internet	3



To get to suffix trees, let's first define a trie, which is the smallest tree representing one or a collection of strings (keys) S, such as that:

- Each edge is labelled with a character c ∈ S
- For a given node, at most one child edge has a label c, for any c ∈ S
- Each key (string in S) can be reconstructed along some path in the trie, starting from the root node.

Nodes are empty, meaning that they have no value, except the leaves (terminal nodes), which in this example have a value corresponding to one of the keys used to build the trie (i.e. 1, 2, 3).

How to look for the presence of a given string in the trie? Start at root and match successive character of the query string (key) to edges in the trie:

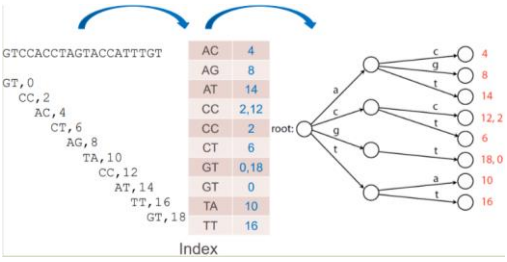
1. If you match all characters of the key, and the last node has a value, then the key is in the trie
2. If you match all characters of the key, and the last node has no value, then the key is a substring of one the keys in the trie
3. If you 'fell' off the trie, then the key is not in the trie.

Examples:

- Matching 'infer' --> we fell off the trie, since no outgoing edge from the last reached node is labelled with character 'f'.
- Matching 'interesting' --> we fell off the trie, since we can match 'inter' but when we arrive at the edge 'r' we can't find an 'e' to continue
- Matching 'insta' --> there is a path matching the entire query string 'insta', but the last node has no value. This means that the query string is not one of the keys of the trie, but is a substring of one of them
- Matching 'instant' --> there is a path matching the entire query string 'instant', and the last node has a value. This means that the query string is a key of the trie.

In general, if the total length of all strings (keys) used to build the trie is N characters, the trie has ≤ N edges, therefore is a compact representation of all keys. To check for the presence of a key P of length n in the trie, the path traverses ≤ n edges, and the search is done in linear time O(n).

A trie can be built for any set of strings, and allows for the fast identification of the presence of a query string in the trie. Tries can be used to represent k-mer indexes, i.e. to map where each k-mer occurs within the sequence. Here a string was decomposed in all non-overlapping k-mers of length 2.

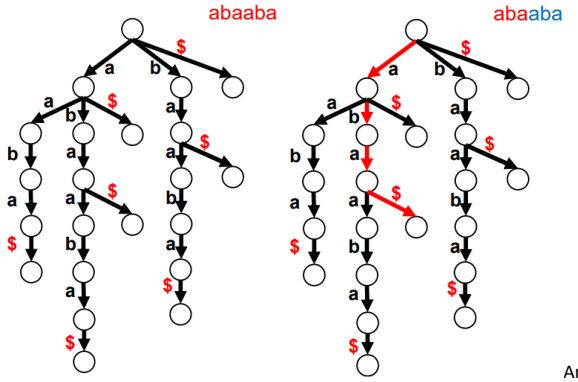


If we want to know for example where a dimer 'CC' occurs within a string, we can look for a path in the trie representing all dimers from the key string to a leaf.

But this is not the only way; a trie can be constructed from any set of strings. Since we are interested in finding suffixprefix overlaps, we can use a trie representing all suffixes of a string. For example, for the string abaaba, the suffixes are:

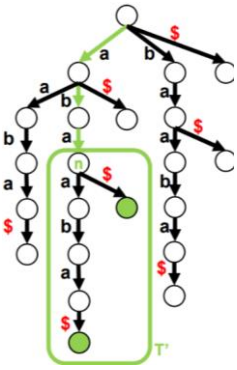
```
abaaba$
 baaba$
  aaba$
   aba$
    ba$
     a$
```

We can build a trie in which all keys are suffixes of a string (to facilitate matching operations, we add a special character, usually $, that is not included in the string alphabet, to indicate the end point of each suffix).



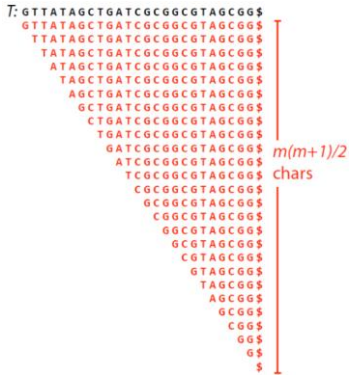
Any path from root to leave is a suffix.

Let's consider these examples:
- Matching 'abaa' --> key is not a suffix but just a substring, since the edge outgoing from the last reached node is not labeled with a $
- Matching 'aaba' --> key is a suffix, since the edge outgoing from the last reached node is labeled with a $
- Matching 'ba' --> key is both a suffix and a substring, since one of the edges outgoing from the last reached node is labeled with a $, while the other outgoing edge is not. That's why we need an exit character!! If we did not add these exit points, it would be difficult to determine whether a key is a suffix or just a substring.



As a consequence, suffix tries can also be useful for checking for repeated substrings in the set of keys used to build the trie. For a string S, how many times S is found in a trie T?
Follow from root the path with the labels spelling S: if we fall off the tree, the answer is 0, while if we end up at node n, the answer is the number of leaves in a subtree T' rooted at node n.

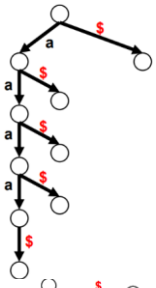
Example: The string 'aba' occurs twice in the trie.



How big is a suffix trie? If the length of a string T is m characters, then the number of all possible suffixes (including T itself) is m and the total number of characters of all suffixes is m(m+1)/2. From the number of suffixes, can we say how many nodes and edges the resulting trie is going to have?

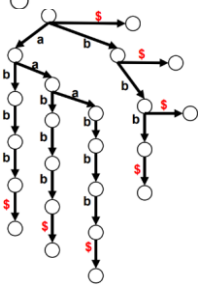
To understand the lower bound (best case) of the size of a trie, we can wonder, for a given string length = m, which is the string that can be represented with the smallest suffix.

It is a string composed by only one type of character. For example, consider the string T='aaaa$'. Its trie is composed by 10 nodes and 9 edges.



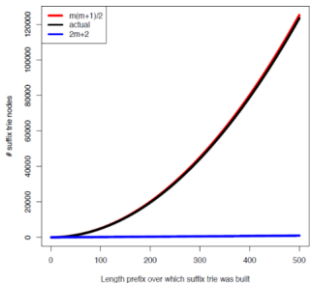
In general, for this kind of strings the trie contains:
- 1 root
- m nodes with outgoing 'a' edge
- m+1 nodes with incoming '$' edge

Total: 2m+2 nodes
Hence, in these cases the size of the trie grows almost linearly with m



Let's now consider instead the string T='aaabbb$'. Its trie is composed by 23 nodes and 22 edges.
In general, for this kind of strings composed by two different characters repeated n times such as 2n = m, the trie contains:
- 1 root
- n nodes along 'b' chain
- n nodes along 'a' chain
- n chains of n 'b' nodes hanging off the 'a' chain (n 2 in total)
- 2n+1 leaves

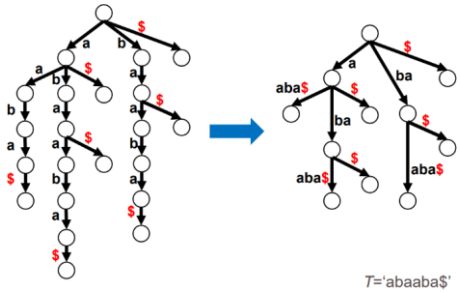
In total, n^2 + 4n + 2 nodes, where m = 2n



For real biological sequences, the suffix trie growth is much closer to the worst case (quadratic growth) than to the best (linear growth). The actual growth is almost identical to the worst case.

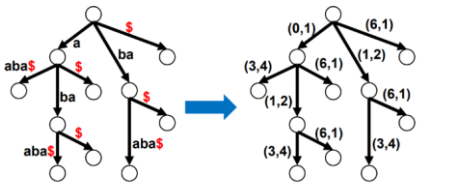
There is another problem: suffix trie built for a large size genome will not fit into memory!

To compact the tree, we can coalesce all non-branching paths into one single edge with a string label (instead of a single character label). This simplification reduces the number of edges and nodes, and guarantees that all non-leaf nodes have >1 child node.



In general, for a string of length m, the number of leaves is m, the number of non-leaf nodes is ≤ m-1, and the total number of nodes is ≤ 2m-1. Therefore, the size of the trie grows linearly with m.

But we need to consider that also the edge labels must be stored in memory, and, for very long strings, these labels can quickly become huge (the total length of edge labels grows with m^2).
We can save a lot of memory by not storing each label as a string, but as a tuple of two values: the first is the offset (where the string label starts within the full string) and the other is the length of the label (i.e. of the substring). For example, for T='abaaba$', the suffix 'aba$', which is one of the labels of the suffix trie, can be substituted with the tuple (3,4), which tells us that the suffix starts at the fourth character of the string and is 4 characters long. In this way, the full size of the suffix trie (nodes + edges + labels) grows as O(m)



For each path in this trie, we can easily reconstruct the suffix sequence of characters by fetching from the string T the substrings specified by the set of tuples (offset, length) encountered along the path. Consider T='abaaba$' and the path from root to leaf (0,1),(3,4).
We get 'abaaba$' for offset 0 and length 1, then 'abaaba$' for offset 3 and length 4.
The resulting suffix is the concatenation of the two, hence 'aaba$'

(As we can see, the ranges specified by the tuples do not need to be consecutive along the string).

It is common practice to report in the leaf nodes the offset (i.e. the starting point) of the suffix in the whole string. The total length of edge labels from root to leaf is called the **label depth**. This is to not be confused with the **node depth**, which is the length of the shortest path between a node and the root.

So, what can we do with suffix trees?
We can do all the string matching operations that we saw before, with the only difference being that we need to take into account that edge labels are coalesced from possibly many singlecharacter edge labels.
As we've already seen, we can also count the number of occurrences of a string P in a tree T, and also get the position of all the matches of P in T.

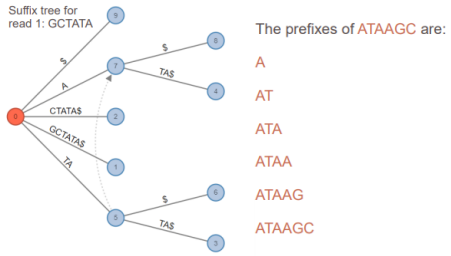
E.g., P = 'ab', T = 'abaaba$'
Step 1: walk down 'ab' path; if we fall of the tree, there is no match
Step 2: visit all leaf nodes below, and report each leaf offset
First step is O(n), second step is O(k) if k is the number of times P is in T .- overall time is then O(n+k)

Problem	Complexity
Time: does *P* occur in *T*?	O(n)
Time: count *k* occurrences of *P*	O(n+k)
Time: report *k* locations of *P*	O(n+k)
Space	O(m)

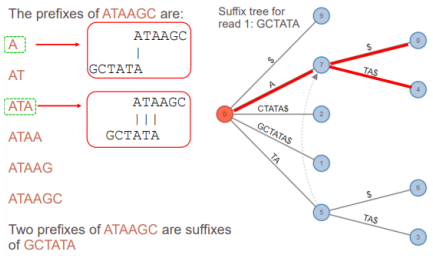
For a string T of size m, and a query string P of size n, if k is the number of occurrences of P in T, then the time and space requirements of common string matching problems are:

EMI Lesson 16 5.05.2023

How can we use a suffix tree to get the suffix/prefix overlap of two strings A and B?
We have a set of strings(library of sequencing reads), we can compute a suffix tree for each one of them and then I can see if there is an overlap between them.
For example, imagine having the strings 'GCTATA' and 'ATAAGC'.
I start creating the suffix of the first tree and then I match all the prefixes of the other one
and then I do the opposite.

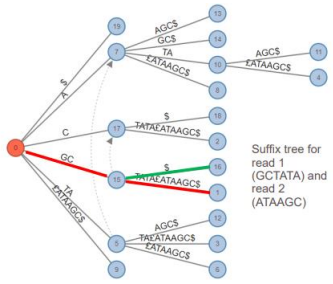


In our example, this is the suffix tree of the first read and on the right there are all the possible prefixes that can be generated from the read 2.
I check all the possible matches between them.



As I can see, in this case there are two possible suffix/prefix overlaps.
Then I do the opposite and I generate the suffix for read 2 and search for all the possible prefix of read 1.

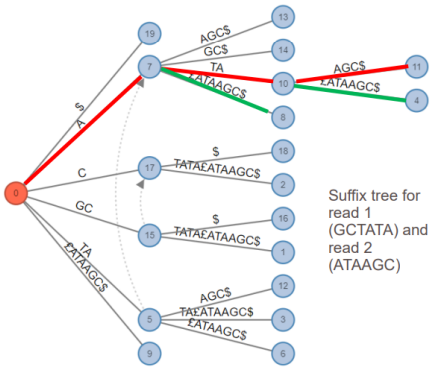
Anyway this is not an efficient way to find the overlaps. What I can do to speed up the process, is to create a unique large suffix tree for the concatenation of the strings. I can create this suffix tree for a very big number of reads.



The $ means that I have already found a match.
In this case I use a special character "£" to separate the two strings.

How can I use this structure to find suffix-prefix matches?
I need to check if there is a path reconstructing one of the input strings. Along it, I need to check if there is a node with an outgoing edge (green) labelled with the end character, this means we found an overlap. In the example it means that GC must be the end of one of the strings (it is the beginning of the read 1 and the end of read 2).

I can then search for the path spelling the entire second string. In this case I need to check if there are nodes with outgoing edges that start with a separation character: it means that the edge is the end of read 1.
In this case 'A' is prefix of read 2 and suffix of read 1.



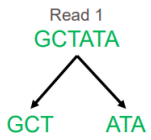
Suffix tree for read 1 (GCTATA) and read 2 (ATAAGC)

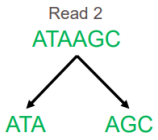
In general, for each string, walk down from root and report any path ending with a terminator or separator character (indicating a prefix) in which there are nodes having also outgoing edges labelled with a different terminator or separator character (indicating a suffix)

At the end the complexity is more or less as before:

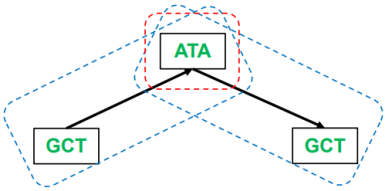
Time to build generalized suffix tree:	O(*N*)
Time to walk down red paths:	O(*N*)
Time to find and reports suffix/prefix overlaps:	O(*a*)
Overall:	O(2*N*+*a*) Linear time

Alternative way of genome assembly
Even with all the optimization tricks that we described for the overlap and layout steps of the Overlap-Layout-Consensus (OLC) algorithms, these algorithms still do not scale well for very large datasets of short reads. For this reason, a new generation of assemblers was introduced, in which a different strategy for overlap detection and graph representation of read sequences is used. The main idea is that we can simultaneously detect overlaps and build the graph.



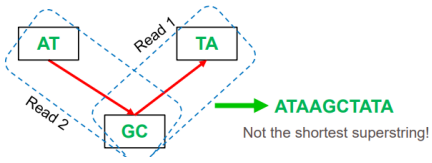


I start splitting the reads in suffix and prefix. Given the set of substrings I will build a graph in which each substring will be a node



I start creating this graph and, if a substring is already present, I don't need to add it.
Since all the substrings are suffix and prefix, the nodes in common are overlaps.
Each node has an outgoing edge with a direction, following it I can recreate the sequence → GCTATAGCT

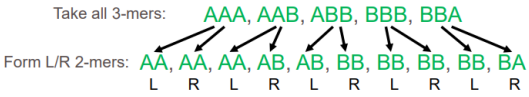
If I change the length of the suffix/prefix (k), in the previous case it would prevent the concatenation of the two strings since TATA is different from ATAA, for this reason if k is too big the graph could be disconnected. Also some problem could raise if k is too short, we could find a superstring that is not the shortest:



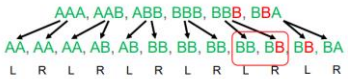
In the end the successful of this strategy depends on the right choose of k, so sometimes I need to try with multiple k

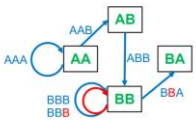
To summarize, given a set S of strings, we need to:
1. Split each string si in the set S into a prefix and a suffix of fixed length k
2. Add to the graph a node labelled with the prefix and a node labelled with the suffix, but only if there are no nodes already in the graph with the same label
3. Draw an edge to connect the prefix node with the suffix node

Suppose we have a genome and a set of substrings of 3-mers and then split it into two overlapping substrings of length 2. Call these the left (L) and right (R) 2-mers. We can from that create a graph:

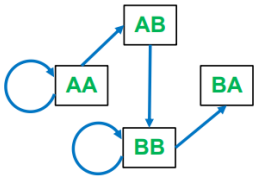


The circular edge means that I repeat the same 2-mer twice.



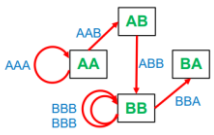


If I have two identical strings, I need to point it out by adding a multi edge:



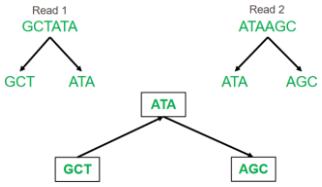
This graphs are called *De Bruijn Graqph* (DBG)

Once we have built the graph how can we create a path that lead to a superstrings, hopefully the shortest one?
Before I needed a string passing through each node once, now I want a graph passing through each edge (because the nodes are not entire strings but just substrings). In this case is fine to pass through the same node multiple times.
We first search for the starting node (not incoming edge). From here I can start writing my superstring, since I know the k I choose. I end when I reach the node with not outgoing edges.



T=AAABBBBA

When we pass trough each node only once I compute an Eulerian Path. I can do it in linear time.
Two main advantages:
1. no complex way to compute overlaps
2. when I have the graph I can compute the superstring in linear time

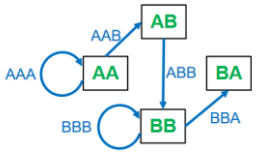


After the graph is constructed, we need to follow edges which do not represent anymore the entire reads but just prefixes and suffixes.
In this case the node ATA represent the overlap, it has an incoming and outgoing edge.

K is the size of prefixes and suffixes
If the lengths of the suffixes and prefixes are too long, it means that we could not have overlaps between reads.
When instead the lengths suffixes and prefixes are too short, it might mean it is not the correct shortest superstring because there might be repetitions.

G = AAABBBA

Imagine having perfect sequencing: Take all 3-mers: AAA, AAB, ABB, BBB, BBA
Perfect sequencing is when the whole genome is covered and each read is shifted by +1.

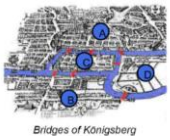


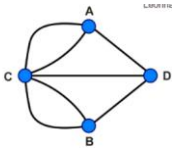
Start by creating the graph one read at the time.

An edge in this graph corresponds to an overlap (of length k-2) between two k-1-mers. More precisely, each edge in this graph corresponds to a length-3 k-mer from the input string G.

We need a path passing through edges.

A path passing through each edge of a graph exactly once is called a **Eulerian path**. The characteristic of this path is that it is possible to find its solution efficiently (polynomial time).

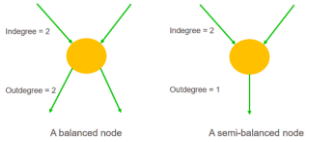

Bridges of Königsberg



Euler is consiedered the father of graph theory, which was first introduced by the problem of the bridges of Konigsberg.

It gave criteria to understand wether there is a path that allow to cross every bridge only once.
It is important because this allows to recognize whether the solution of our problem can be achieved or some adjustments are needed.

Criteria: A directed, connected graph is Eulerian if and only if it has at most 2 semi-balanced nodes, and all other nodes are balanced.



The **degree of a node** in a graph is the number of edges incident to it.
A node is **balanced** if its indegree (i.e. how many incoming edges the node has) is equal to its outdegree.
A node is **semi-balanced** if the indegree differs from the outdegree by 1.
A graph is **connected** if each node can be reached by a path from any other node.

Back to the example to check whether the DBG is is Eulerian.

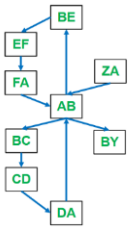
Node AA has 2 outgoing edges and 1 incoming edge, so it's semi-balanced.
Node BA has 1 incoming edge and no outgoing edge, so it's semi-balanced.
The other 2 nodes (AB, BB) are balanced.

Therefore, the graph is Eulerian.

Moreover, in general, semi-balanced nodes represent the start and the end of the genome sequence.

The problem is that when computing the alignment in a real situation, this approach is most likely not to retrieve the correct solution, or not retrieving one at all.

Taking for example T='*ZABCDABEFABY*'



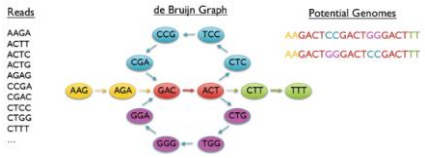
This is the DBG that is created.
It is possible to check that it is Eulerian.

The problem rises when the genome wants to be found because there are two equally good possible paths.

ZA→AB→BC→CD→DA→AB→BE→EF→FA→AB→BY

ZA→AB→BE→EF→FA→AB→BC→CD→DA→AB→BY

This represent another problem that could rise is that the graph is not Eulerian the solution is to find a path passing through each edge but more than once.

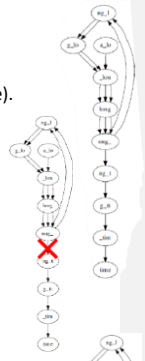


Is it still possible to find a path that leads to the correct genome? (The issue is similar to the one of the example before)

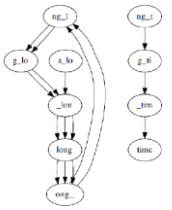
Now the graph is not Eulerian, and there are 2 solutions to the alignment.
In both paths, the red edge in the middle of the graph is used more than one time. Nevertheless, even if these paths are not Eulerian, one of the two should be the correct solution.

Few other cases in which this strategy might fail:

Consider a sequence T = 'a_long_long_long_time' perfectly sequenced (which never happens in real life).



- Imagine now that one of the k-mers of the string T is not present in the string dataset. Missing k-mers can be a problem, since that might cause the de Brujin graph to break down to two or more disconnected graphs, which might (or not) be Eulerian.
  We don't have anymore the connection between the prefix 'ong_' and the suffix 'ng_t' of 'ong_t'.



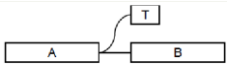
We obtain two disconnected graphs, since we are missing the k-mer that would have connected the two. Each disconnected graph is Eulerian, hence we can at least compute a Eulerian path and a superstring for each of them, but we don't know how to connect the two superstrings, and we don't even know which one comes before the other in T.

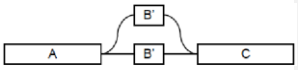
- Other problems might arise when one or more k-mers are copied more than once.
  This often happens when the coverage is not uniform, causing an unbalanced increase of multiedges.

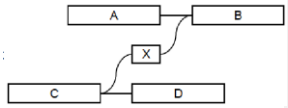
- What happens if one of the k-mers is wrong?
  For the same T, let's imagine that the k-mer 'long_' contains an error that converts it into 'lxng_'. The resulting de Brujin graph is disconnected, moreover, the largest component is not even Eulerian, since it contains 3 semi-balanced edges, violating the requirement of having at most two.
  We might decide to remove the disconnected part of the graph because we recognize the sequencing error, but the main graph will still be not Eurelian and retrieving the target sequence could be tricky.

The idea is to simplify the graph, not only to make the process faster, but also to make the probability of finding the correct sequnece grow.
The topology of the graph can reveal some commonly occurring problems, usually a consequence of sequencing errors

and repeats, that can be solved in many cases. To solve these issues it is possible to recurr to assumptions that hold true for most cases.

- **Tip**
  For a node, there are two outgoing edges that connect with nodes that have different characteristics, for example considering the length of the chain of the nodes that follow.
  

  A tip is a chain of nodes disconnected at one end, originating from a node from which a longer chain also originates. If the tip is short (usually <2Kb in genome assembly), or there is a remarkable difference in length between the tip and the supposed main chain, the entire chain of nodes composing the tip is removed.
- **Bubble**
  The difference with the Tip is that the bubble goes back to the same chain and the Tip doesn't.
  

  Bubbles are resolved by first determing the sequence of each of the two chains of nodes in the bubble, then comparing the two obtained sequences, finally merging them if they are sufficiently similar.
  Bubble collapsing (removal of bubbles) might be dangerous because it could not be a sequencing error but could be caused by heterozygosity.
- **Chimeric connection**
  Chimeric connections are short chains of nodes joining two longer chains. Are generally caused by sequencing errors or by random similarity. If the chain is short it can be removed without being a problem usually.
  

If it is possible to identify all these topological features, it is possible to simplify a lot the graph allowing a more clean alignment of the reads.

SOFI Lesson 18 09.05.2023

We need to build this… AAABBBA

…from these AAA, AAB, ABB, BBB, BBA

AA, AA, AA, AB, AB, BB, BB, BB, BB, BA

L R L R L R L R L R

Up to this point, we started from a set of substrings of the unknown superstring, and we divided them into a left and right sub-substrings. In genome assembly projects, each read (i.e. a substring of the genome) is instead not split just into a left and right part, but into overlapping k-mers, and each one of them is then split into left and right.

This is a single read, NOT the superstring that we want to build ATGCTAT

ATG, TGC, GCT, CTA, TAT

AT, TG, TG, GC, GC, CT, CT, TA, TA, AT

L R L R L R L R L R

The process is repeated for all the reads in the dataset, and the de Brujin graph is built from all left/right k-mers from all reads. The consequence is that the graph does not grant anymore the identification of prefix/suffix matches between different reads.

We have prefixes and suffixes that are not prefixes/suffixes of our sequencing read, they are prefixes/suffixes of the central part of the sequencing read.

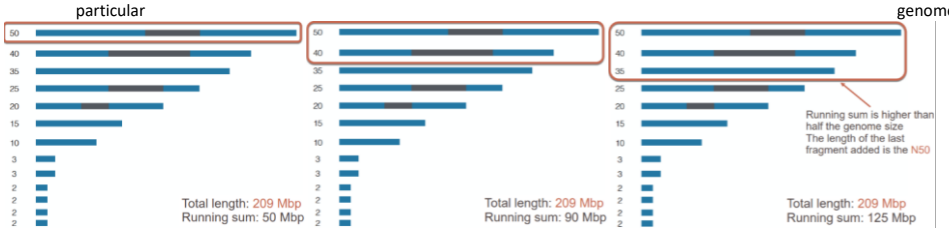
The reason is that, if we have 150bp reads, and divide each of them into a left and a right part, each with length 149bp, the required overlap is too large to be realistically expected. If instead we split the read in large k-mers, and divide these k-mers in left/right k-1-mers, we should be able to detect overlaps more easily, even though we cannot know whether these overlaps are of the prefix/suffix kind. Yet, if k is sufficiently large, random overlaps should be infrequent, and the majority of them is going to be prefix/suffix overlaps.

The choice of k is crucial:
- if k is small: all suffix/prefix overlaps are detectable, but also some random overlaps, high risk of chimeric connections, also repeats might collapse
- if k is large: not all prefix/suffix overlaps can be detected, but those that are found are less likely to happen by chance, high risk of disconnected graphs

So how can we choose an optimal value of k? There is no way to know in advance which k would produce the better assembly, so we proceed by trial-and-error. However, k should never be <20; a starting point could be 2/3 of the read length, and then decrease k gradually.

How good is an assembly can be estimated by some parameters, based on the size and distribution of the contigs and/or scaffolds that were obtained:
- Mean and max length of contigs and scaffolds
- Total length of the reconstructed genome
- Average coverage (coverage = average number of reads that participate to the assembly of the genome)
- N50 I check which is the smallest fragment in this sorted list of strings corresponding to half the size of the genome.

Example: consider 9 contigs of lengths 2,3,4,5,6,7,8,9,and 10. Their sum is 54 (so it is the size of the genome), half of the sum is 27. To get the N50 value start at the longest read and add the length of the next shorter one until you reach (at least) 50% of the total number of nucleotides: 10 + 9 + 8 = 27 (half the length of the sequence). Thus, the N50=8, which is the size of the contig which, along with the larger contigs, contain half of sequence of a particular genome

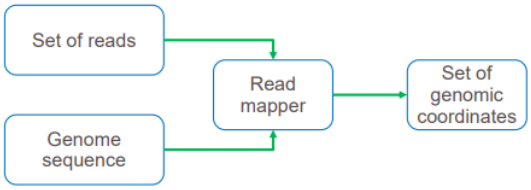


When the assembly is finished, make it available for the scientific community – deposit this assembly in a public database (to make it useful you have to annotate – where genomes features are).

EMI Lesson 19 11.05.2023

READ MAPPING

To map a sequencing read on a genome means to predict the genomic locus from which the read originates.

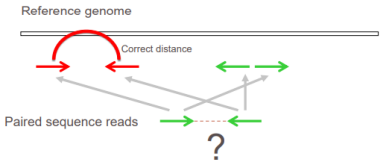


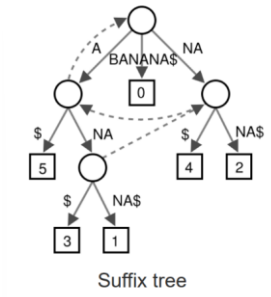
Differences between the read sequence and its locus of origin not only arise because of sequencing errors. It's important to consider that when mapping a read, we are comparing to a reference genome a read

sequenced from the genome of cells in a sample, and the genome of this cells is surely different from the reference genome in possibly a large number of positions. Differences between the reference genome and the sample genome can be of any kind, from single nucleotide polymorphisms, to short or large indels, to more large scale rearrangements. In practice, a read is likely to map at a locus if sequence similarity is high.
The problem when performing the read mapping is that there could be more than one genome locus to which the read can be mapped equally well. This is even more true if we tolerate imperfect matching during the mapping (and we must do it to compensate for sequencing errors and genetic variation).

Read mapping can be made easier if I have pair end sequencing, in which I sequence both ends of the genome.
In the end I get two different reads that I need to map to the genome sequence at some distance.

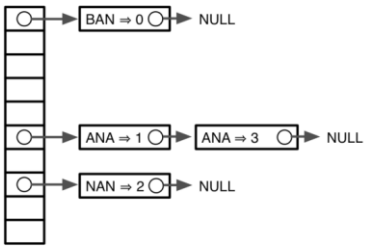
If I have reads coming from both directions, the alignment algorithm could not know where to put these reads and could suggest many positions and usually only one of them preserve the nature of the sequence. In the end I discard multiple mapping positions in order to get the fragment with the length roughly expected. Reading in both directions it's mostly done to facilitate mapping.





Suffix tree

Read mapping procedures are the same, regardless to the kind of experiment that originated the genome fragment. It's different if I sequence RNA: mRna must be retrotranscribed and then sequenced.
Sometimes the reads that I get from rna are those covering the exon-exon junction and do not map over the genome; in this case I need to process them differently. This is due to the fact that I don't need to map the entire read but just a fragment of it. Reads at the exon-exon junctions are broken during the mapping step. If the two (or more) fragments are too short, the alignment could be judged as of insufficient extension and the read would be discarded.



Seed hash tables

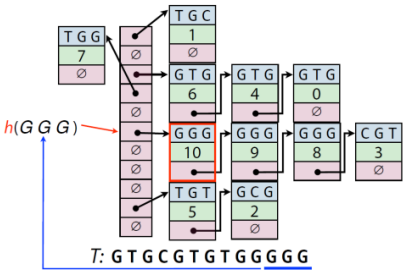
How can we perform read mapping?
There are many different ways to map reads but there are some common traits in common between them. Some of the approaches are based on Suffix trees and hash tables.

In suffix tree the time complexity is fine but the problem is space complexity since it can be quite large.

Hash Tables are also a possibility, in this case I spit the genome in k-mers and store each of them in a hash table. Then, given the sequence read, I split it in fragment of k length and perform matches. This is done in constant time but then you need to deal with collisions and the space problem.

The conversion of the genome into a data structure that facilitates the read mapping is performed following some common procedure. An important step is to create a genome index, the description of the genome sequence onto which we want to map reads. It must be loaded entirely into memory for the mapping to be executed. Once I have it I run the mapping itself.

Read mapping using hash tables.



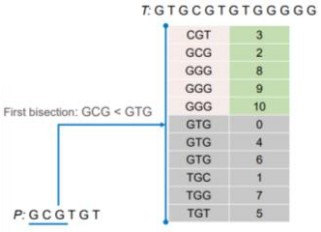
Hashing works but time complexity is uncertain. You take the genome string, decide the size of substrings (kmers) and hash each substring.
When indexing (i.e. hashing) is over, we can match a read P to the genome by splitting it into k-mers and then do a lookup into the index for each read k-mer.
For each substring I have to check if it is a match in the genome somewhere and if they are coherent(two matches of consecutive k-mers should be at consecutive positions, i.e. they extend each other).
Once a match between a read k-mer and one of the k-mers in the index is found, the characters right before and after the initial match (seed) are checked (seed-and-extend). Hashing can perform just exact matching.

An alternative strategy is binary search.

It is a way for searching in a set of strings in possibly a reasonable amount of time.
T is our genome sequence, I decide k for the k-mers. Then I put in an array the substrings and the starting positions of all of them (length: n-k+1 positions).
In this way there is no collision.

T: G T G C G T G T G G G G G

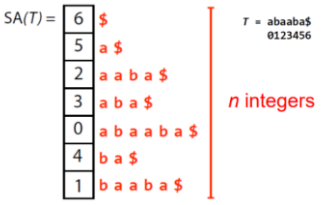
GTG	0
TGC	1
GCG	2
CGT	3
GTG	4
TGT	5
GTG	6
TGG	7
GGG	8
GGG	9
GGG	10



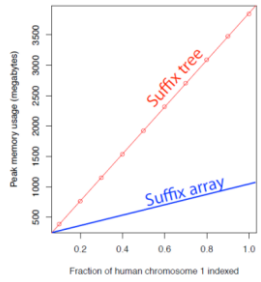
I sort the substrings in alphabetic order and divide the array into two halves. Then I check the first substring of P and if it's before the middle value of the array I search in the first half of the array, otherwise in the second half.
Then I compare the k-mer again with the middle value of one of the halves and so on until I find (or not) a match. At each cycle I perform a bisection.
I do the same for all the other substring of the query P. Also in this case I then need to check if the matches are coherent. At the end I perform the sequence alignment.
The problem of binary search is that I'm going to find if the query sequence is in the genome at the end of the binary search and so I need to perform many operations.
The time complexity in the end is O(m $\log_2$ (n)). Comparing with the hashing of the same T, this might be better or worst depending on the number of collisions in the hash table.
Binary search is often performed on suffix array.

Other data structures that can be used: **suffix array**.

A suffix array derives from a suffix tree. The latter one is quite efficient in string matching (linear time) but these are structures quite large, so space complexity is big. Suffix trees are usually converted in suffix arrays that allow binary search and occupy much less memory. These contain all the suffixes of the genome and all the starting position of these ones. Then we perform matching with the string P, with exactly the same approach as before(bisection).
If the string contains n characters then the array will contain n suffixes.

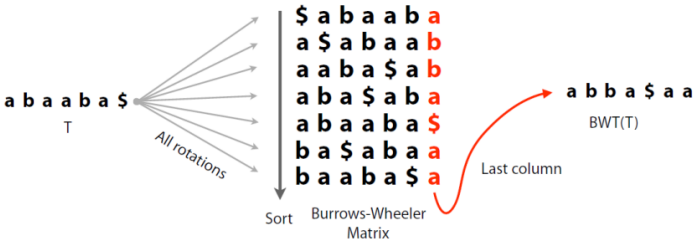


For a string T of size n, and a string P of size m, the time required to look for the string in a suffix tree is O (m), while in a suffix array is O(m log2 (n)), so suffix arrays are slower.
The index of the genome is smaller. It wouldn't be if we store the entire string for each suffix. To overcome the problem we just store the array with the offset positions after sorting all the suffix in lexicographic order



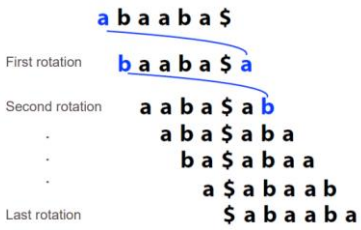
and the complete sequence of the genome. At each bisection I need to slice the genome to get the suffix corresponding to the offset position.

The most effective way to read mapping is the **Burrows-Wheeler transform**. That minimizes the required space and also time converting the genome sequence in a compact string.



The Burrows-Wheeler transform (BWT) is a reversible permutation of the characters of a string, used originally for compression. Of course it contains the same character of the input string in a different order. From the transformation I can come back to get the original string.

This transformation is obtained by computing the cyclic rotation of T, meaning that given T I take the last character and I place it at the beginning of T. I continue until I reach the string from which, if I take the last character and put it at the beginning, I obtain the starting string.



In the end I obtain a list of strings that, when sorted in lexicographic order, forms what is called the BWT matrix. If I take the last character of each string, I obtain the BWT(T).

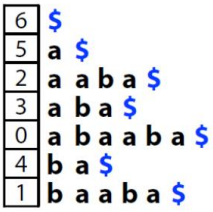
Resume:
Given a string T (i.e. a genome), with a special character signaling the end of the string (e.g. a $) and often (but not always) another special character signaling the string beginning (e.g. a ^):

5. Generate all cyclic rotations of T (e.g. if T=^ATCG$, rotations are $^ATCG, G$^ATC, CG$^AT and so on)
6. Sort all rotations in lexicographic order
7. Take the last character of each sorted rotation, to get the string BWT(T)
8. Store also the position I, in the sorted rotations list, of the original string T

The resulting BWT(T) is more compressible than T, allowing to reduce its size when loaded into memory, since identical characters tend to be more often adjacent, and the transformation is reversible, meaning that from BWT(T) we can get back T.

The BWM matrix is related to the suffix array since it's also a sort of suffix array ordered in lexicographic order.

BWM(T)



SA(T)

The advantage of having a BWT matrix is that matching strings will be much faster than binary search in SA(T).

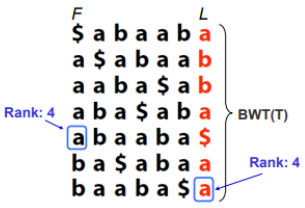
How can we reconstruct the starting string T from the BWT(T)?

The property of BWT(T) that makes it reversible is called **LF Mapping** and it's based on the correspondence of the occurrence of characters in the First and Last column.

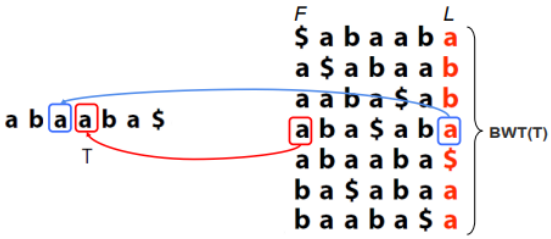
The i^{th} occurrence of a character in the Last column corresponds to the i^{th} occurrence of the same character in the First column. That means that the characters have the same rank in the last and first columns.

For example, if we take the first "a" of the string T it will have rank 4 in both columns.

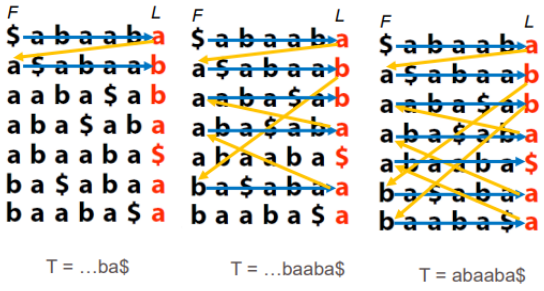
T



Another important property is that the last character of each row in the matrix (i.e. the character in L) occupies in T the position right before the first character of the same row (i.e. the character in F)



Thanks to these properties we can start to reconstruct the initial string from the end of it. The first row, in the F column, will always contains the terminator symbol ("$") and, for the LF index property, the first character of the L column will contain the second to last character. The next step is to find, in the F column, where this character is (i.e. the character having the same rank, in the example the first occurrence of "a"), and add the corresponding character of the L column (the character on the same row) to the string.
We proceed in this way, by moving from L to the corresponding character in F and from F back to L in the same row, and adding at each step another character to T, until we reach a row having $ as character in L.



In this way we don't need to store in memory all the matrix but just two strings that occupy much less memory.
But do we really need both the F and L columns to reverse BWT(T)?
Because of the LM Mapping property, and since we obviously know the number of times each character appears in T, we just need L.
Let's explain it better: in the F column, characters are disposed in lexicographic order and, since we know the number of time a character appears in T we can calculate the position at which a given character with known rank is.
An example:
Say that T has 300 As, 400 Cs, 250 Gs, and 700 Ts, and that $ < A < C < G < T
Which rows in the BWM begin with G?

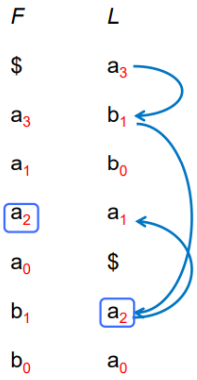
- Skip row starting with $ (1 row)
- Skip rows starting with A (300 rows)
- Skip rows starting with C (400 rows)
- Rows from (1 + 300 + 400) = 701 to (701 + 250) = 951 begins with G

Which row in the BWM corresponds to G_{100}?

- Skip row starting with $ (1 row)
- Skip rows starting with A (300 rows)
- Skip rows starting with C (400 rows)

- Skip first 100 rows starting with G
- G_{100} is at row (1 + 300 + 400 + 100) = 801

Let's say that the string T is composed by 1 **$**, 4 **a**s and 2 **b**s. The starting point is as before the first character in L, which is $\mathbf{a}_1$. This is the first **a** in L, which comes after the $ in lexicographic order, therefore, for the LF mapping rule, in F the corresponding row is the second, and the corresponding character in L is $\mathbf{b}_1$.
$\mathbf{b}_1$ is the first b in L, and in F it must come after the $ and the 4 as, therefore its corresponding column in F is the sixth, which in L leads to a_2.
a_2 is the third a in L, therefore it must come in F after the $ and 2 a characters, so its row in F is the fourth, which in L corresponds to a_1.
At the end, the string in reverse order is $a_0b_0a_1a_2b_1a_3$$, which is the initial string T.

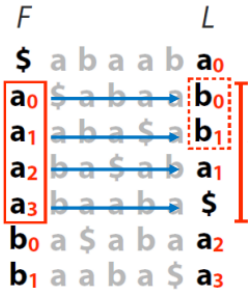


The properties of BWT(T) that allow to reverse it are the same that allow the matching of a query string P over T. With some additional steps we can find not only if P is a substring of T, but also at which offset(s) P occurs in T. The procedure is based on the construction of the so called FM-index.
This procedure is quite efficient in terms of time complexity but also of space complexity.

The FM-index is an index combining the BWT with a few auxiliary data structures. The core of the index are the columns F and L of a BWT matrix: L is stored as a string, F is represented as a string or by an array of integers. The rest of the BWT matrix, and the original string T, are not needed.

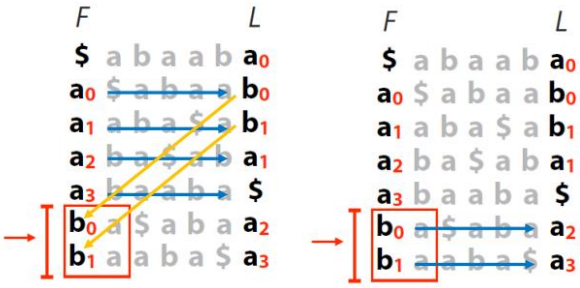
How to query a FM Index for a query string P?
Let's explain it by following an example.

$P = aba$



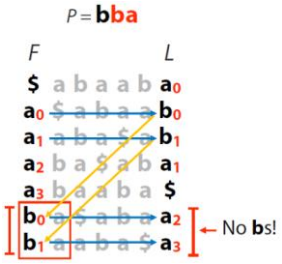
We We start from the last character of P and we identify in the F column where all the "a" characters are. We don't know yet if P is a substring of T but, if it does, then one of these "a" must correspond to the shortest suffix of P(i.e.

We go on, as in the reverse BWT, matching successively longer suffixes, until all P is matched.
So we find in F which are the rows corresponding to these selected "b" characters in L.
A new range of rows is therefore selected in F, corresponding to substrings "ba" of T. To match the next character of P, move to the end of the rows in this new range.



In this example we have two occurrences of P in T.

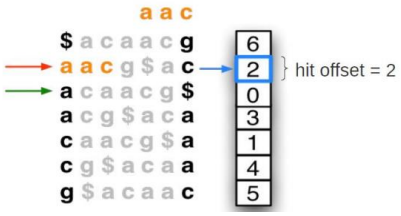
If P does not occur in T, at some point we are not going to find the next character in the L column of the index. An example:



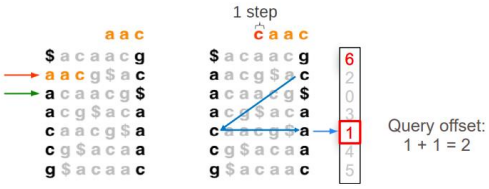
We can imagine the search as a range in the matrix rows, that at each step might change position and/or shrink until only rows having the query P as prefix are left. At each step of the procedure, the range size might remain the same or shrink, as the prefix matched by the rows delimited by the range increases by one character, moving from the last character of P to the first.

This search, in the end, tells us whether P is present in T and how many times, but, contrary to a suffix array(where I store the relative position of each suffix), it doesn't tell us where the matches happen.
Just matching two strings is not enough but I need to know where they are located.

To overcome this problem I can store into an array the position of each character in the transformed string.
The array reports the offset (i.e. the relative position) of each suffix of T starting in column F.
In this way, if I find a match, I just need to check the offset in the array.
In this case, the space needed is the same of a suffix array, since we need an array for all offsets and a string having the same length of T, but searching into a BWT(T) is faster.



In order to save some memory, an alternative way is to store not the entire offset array, but only the offset for some rows. If for example I keep track of one nucleotide each ten, then our array will be much smaller.
In this way I might not immediately know what is the starting position of a match that I find. If the matching row does not have the offset stored in the array, we can add characters to the query until a row with stored offset is reached. In other words I can move from my character backward until I find a nucleotide for which the position is stored.
The query offset is equal to the offset of the row for which offset is stored, plus the number of steps necessary to reach it.



Resume:

	Suffix tree	Suffix array	FM Index
Time: Does P occur?	$O(n)$	$O(n \log m)$	$O(n)$
Time: Count k occurrences of P	$O(n + k)$	$O(n \log m)$	$O(n)$
Time: Report k locations of P	$O(n + k)$	$O(n \log m + k)$	$O(n + k)$
Space	$O(m)$	$O(m)$	$O(m)$
Needs T?	*yes*	*yes*	*no*
Bytes per input character	>15	~4	~0.5

$m = |T|, n = |P|, k =$ # occurrences of P in T

SOFI Lesson 21 17.05.2023

RNA-Seq

High-throughput sequencing techniques can be applied to RNAs extracted from a sample, fragmented and converted to cDNA. It is not currently possible to sequence all RNAs from a sample, but only a (large) subset of them. The assumption is that this subset is representative of the population. The more a gene is transcribed, the more of its RNAs are present

in the subset, and the more will be sequenced. Hence, the number of sequencing reads from a given gene is related to its expression level.

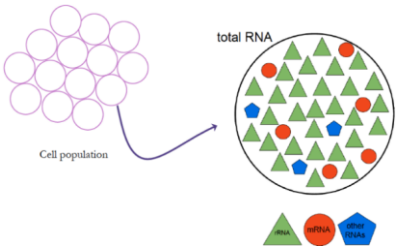
In the most common setting, two different conditions are compared (e.g. treated vs. untreaded, disease vs. healthy controls, etc.). What we are looking for are genes whose expression changes significantly in the two compared conditions. From the list of these differentially expressed genes, one can start making hypothesis on the molecular differences between the compared conditions.

Statistical testing is employed to evaluate the significance of the observed changes in expression level values for each gene expressed in the two compared conditions. If multiple conditions can be ordered (e.g. by time after treatment, increase of dosage, cell cycle stage, etc.) one can also cluster genes having a similar expression profile through the ordered conditions.

Applications:
- Gene expression levels, differential expression, identification and expression estimate of different RNA sub-classes (nuclear, cytoplasmic, poly-adenylated, small, associated to ribosomes, transcribed from a specific strand)
- Splicing variants expression levels, identification of novel splicing variants
- Identification of novel genes
- Identification of gene fusion events
- Identification of SNPs and small insertions/deletions
- Trans-splicing
- Circular RNAs
- Protein-RNA interactions
- RNA editing and methylation

Decisions to make before starting the experiment:

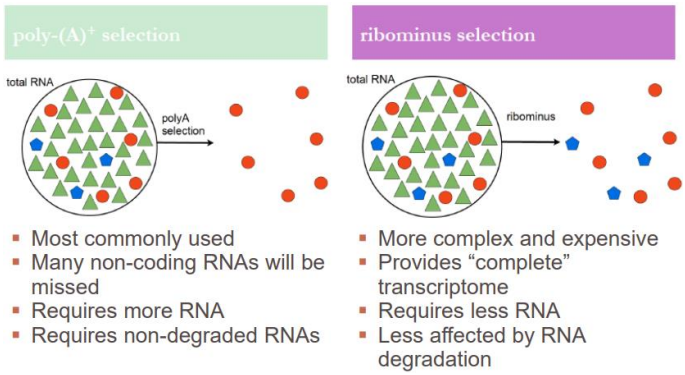
- Library type and preparation
    - RNA extraction protocol:
        - rRNA removal
        - RNA integrity check
        - Size selection
- Sequencing depth
- Number of replicates
- Control and avoidance of biases



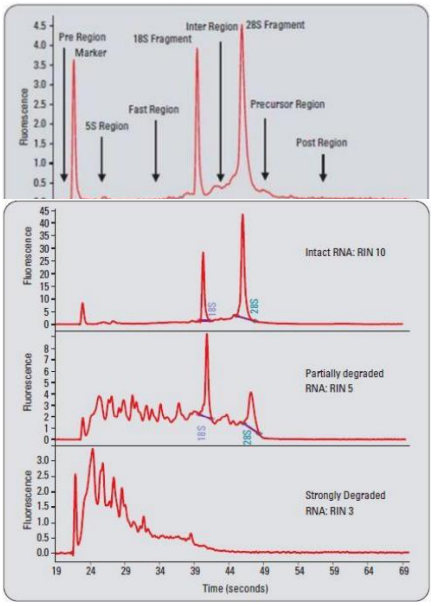
When you extract RNA from a sample, the vast majority of this RNA (about 90%) is ribosomal RNA. Usually you don't want to sequence it, because:
- the expression level of rRNA is more or less constant
- when you sequence something, the number of reads that you get is fixed. If 90% of reads that you get comes from rRNA, the number of reads representing all the other genes, the ones we are most interested in, will be very small – less accurate. Therefore we need to find a way to get rid of rRNA.

This can be done in two ways:
The most common form of rRNA depletion is **poly-A selection**, in which we select from the total RNA those molecules that have a poly-A end. RNA polymerase II directs transcription of messenger RNA genes, and those transcripts all end in a tail of consecutive adenines; since rRNA is not transcribed by RNA polymerase II, they don't have the poly-A tail. In this way, you can isolate only the transcript of RNA polymerase II and discard the others. This method is quite effective and relatively simple to set up, but the risk is to discard also other RNAs that are not transcribed by RNA polymerase II, so we do not get information about immature pre-mRNA, non-coding transcripts and all other RNA molecules not having a poly-A end.

Another way is target directly rRNA such that what is left is not only mRNA, but also other RNAs that might not be transcribed by RNA polymerase II.
This can be done through **subtractive hybridization**, which uses rRNA specific probes (short oligonucleotide having sequence complementary to some part of rRNA). Probes are built in order to bind specifically to rRNA, so that all the RNAs that are not bound to these probes can be collected.
rRNA depletion from an RNA extract can also be done using **selective degradation** of rRNAs (and of all the other 5'monophosphate RNAs) using exonucleases. The exonuclease degrades all the rRNAs, while mRNAs are protected from degradation by the 5'cap, or, in prokaryotes, by a 5' triphosphate.



RNA molecules are easily degraded, hence it is usual practice to check for RNA integrity during library preparation. RIN (RNA integrity number) is a measure of RNA degradation based on gel electrophoretic runs.



RIN is computed on the RNA extracted from the sample. Depending on the analysis of RIN, the more suited rRNA removal protocol is chosen. If RNA degradation is excessive, a new extraction should be done.

In this graph we can see an ideal profile. There are few major peaks containing rRNAs.

When you perform RNA extraction on the sample you want to sequence, you have to run an aliquot on a gel, separate the fragment by size and compare the electropholetic fluorescence profile whith the ideal profile.

The profile on top has major peaks at right position (RIN=10 : max value – completely intact, not degraded). If some RNAs were broken down into fragments, we see secondary peaks of smaller size corresponding of those fragments, while the major peaks will be smaller.

Sequencing depth (also called library size) is the number of sequenced reads for a sample

Higher sequencing depth generates more informational reads, which increases the statistical power to detect differentially expressed genes. Nevertheless, very high depths not necessarily lead to any novel insight, reaching saturation (i.e. the expression estimates do not become more accurate). There is also evidence that too large sequencing depths can result in the detection of transcriptional noise and off-target transcripts.

Given a genome or transcriptome fragment one can determine the sequence starting from:

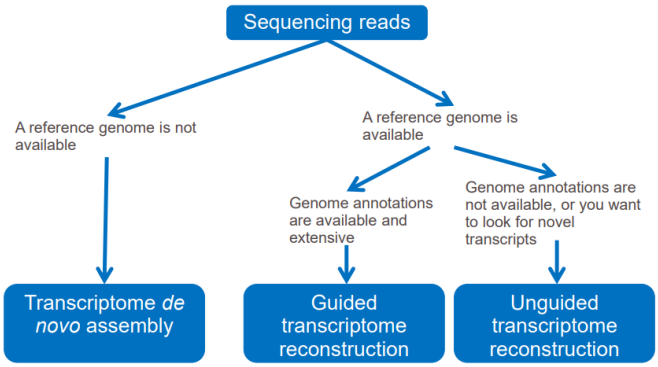
- one end of the fragment (single-end sequencing)
- both ends of the fragment (paired-end sequencing)
    - The paired reads that are generated through these protocols usually are still not long enough to cover the entire fragment sequence, but they are still useful to connect sequences at relatively long distance
    - The central part of a DNA fragment that is not sequenced is called insert. The insert size is equal to the fragment length, minus the sum of the paired reads.

The number of replicates that should be included in a RNAseq experiments depends on the amount of technical variability and the biological variability of the system under study. **Technical variability** is generally very low, and can be minimized by controlling batch effects and randomizing samples (by multiplexing, running on more lanes, etc). **Biological variability** is instead often very large, requiring the preparation and analysis of different biologically equivalent samples.

Biological replicates: different cell cultures (independents) are used, from which different libraries are prepared in parallel (all replicates are in the same condition - each gene in each replicate should be expressed in a singular way - intrinsic biological differences)

Technical replicates: one culture is used, one library is prepared then divided into aliquots (split before sequencing - expression levels should be equivalent in all the aliquots because they come from the same sample)

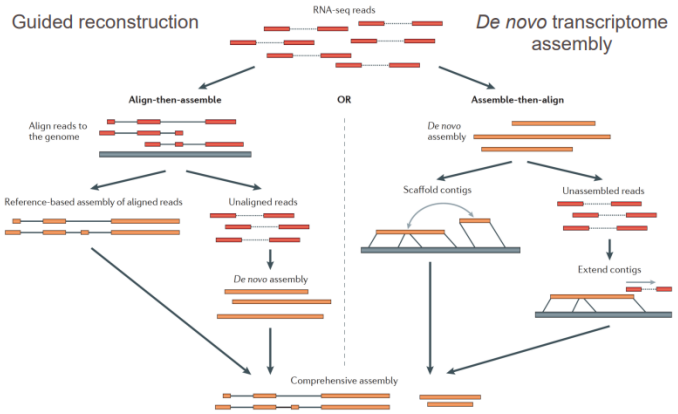
In RNA-seq biological variability >> technical variability



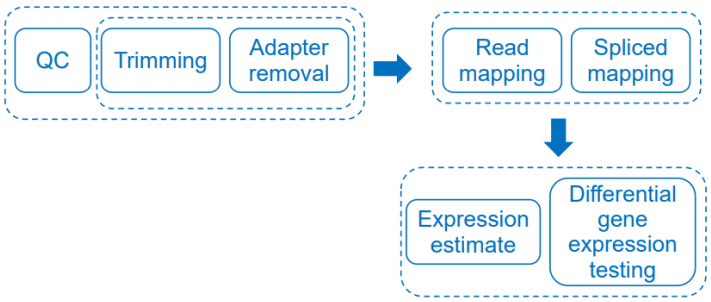
There are three ways to analyse the transcriptome, which depend on the presence of genome annotation and the reference genome itself:

- **De novo assembly of the transcriptome**: transcriptome assembly. Reads from different transcript are compared between each other, looking for shared segments that suggest provenience from the same transcript. Starting from these overlaps, reads are assembled into contiguous sequences that should correspond to while RNAs (no mapping over a genome – no reference genome available)
- **Guided reconstruction of the transcriptome**: reads are aligned onto the genomic sequence. Gene annotations (gene boundaries, their structure and known splicing variants) are used to estimate gene and splicing variants expression.
- **Unguided reconstruction of the transcriptome**: reads are aligned onto the genomic sequence, but gene boundaries, their structure and splicing variants are not known, or are not used, or are used only as a general reference. The algorithms try to infer gene structure based on read clusters on the genome, aided by spliced reads at exon-exon junctions and, when available, on paired-end reads

The three approaches are not mutually exclusive:
- if you have reliable reference genome and annotations, you always starts from a guided transcriptome reconstruction
- if annotations are incomplete, you can try also an unguided reconstruction, to see whether new genes can be found
- if the reference genome is incomplete or not well tested, you can try a de novo assembly to identify genes hosted in genome loci that are absent from the current assembly, or where the assembly is wrong.



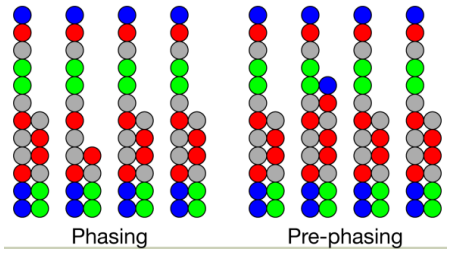
RNA-seq analysis pipeline steps:



First step is the quality control.

```
@SEQUENCE1
GCCCGGCGGGTTCATGCTGAAGAAAGGCGAAGTGTTCGGTTGGGCGGC
+
ffffffefe^eeceedffdcd^dXecffbeed`Reebe`db\]XWSS
```

Remember:
in a FASTQ file, each nucleotide has associated a quality score, called Phred score (4th line). The Phred score is determined by platform-specific algorithms, and its range can also vary.



The increase of base substitution rates when progressing through the reaction cycles is due to the loss of synchronization of the amplified fragments in a cluster (phasing).

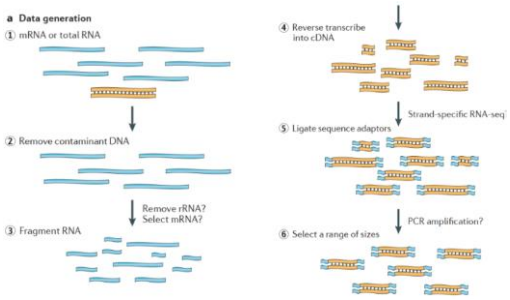
Phasing and pre-phasing are caused by: incomplete removal of the 3' terminators and fluorophores, sequences in the cluster missing an incorporation cycle, and incorporation of nucleotides without effective 3' terminators or fluorophores

The presence of sequencing errors in your read library can have two major consequences:
- A read containing too many wrong nucleotides might become impossible to map, and therefore is not going to be used for expression estimates
- A read containing too many wrong nucleotides might by chance become similar/identical to a part of the genome

which is not the locus from which the read was transcribed, therefore the read is going to be mapped to the wrong gene and used to estimate its expression level

IO Lesson 22 18/05/2023

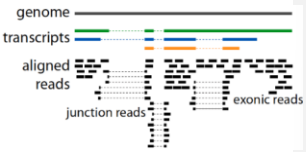
RNA-Seq analysis protocol



RNA-Seq is set up starting from the sample preparation. There are further steps which then depend on the biological questions that need to be answered.

It must include a step of conversion from RNA to dsDNA depending on the sequencing platform (most of them require this step).

Removing contaminant DNA and excessive rRNA is also required since the output of the sequencing platfom is finite and it's important not to loose informations.



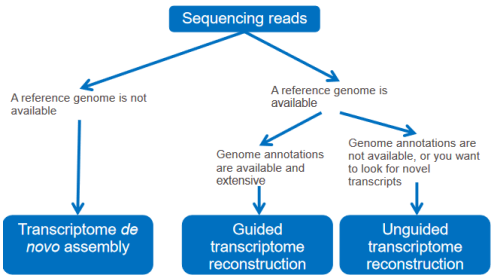
The output of the sequencing platfom is now aligned on a known genome, and the result will be segmented by the division of introns and the maturation of the RNA. This phenomenon is given by junction reads (caused by reads that differ in part of the sequence due to maturation of RNA) and exonic reads that happen in regions in which there is splicing.

Together with the output, also a **base-resolution expression profile** is considered which gives expression level estimates that need to be normalized in order to compare the data both within the sample and across sample.

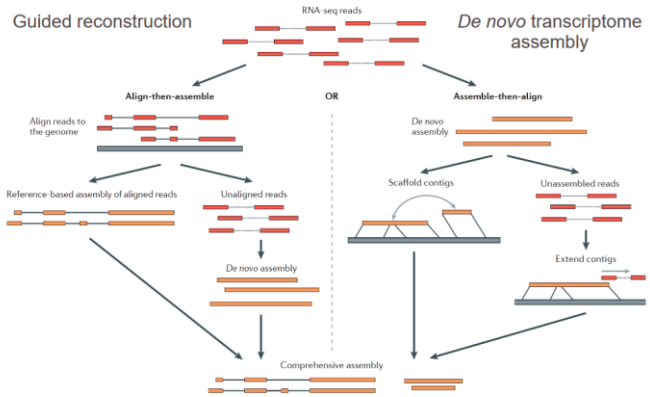
The **experimental design** consider the decisions to make before starting the experiment:

- Library type and preparation
  - RNA extraction protocol:
    - rRNA removal
    - RNA integrity check
    - Size selection
- Sequencing depth
- Number of replicates
- Control and avoidance of biases

There exist three alternative ways to analyse the transcriptome which depend on the presence of genome annotation and the reference genome itself.



Guided transcriptome reconstruction and de novo transcriptome assembly can be used together to get a better comprehension of the situation (**comprehensive assembly**).



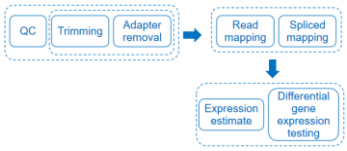
Guided reconstruction works as align-then-assemble, de novo assembly instead works as assemble-them-align. Both these procedures can find unaligned or unassembled reads due to the fact that there could be fragments of the genome that are missing or they could be wrong, uncorrectly mapped, etc.

In order to compensate for these potential sequence errors, a de novo assembly can be carried out if the starting procedure adopted was guided reconstruction, otherwise the extension of contigs might solve the issue.

Guided transcriptome reconstruction

Which means that a good genome sequence and a good set of genome annotations is available. Limit the mapping and the analysis only on reads which can be mapped onto known genome regions.

The pipeline for RNA-seq data analysis is described by this graph:



Trimming

The starting point is always FASTQ files. The file has to contain at least a number of replicated reads of 3, less than that it has to be discarded.
FASTQ file might come from single-end sequencing or pair-end sequencing (get 2 files).

When receiving the sequencing output files it is important to consider:

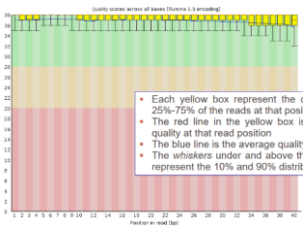
- Check read quality
- Identify the presence of general issues such as problems during the run or contamination
- Consider if it is necessary to discard entire reads, part of them and the ones that are too short, also remove adapter sequences (trimming step).

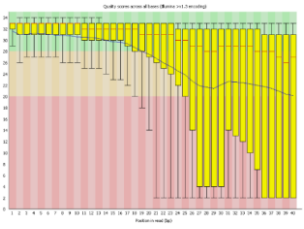
For quality control there are many bioinformatic tools that might be specialized in some particular tasks or more general, such as FastQC.

FastQC

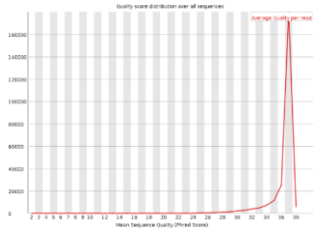
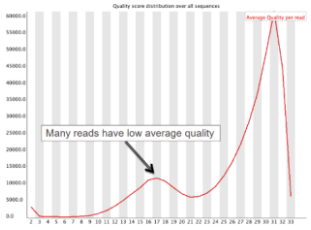
This software is able to provide several analysis tools that are needed to better comprehend the data in possess.

1. Boxplots for the distribution of quality scores position by position.
   Each yellow box represent the quality of the 25%-75% of the reads at that position
   The red line in the yellow box is the median quality at that read position
   The blue line is the average quality
   The **whiskers** under and above the yellow box represent the 10% and 90% distribution limit

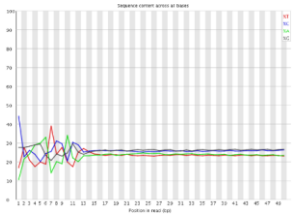
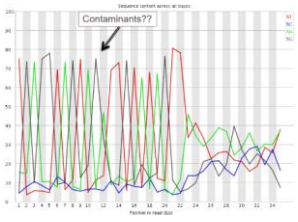




2. Distribution of the average quality of each read
   In case the read with low average quality has nucleotide reads with low quality at the end or beginning of the read, it is possible to delete only the nucleotides, otherwise it is necessary to remove the whole read.

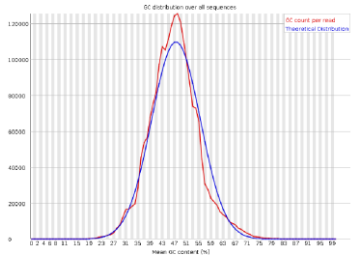



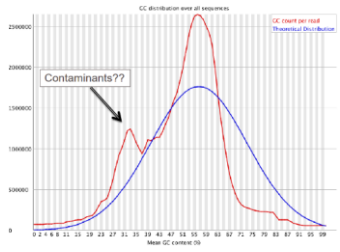
3. Nucleotide content per position
   It should always be the case that the nucleotide content percentage is more or less equal through all the positions in the reads. Otherwise there might be an overrepresentation of reads, that are related to erros in the preparation of the library: forgot to remove rRNA, presence of virus or other kinds of contaminants.

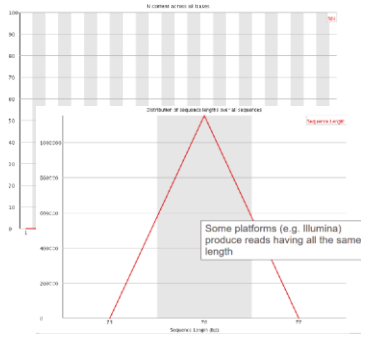
4. Plot of the distribution of GC content per read

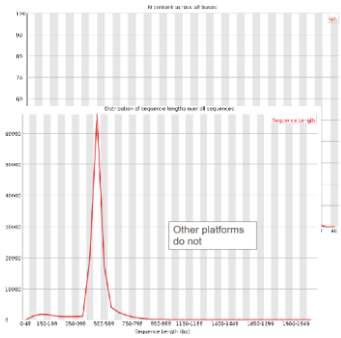
Since the GC content of a genome is known, it is an indicator for contaminants. But this might not apply to all cases since the sequencing could be done only on some specific part of the genome which have a different GC content.





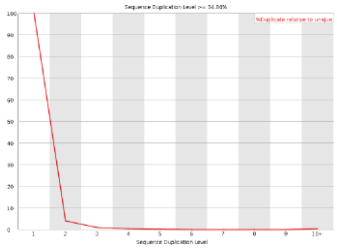
5. Frequency of undetermined bases (N) position by position
   Sometimes during a cycle the sequencer might not be able to determine the nucleotide. There's no way to guess which was the nucleotide had been lost, but it's good to know that thiskind of error happened.

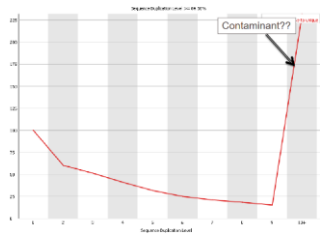




6. Read length distribution
   Not particularly useduf in illumina sequencing since all the reads have the same length but it might be different for other platforms.

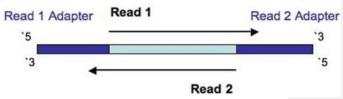
7. Read duplication level
   Identical reads can be the outcome of contaminants or of amplification artifacts, resulting from copies of the same template that were produced in two or more distinct clusters.
   In general, in quantitative analyses, these artifacts might lead to overestimation of signals, therefore identical reads should be removed (keep only one copy).
   Nevertheless, in RNA-seq, if one gene is expressed at very high levels, many identical copies of its transcripts are present in the extract, and since RNAs are relatively short molecules, they might be fragmented in a similar way,





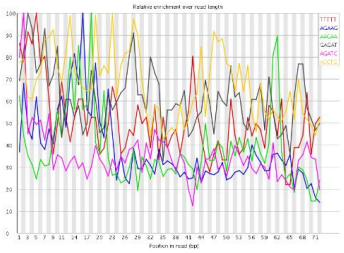
leading to identical reads even in the absence of amplification artifacts. Hence, in RNA-seq, there is no need to remove duplicates.

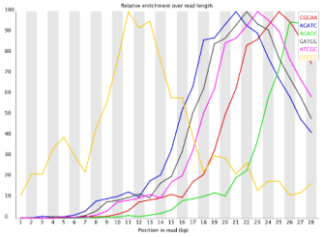
The software checks the presence of substrings of reads occurring with high frequency and reports them as overrepresented sequences. It can also checks if the sequence represents something (just BLASTing it), usually it is rRNA.



Other forms of overrepresented sequences could be adapters sequences coming from library preparation. Adapters are sequenced when the read of interest is shorter than the length that the sequencing machine is able to synthesize in one read.

8. Read k-mer content by position
   Might reveal some characteristic patterns. It is not necessary to be considered for RNA-seq.





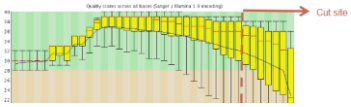
Trimming

Based on the quality control results, you could decide to clean up the dataset to facilitate and make more accurate the following steps of the analysis. You can:

- Discard read ends having low quality
- Discard reads that after this step remain too short to have an unambiguous alignment
- Discard whole reads having low average quality
- Remove adapter sequences

All these steps are performed using trimming algorithms

Trimming can be done in easier and also more complex ways:



- **Static trimming**
  Cutting all reads to/from the same position
  The problem is that there could be good quality nucleotides also after the cut site that get lost.
- **Dynamic trimming**
  Cut the 5' end until quality of the nt is below a threshold and do the same from the other side.
  In this way some reads will be shorter than others in your dataset, but it is not a problem.
- **Windowed Dynamic trimming**
  Cut the last nt when the average quality of the window in consideration is below the threshold, and then shift the window of one nt. Same thing starting from the 5' end.
  This avoid cases in which one nucleotide with sufficient quality is dispersed among nucleotides with poor quality. Trimming can be done in windows of fixed size based on the window average quality score.

Trimmomatic is a tool used for trimming that has good functionalities.

After the trimming step, it is necessary to check again the quality. It is also important to consider then the read lengths, if after the trimming there are reads shorter than a certain length it is necessary to remove them.

The output of files after trimming depends on the software that had been used but generally as FASTQ files are input, they're also the same format as output.
In single-end sequencing as one file is input, a file is output. But in paired-end sequencing there is both forward and reverse as input which have the same length, but after trimming the length could be different and also some reads can be discarded. This means that it could be output only one file because all the reads of the other file did not survive and had been deleted.
Another solution could be using softwares that create four files in which there are 2 of unpaired reads also in order not to loose informations.

Read Mapping

In this step is important to increase the possibility that a read is mapping correctly which is essential for a quantitative analysis like with RNA-seq data. Together with read mapping it could be necessary to also consider the **Spliced mapping** in order to rescue all the junction reads to increase coverage and also allowing to reconstruct which kind of alternative splicing variant the gene is encoding. This last information is necessary to understand the exact functional impact of the transcriptome.

It is possible to perform mapping with various software, it is therefore necessary to check recent publications about them to find the one more suitable for each case. i.e., input data, hardware requirements, ease of installation and use, quality of result, documentation, user community, maintenance and how easy it is to plug into a pipeline.

The most commonly used tools today are based on FM-indexing, but recently, since memory cost is getting cheaper, other versions of suffix trees are getting back in use.
FM-indexing tools that are going to be described in this course are Bowtie/Bowtie2 and TopHat/TopHat2 which are part of a pipeline called tuxedo suite; in the last part of the pipeline are Cufflinks (gene expression estimate) and Cuffdiff (differential gene expression estimate). It is a relatively old pipeline, the author of this pipeline have promoted a new version of it using HISAT2 instead of Bowtie and TopHat which is still similar anyway.

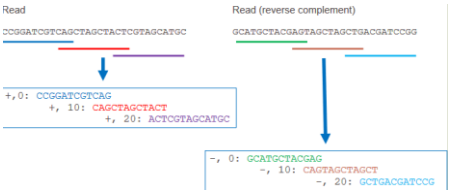
Bowtie2

It is not particularly accurate, but it is fast and can be used for Illumina reads.

The algorithm implements a FM-Index through a Burrows-Wheeler transform (BWT), to index and compress the genome. After a BWT the result is a string with the same character but in different order to facilitate the mapping. When performing a matching of a query string which can be a sequencing read over the genome index, the result is whether or not the query is a subset of the index. Usually, it's important to know not only if it is a substring but also the position of it but the FM-Index doesn't provide this index automatically. It is therefore necessary to store, for each nt in the transformed genome string, the position also in the original string forming an array of positions.
To save more space it is not needed to store the offset of the position of each nt in a transformed string but just the position of some nts at fixed interval, i.e., Bowtie takes one nt every 36 by default.

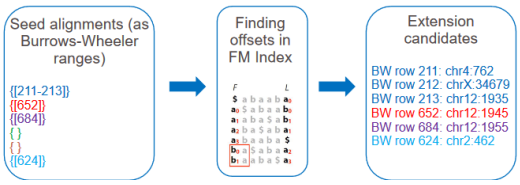
Resuming, the matching is done by starting from seq of each chromosome through which BWT is carried out in order to retrieve an index array with offset 36 nts, then with the FASTQ file (raw or trimmed) each read is (possibly) assigned to a position which is reported in the standard SAM or BAM format.

It is important to take in consideration that usually algorithms are designed on optimal datasets and when run with real life data they do not work as expected. When mapping onto a known genome, there could be mismatches between the reads and the genome itself since these data are coming from two different organisms that are most likely not the same. These differences can involve on average 1% of nts, so in order to include the possibility of not exact matches still maintaining a low complexity algorithm, the read is split into overlapping fragments (interval between fragments can be set).



The problem in this situation is that the index of a genome is computed over one strand of the genome but reads can come from ither strand. It is not possible to know which of the two strands align to the genome sequence. Therefore, the solution is to take one read, split it into fragments, generate the reverse complement of the read and split also that one. In this way there are all the possible substrings (rev. complementing and not reverse complementing) are generated.

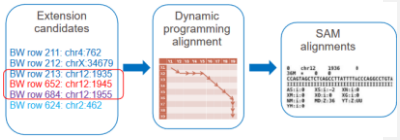
Then each one of these fragments a standard matching using FM-Indexing is done. For some of them it is possible to find match positions on chromosomes locations.



In this case, the first fragment has been located in 3 different chromosomes, and the other three only to one chromosome.

The next step that Bowtie goes through is checking the coherence of the matches.
Since in this case the fragments found consecutive matches in chromosome 12 and only the first fragment found matches in chromosome 4 and X, it is safe to assume that the original location of the first fragment is more likely to be together with the rest of them.

Furthermore, Bowtie is able to align these fragments found together. It is a necessary step when it was not possible to match each substring of the read, which might mean that some of them contained errors. The result of the alignment is output in a SAM/BAM format file.



The alignment is important because together with it, it comes its score, which is able to better give understanding of the alignment.

Bowtie have a modified scoring scheme in which the mismatch penalty can be scaled based on the quality score. i.e., if there's a mismatch in a nt with low quality score, it is possible to assume that a sequencing error occurred and reduce the mismatch penalty.

The software does different alignment for different mapping positions and then gives the user the possibility to chose the preferred one.

SOFI Lesson 23 19.05.2023

Since we are performing an alignment, we will obtain a score, which depends on a scoring function.

It might include match/mismatch scores, gap penalties ecc. In Bowtie these scores can also be weighted by the quality score of each nucleotide in the read (you can give less weight a mismatch for example because it could be wrong).

So, for each one of the multiple positions on which the read can be mapped, we get a score; the one having the higher alignment score will be our mapping position for the read.

Presets

In end-to-end mode :
- `--very-fast -D 5 -R 1 -N 0 -L 22 -i S,0,2.50`
- `--fast -D 10 -R 2 -N 0 -L 22 -i S,0,2.50`
- `--sensitive -D 15 -R 2 -N 0 -L 22 -i S,1,1.15` (default)
- `--very-sensitive -D 20 -R 3 -N 0 -L 20 -i S,1,0.50`

In local mode :
- `--very-fast-local -D 5 -R 1 -N 0 -L 25 -i S,1,2.00`
- `--fast-local -D 10 -R 2 -N 0 -L 22 -i S,1,1.75`
- `--sensitive-local -D 15 -R 2 -N 0 -L 20 -i S,1,0.75` (default)
- `--very-sensitive-local -D 20 -R 3 -N 0 -L 20 -i S,1,0.50`

--end-to-end: align the entire read from one end to the other

--local: some characters may be trimmed from the ends in order to achieve the greatest possible alignment score

-L <int>: length of the seed

-i <func>: interval between extracted seeds

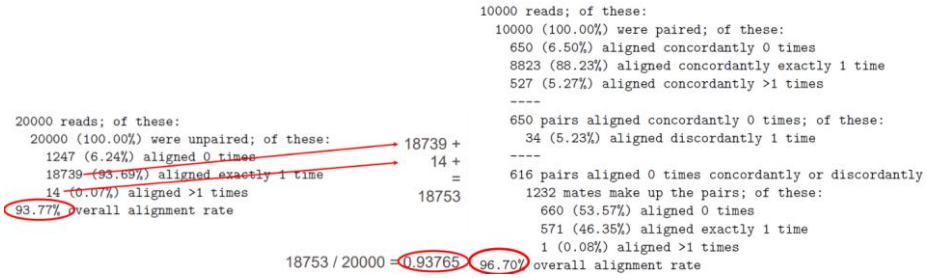
There are also parameters for multi-reads:

Default: Bowtie2 returns one good alignment – no guarantee that this alignment is the best possible in terms of alignment score

-k <int>: specify how many alignments to return

-a: return all of the found alignment – very slow

The output of a Bowtie is a SAM/BAM file, plus a short summary of the mapping

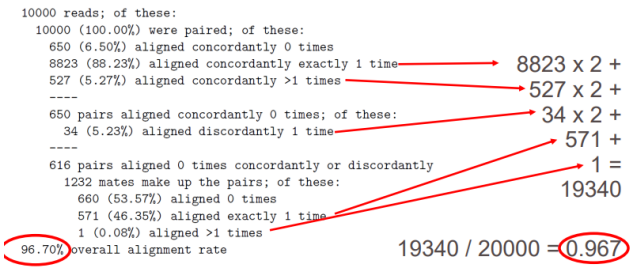


Concordant: pair of reads coming from a paired-end sequencing which alignes over the genome at the right distance and orientation. The pair aligns with the expected relative mate orientation and within the expected range of distances between mates

Discordant: Both mates have good alignments, but the alignments do not match paired-end expectations (distance, orientation)

Neither concordant nor discordant: Only one member of the pair is aligned

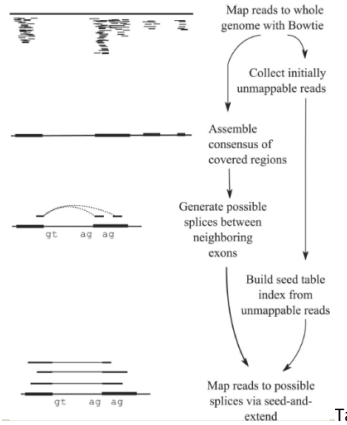
The alignment score for a paired-end alignment equals the sum of the alignment scores of the two mates



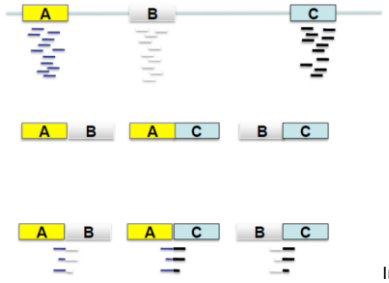
TopHat

Once Bowtie has completed its run, what you have is a BAM/SAM file containing the mapping position of each read. Also, there are included reads for which no position in the genome is reported. This will be the input for the TopHat.

TopHat /TopHat2 utilizes either Bowtie or Bowtie2 to align reads in a splice-aware manner and aids the discovery of new splice junctions.



Take all the reads that Bowtie was not able to match and try to match these reads over all possible exon-exon junctions. At first, it tries to reconstruct all possible ways in which the exons of one gene can be combined. TopHat seeks "islands" in which reads accumulate on the genome. Then they seek reads that can join different islands, which corresponds to the exon-exon junctions. This way, it is possible to reconstruct the exon-intron structure of a gene and infer the splicing variants expressed by the gene in the analysed sample.



In the example we have 3 exons, that you can combine in those different ways (possibility of internal splicing site not included - coverage more or less uniform). Once you have joined the 3 exons in all possible combinations you just generate the sequence corresponding to the junction and then you align over these sequences all the reads that Bowtie didn't map.

This is the output of TopHat

```
tophat_out> ll
total 922M
-rw-r--r-- 1 mlove 619M Apr 25 19:36 accepted_hits.bam
-rw-r--r-- 1 mlove  204 Apr 25 19:32 align_summary.txt
-rw-r--r-- 1 mlove 430K Apr 25 19:32 deletions.bed
-rw-r--r-- 1 mlove 371K Apr 25 19:32 insertions.bed
-rw-r--r-- 1 mlove 6.4M Apr 25 19:32 junctions.bed
drwxr-sr-x 2 mlove   47 Apr 25 19:32 logs
-rw-r--r-- 1 mlove   70 Apr 25 12:11 prep_reads.info
-rw-r--r-- 1 mlove 296M Apr 25 19:37 unmapped.bam
```

There's a bam file with the accepted reads and one with the unmapped ones. Then we have bed files reporting the junctions/deletion...

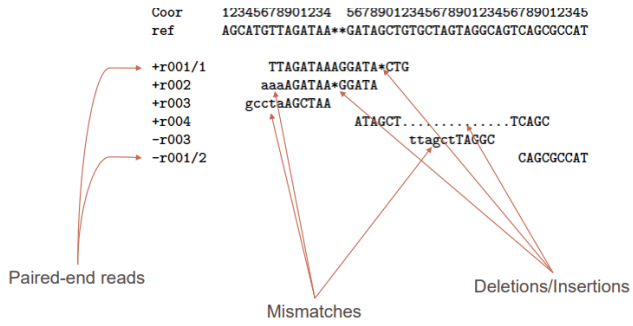
- unmapped.bam (BAM): all unmapped reads, in a BAM file with no alignment coordinates and description
- accepted_hits.bam (BAM): all successfully mapped reads
- junctions.bed (BED): junctions used to map spliced reads. Each junction consists of two connected genome blocks
- deletions.bed (BED): reports all deletions in the reference genome with respect to the reads
- insertions.bed (BED): reports all insertions in the reference genome with respect to the reads

The most important output of is the mapping. Since mapping can be very large and complex, you need a standard and compact way to represent it.

Commented [Ss2]: BED (Browser Extensible Data) format: generic format for genome annotations

Three mandatory fields:
1. chrom – name of chromosome or scaffold.
2. chromStart – Annotation start position
3. chromEnd – Annotation end position

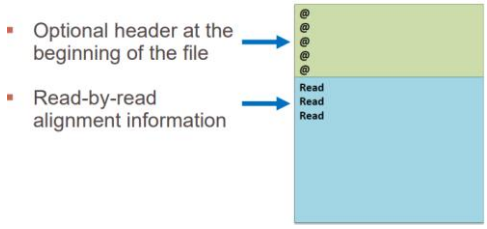
Optional fields:
4. Name
5. Score – a value between 0 and 1000
6. Strand
7. thickStart – This and the following two fields are used for visual representation in Genome Browsers 8. thickEnd
9. itemRgb



Is very important to have standard format in genomics. Genome-wide data (sequences, annotations, mapping, variations) are reported and exchanged in standard file formats, to facilitate their spread and bioinformatics analysis. No matter which software you are using, the input/output formats are generally one the following:

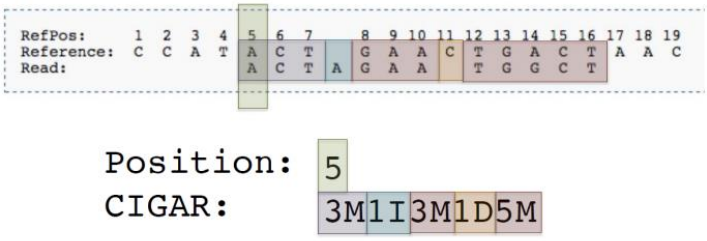
- Sequences:
  - FASTA (just the sequence)
  - FASTQ (sequence and Phred quality scores)
- Alignments of reads to a genome:
  - SAM, BAM (binary version of the SAM format)
- Genetic variants (SNPs, insertion, deletions):
  - VCF
- Annotations:
  - BED (coordinates and description of general genome features)
  - GFF, GTF (coordinates and description of genes)
- Genome multiple alignments:
  - MAF

As we said, the output of read mapping tool is always a BAM/SAM file.



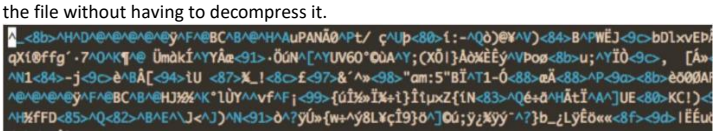
SAM is a plain text format, tab-delimited, useful for reporting alignment (and info for the alignment) in a compact way. The file starts with an optional header, which starts with a @. It contains metadata, e.g. what was mapped, to what it was mapped to, how it was mapped, plus the version of the used SAM format, whether or not the file is sorted, and, in case, how it was sorted, which is the reference sequence, the pre-processing steps done on the reads library, which software was employed for the mapping... then you have the alignment for each read. divided in type separated fields (like offset, mapping quality, CIGAR string, mate reference name, position of mate mapping, observed pair distance, read sequence, read quality scores encoding, and other optional information).

CIGAR encoding is a way to describe the algorithm between two strings in a compact way.



Op	BAM	Description
M	0	alignment match (can be a sequence match or mismatch)
I	1	insertion to the reference
D	2	deletion from the reference
N	3	skipped region from the reference
S	4	soft clipping (clipped sequences present in SEQ)
H	5	hard clipping (clipped sequences NOT present in SEQ)
P	6	padding (silent deletion from padded reference)
=	7	sequence match
X	8	sequence mismatch

BAM file is the binary version of the SAM format, up to 75% smaller than the corresponding SAM. You can lookup into the file without having to decompress it.



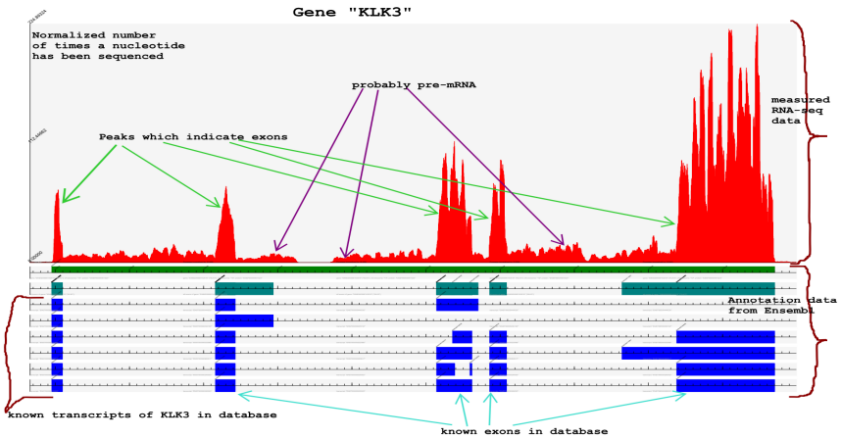
You can map read over genome or transcriptome:

- Genome:
  - Requires a decent genome sequence
  - Requires spliced alignments
  - Can find novel (previously un-annotated) genes
  - Can find novel exons/isoforms
- Transcriptome:
  - Sometimes the genome is unknown, but there are cDNA libraries
  - Spliced alignment is not necessary
  - Multireads are an issue
  - Cannot find what is unknown

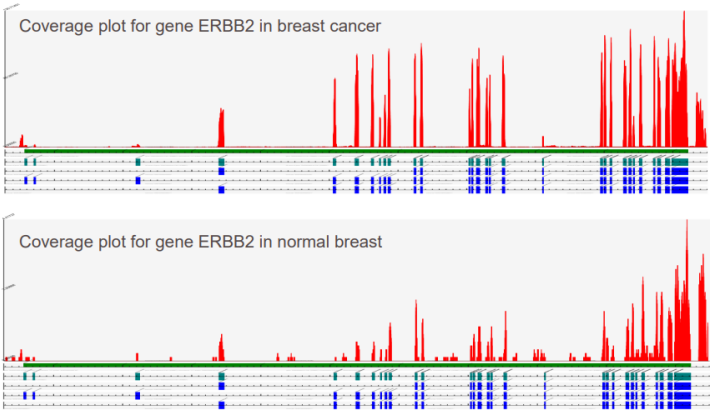
Multi-reads: reads that can be mapped in different places
They can originate: by chance (especially if the read is short), due to paralogs sharing exons/domains (when mapping to the genome), due to splicing variants sharing an exon or part of it (when mapping to the transcriptome)

Then, what you want to do is estimate how much genes are expressed. Sometimes this step is not done.

It is done from the distribution of reads over a gene. The histogram reports the coverage of a gene.



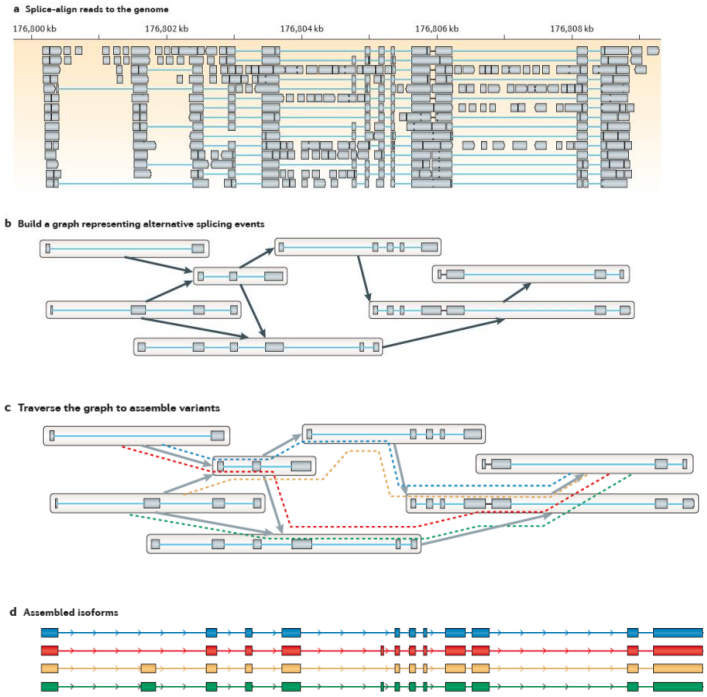
Consider all these peaks and convert them in numbers that represent how much that gene is expressed (to allow comparison).



This is done by a lot of algorithms. We will look at Cufflinks.

Cufflinks tries to estimate the expression of a gene by the expression of individual transcript that the gene might encode. Estimate gene expression at transcript level can be useful to know expression level for a transcript, for a splicing-variant, and not just an overall estimation of a gene (that can be obtained by summing al the transcript).

A graph representation of the splicing is built. It can be based on previous knowledge or on our data. This graph reports different ways in which exons can be connected. Each path from the starting node to the terminal node corresponds to a different splicing variant.



Once this graph is built, Cufflinks tries to assign reads, mapped over the exons of the gene, to a different path, to a different splicing variant. Then it finds a set of paths with a different weight, that corresponds to the number of reads which were assigned to that particular splicing variant.
One assumption that Cufflinks does is that it tries to find the simplest explanation of the distribution, it tries to explain the splicing in the simplest way.

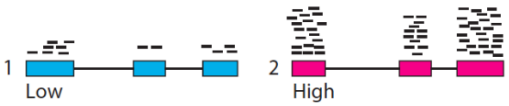
SOFI Lesson 24 22.05.2023

Regardless which is the method for building transcript and assigning reads to each individual transcript, the overall estimation of gene expression at the gene or transcript level is computed in a quite simple way.

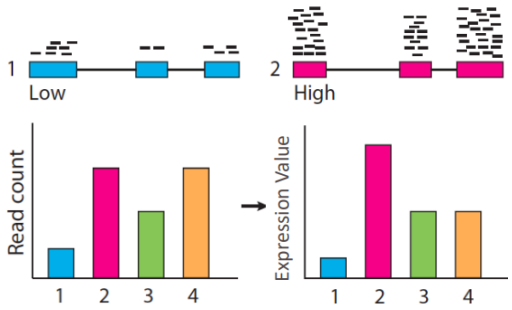
Usually, a normalization of data is done, which allows the comparison of the gene expression level within the same sample and across samples.

Normalization is based on few simple considerations.

How much a gene is expressed is related to the number of reads you can map over the gene exons. In the example we have two genes: from counting the reads mapped on their exons, gene 1 seems less expressed than gene 2.

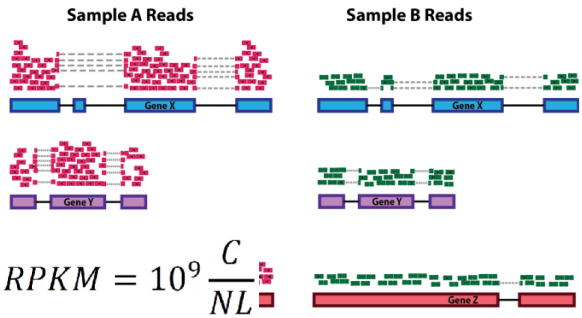


However, read count is not good estimator of gene expression. Let's consider gene 3 and 4: gene 3 has less reads than gene 4, so it might seem less expressed. But gene 4 has longer transcripts, which means that more fragments are



generated and so more reads. Therefore, even if genes 3 and 4 are expressed at similar levels (producing a similar amount of RNAs), gene 4 has a higher read count

For this reason, counting the reads is not probably the best solution. We need to normalize read counts by gene length.
In this way we can clearly see that genes 3 and 4 are expressed at similar level even if gene 4 has an higher number of reads.



Another form of normalization is based on the sequencing depth. In the example, each gene appears to have doubled in expression in Sample A relative to Sample B, however this is a consequence of Sample A having double the sequencing depth. Each sequencing run can provide a different number of reads.

$$RPKM = 10^9 \frac{C}{NL}$$

RPKM: **R**eads **P**er **K**ilobase of exon model per **M**illion mapped reads.
Formula used for estimation:

C = number of reads mapped on the exons of a gene
N = total number of reads produced by the sequencing
L = Total length of the exons of a gene

FPKM: **F**ragments **P**er **K**ilobase of exon model per **M**illion mapped fragments
Another formula for estimation
the real estimator of gene expression here are fragments, not how many reads you get from them.
If you sequence using single-end seq from each fragment you get one read, if you use paired-end seq from each fragment you get two reads. You must count fragments

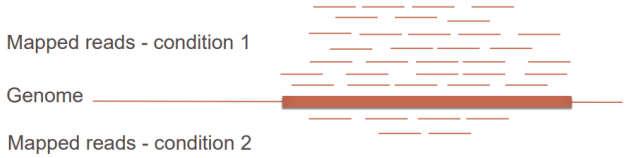
$$FPKM = 10^9 \frac{C}{NL}$$

C = number of paired reads mapped on the exons of a gene
N = total number of paired reads produced by the sequencing
L = Total length of the exons of a gene

Even normalizing by library size, the sum of all RPKMs of all genes detected in a sample might not be the same in all analysed samples. To overcome this, one can perform the two normalizations (by length and by library size) separately. First, one can divide the reads count C of a gene by its length, obtaining C'. Then, instead of dividing by the total number of reads, one can divide by the sum all C' values. Then one multiplies this value by 109 to obtain the so-called TPM (transcripts per million).

Ranking genes by expression level on a sample might not tell you that much over the nature of the sample and cannot be used directly for a comparison across different samples. You can compare the expression level of a gene in one condition and the same gene in different conditions, but the difference might explain some biological differences. To

be sure that the gene is truly changed in expression forced by some biological reason which can explain the biological differences in the two samples that you are comparing, you need to do some statistical tests.



Is the observed difference statistically significant?

If so, which are the biological causes and meaning?

A simple way to report expression differences is computing the ration between the expression in one condition and the expression in the other: this is called **fold change**, usually reported as:

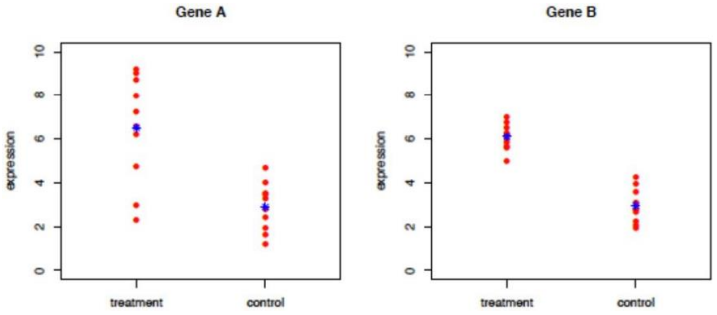
$$\text{Fold Change} = \log_2 \frac{\text{Expression in condition 1}}{\text{Expression in condition 2}}$$

Which could be a good threshold for considering a difference in expression as significant?
This threshold should be the same for all genes or not?
It is based on expression means or medians, and variance is not considered
The fold change is not a statistical test

A test for differential expression measures the probability that observed expression differences between two conditions are due to a biological difference and not just to random fluctuations. Fluctuations can be due to measure errors, technical variation due to sample preparation and/or analysis biological variation not associated to the differences between the two conditions.

Generally, the expected variance is estimated, calibrated on the experimental replicates (when available), and compared to the observed variance based on a probabilistic model of the read mapping.

Some methods employ normalized expression levels expressed in RPKM or TPM, but, more commonly, the read counts are used. These counts are always normalized in a way or another, not by the counts of all genes in a condition, but rather on all counts for the same gene in all analysed samples.
A common assumption is that gene expression is not globally changing between the two compared conditions, meaning that the number of truly differentially expressed genes is small compared to the total.



Each red dot is a different replicate for that condition. The blue dots are the means.

Now we want to compare distribution.
We want to test:
- Is the expression of one gene measured in a number of experimental replicates different in condition A and
condition B?
- Is the mean expression of a gene in condition A different from its mean expression B?
- If the mean is different, with which confidence we can say that this difference is not due by chance only?

Assumptions:
- The distribution of expression values of a gene measured in the experimental replicates is normal
- All measures are independent variables

Distribution is represented through the mean and the variance/standard deviation.

Not enough comparing means.

Simplest way to compare normal distribution is the T-test

$$t = \frac{\text{Difference of the means}}{\text{Variance}}$$

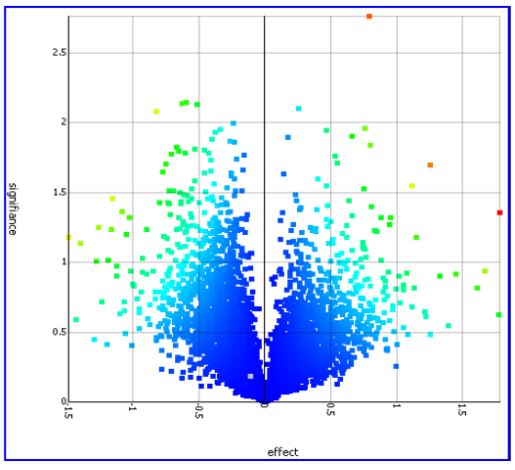
$$t = \frac{\bar{X}_T - \bar{X}_C}{\sqrt{\frac{var_T}{n_T} + \frac{var_C}{n_C}}}$$

However, T-test is rarely employed for RNA-seq data. The following tests are more often employed for differential expression testing in RNA-seq experiments:
- Z-test
- Wald test
- Fisher test
- Wilcoxon non-parametric (cannot be applied on non-normal distribution)

All these are in some ways based on the comparison of means and variances. The probability is computed and then used to reject or accept some hypothesis.

When differential expression is tested directly on normalized values, one can use parametric or nonparametric tests. When read counts are used, the distribution that seems to be the better in modelling the experimental variability is the **negative binomial** (NB), in which the variance is not just computed over distribution of all the expression measure of one gene in all the replicates of one condition, but also using the distribution level of all the sample, looking at all the other genes, that allow to estimate the expression measure from the mean using a dispersion parameter Φ. Dispersion can be estimated from the data, assuming that most genes do not change expression in the two compared conditions. Variance is calibrated on all the genes in the examined samples! Negative binomial distribution is often used for RNA-seq data (NB). NB model is a good approximation for data where the mean < variance, as is the case with RNA-Seq count data.

Hypothesis testing methods:
For each gene:
- The null hypothesis (H0) is that the gene has no expression variation in the two conditions
- The alternative hypothesis (Ha) is that there is an expression change
- A test is employed to estimate the probability that the null hypothesis is true (p-value)
- If the p-value < some threshold, one can reject H0 and accept Ha



For each gene expressed in the samples we are comparing what you get a distribution of expression level from which you can compute the fold change and then a statistical test.

A volcano plot is a scatterplot of - log(p-value) vs. the fold change.

Comparison of the significance (p-value – if this effect is different, do we need to consider it significant or not?) with the effect (fold change – how much does the gene change in the expression?).

Each dot is a different gene

One must choose the maximum p-value for rejecting the null hypothesis. Usually, genes are considered differentially expressed if p-value < 0.05. This value

corresponds to a risk factor of 5%, which is frequency of rejecting the null hypothesis H0 even if it is true (i.e. the gene is not differentially expressed).

	Reality: H_0 is false	Reality: H_0 is true
Decision: Reject H_0	H_0 correctly rejected (True Positive)	False Positive (type I error)
Decision: Do not reject H_0	False Negative (type II error)	H_0 correctly not rejected (True Negative)

If p-value is too low, we might be discarding many genes. We are considering as negative genes which instead should be positive (false negatives). If the p-value threshold is too high, then the risk is that we are considering as positive genes that are not differentially expressed (false positives). Therefore, you have to set a threshold which balances these kinds of errors.

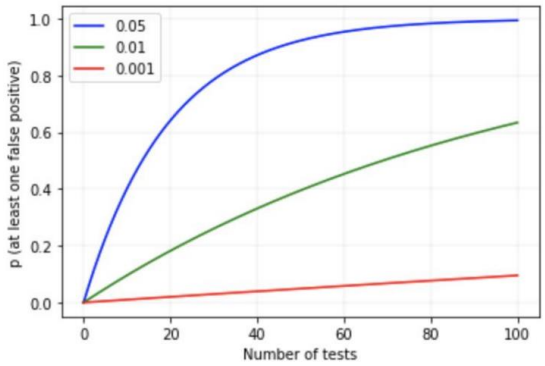
At the end of the differential expression testing procedure, to each gene a p-value is assigned, that is the probability of rejecting the null hypothesis of having no differences between the two samples for that gene. In simpler terms, **p-value can be considered as the probability that the detected differences in expression values are only due to chance and technical factors. The smaller is the p-value, the more convincing is that there is a significant expression change between the two conditions**. For each gene, a different test must be performed, raising the issue of multiple testing.

Each gene requires a different test. In an RNA-seq experiment, you might perform tens of thousands of tests. If the p-value threshold is 0.05, and you are testing 20,000 genes (then 20,000 tests), you can get in the end a possibly large number (risk factor: 5%) of false positives (type I errors), that are genes that you consider differentially expressed even if they are not. These false positives can affect the biological characterization of the differences between the two samples.
There are correction methods (for example the methods of Bonferroni, Benjamini-Hochberg, Sidak, Duncan, Holm)

If you are testing a null hypothesis that is true, using a threshold of significance α then the probability of having a correct non-significant result is 1-α.
If you are testing 2 independent hypotheses (e.g. two different genes) the probability that neither test is significant is the product of the two probabilities (1-α)(1- α). In general, for k independent tests (= k genes), the probability that all tests are not significant is (1-α)^k.
Hence, the probability of having at least one significant test by chance (the negation of the previous case, where all were not significant) is 1-(1-α)^k. This probability is called Family Wise Error Rate (FWER).

Most correction methods are based on the assumption that the global error associated to the experiment is function of the p-value of each individual test. If the significance threshold α is lowered, also the global error is reduced. Methods based on FWER control seek for new thresholds (or, alternatively, they seek corrections of the p-values) in order to reduce false positives. This usually comes at the cost of having an increase of the false negatives.

Commented [Ss3]: For example, if you are testing 20 hypotheses (20 genes) using a significance threshold α = 0.05, then the probability that none of them is significant is: (1-α)^k = (1-0.05)^20 = 0.9520 = 0.36
Therefore, the probability that at least one test is significant by chance (a false positive) is: 1 - 0.36 = 0.64
This probability is quite larger that the threshold value α, hence we need correction methods, especially when the number of tests is high



The most simple and used method is the Bonferroni correction. Imagine that you are testing 5 hypotheses (e.g. five genes), and you want the FWER to be at most 5%. Then, type I error for each individual test must be set to 0.05/5=0.01, or, alternatively, p-values must be multiplied by 5. These corrected p-values are called adjusted p-values (or q-values). Bonferroni correction is very conservative, and when the number of tests increases, thresholds become so small that no test is significant anymore. This way, false positives are effectively reduced, at the cost of increasing the false negatives, that are tests for which the null hypothesis is truly false but that do not result as significant by the test. Hence, when testing a large number of genes such as in a RNA-Seq experiment, other methods, such as those limiting the False Discovery Rate (FDR) are preferred, such as the Benjamini-Hochberg method.

The DESeq2 method is considered one of the most accurate for differential expression testing. Each gene is represented by the number of reads (counts) that can be mapped over its exons in the compared samples.

IO Lesson 25 23.05.2023

Functional Characterization

At this point the sequencing data had been trimmed, mapped and differential genes expression levels had been estimated.

The output at this moment is something like this table.

It's a list of gene and for each one there is:

- fold change
  amout and direction of expression level. If x > 1 means that there's an increase of expression level.
- q (adjust p value)
  Reduce the number of false positive without increasing the numeber of false negative.

How to test for differential gene expression: identifying genes that compared to different conditions have changes in the expression levels.

What you get in the end is a list of genes which are differentially expressed and that can be quite large.
It is possible to start understanding the situation of the sample, to reveal whether a sample reacted to a the change in the environment that is testing for.
Anyway it is difficult to control all the genes with that table and some other image and tables might come to help.

Let's take in consideration an example:

Imagine that the comparison of two conditions led to the discovery of 100 differentially expressed genes, and for simplicity let's imagine that each one of these genes is associated to only one general function.
What is possible to understand about the phenotypic differences between the two?

It is necessary to categorize genes based on functional features such that the problem of interpreting a large list of genes is simplified.

Functional Category	Number of genes
Synaptic transmission	40
Metabolism	20
Transcription	20
Energy production	10
Immune response	5
Protein transport	5
TOTAL	**100**

If, in the list of differentially expressed genes, there is a large number of genes with the same function, participating to the same process, pathway, network, etc., then it is likely that perturbation to that process are responsible for the differences between the two compared conditions. Therefore, to understand what a gene list is explaining about a biological system, we need to test for **functional enrichment**, which means that we need to test whether there is a larger number of genes having similar function than one would expect by chance. The gene list must be compared to a background gene set, which might be the whole proteome, to verify whether a given functional annotation is present more (or less) than expected from a random sampling of the background set.
One test that is often used for this kind of analyses is the **hypergeometric distribution test**.

To perform functional enrichment, a standardized way to describe biological systems with a numerical code or keyword for each function and process is necessary, so that a gene is clearly labeled univocally. Also accurate mapping to associate functions with genes and a statistical method to evaluate whether a particular functional characteristic is found more than expected in the gene list.

Gene Ontology has been created as a tool for the unification of biology, by The Gene Ontology Consortium.

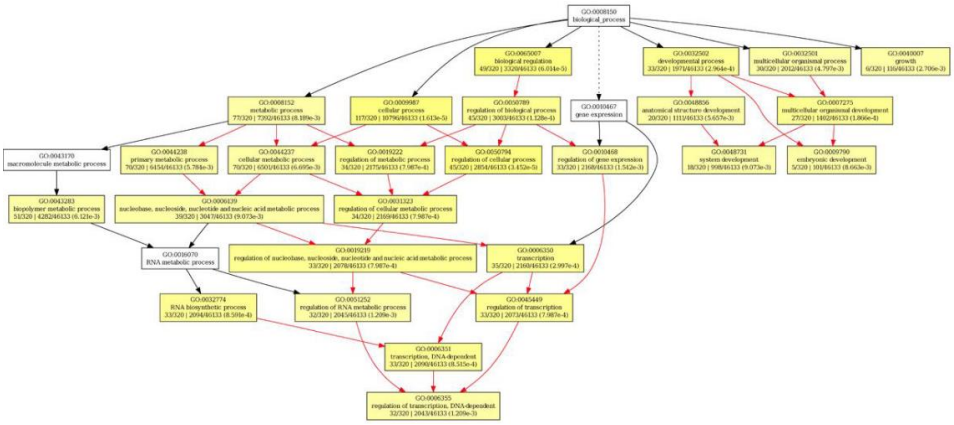
Ontology: is a formal representation of a series of concepts and the relations among them. The aim is to get a stable way to annotate the functions of gene products in an organism and between different organisms.

The three controlled vocabularies are used:

- The biological processes in which they participate
- Their molecular functions
- Their subcellular or extracellular localization

This means that homologue genes from different organisms are labeled in the same way.

Based on these three terms, trees of genes are created. Starting from a root label which is the biological process or molecular function or cellular component.



For each label there exist a page with information about it. It is also possible to move layer by layer to better analyze the tree.

Therefore, now synaptic transmissions are identified with an univocal code.

- GO ID: GO:0007268
- GO term: synaptic transmission
- Ontology: biological process
- Definition: The process of communication from a neuron to a target (neuron, muscle, or secretory cell) across a synapse

The small colored labels describe where the connection between the two labels is formed.

It is important to notice that there are different paths that can lead from the root to a particular node due to redundancy in definitions.

In order to get more descriptive relationships in the graph, it is possible to use colors with specific legends. Colors are also used to identify the organisms from which a lable has been retrieved.

At this point it is necessary to associate a term and a gene product and can be done based on experimental and computational evidences.

Experimental Evidence Codes

- EXP: Inferred from Experiment
- IDA: Inferred from Direct Assay
- IPI: Inferred from Physical Interaction
- IMP: Inferred from Mutant Phenotype
- IGI: Inferred from Genetic Interaction
- IEP: Inferred from Expression Pattern

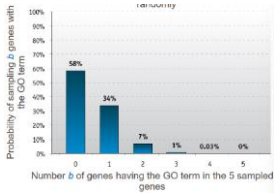
Computational Analysis Evidence Codes

- ISS: Inferred from Sequence or Structural Similarity
- ISO: Inferred from Sequence Orthology
- ISA: Inferred from Sequence Alignment
- ISM: Inferred from Sequence Model
- IGC: Inferred from Genomic Context
- RCA: inferred from Reviewed Computational Analysis

From UniProt it is possible to retrieve all the information about gene's GO.

Let's take for example this exercise to better understand the application of functional enrichment and the use of the hypergeometric distribution test:

Given a genome with 100 (N) genes, of which 10 (B) are annotated with some specific GO term and 90 (N-B) not annotated with GO term, let's sample 5 (n) genes randomly.



The probability of finding exactly b genes annotated with one particular GO term in the n sampled genes is given by the hypergeometric function:

$$HG(N,B,n,b) = \frac{\binom{n}{b}\binom{N-n}{B-b}}{\binom{N}{B}}$$

The probability of finding at least b annotated genes is:

$$HGT(N,B,n,b) = \sum_{i=b}^{\min(n,B)} HG(N,B,n,i)$$

In general, we have a list of N genes in the genome, of which B genes can be associated to a given function described by a GO term, and N-B genes that do not have that function.
Imagine that, among the n differentially expressed genes detected in an RNA-Seq experiment, b genes are associated to that GO term, while n-b genes are not. If this number b is significantly higher than expected, then this means that the gene sampling method (that is the selection of differentially expressed genes among all tested genes) is not just a random sampling but we enriched the gene list in genes having that function.

The test is repeated for each GO term that is associated to at least one gene in the gene set. What you get is a list of GO terms that are significantly enriched in the analyzed gene set, which is very helpful in understanding the phenotypic differences in the two conditions that were compared. Since this is done by applying hypothesis testing methods, and the number of GO terms to be tested can be large, usually multiple testing correction methods need to applied here as well.

DAVID is a free remote service in which is able to perform comprehensive functional analysis of large gene lists which do not all need to have the same label but there can be different genes with different labels (more flexible).

As an alternative to GO terms, it is possible to use pathways. KEGG is the software which stores metabolic pathways; substrates and products are represented as circles and enzymes as rectangles.
KEGG contains also signal transduction and regulatory pathways. (BioCarta is a similar software that is specialized in transduction pathways)

When testing for pathway enrichment, what is checked is whether in the differentially expressed genes there are more genes belonging to a given pathway than expected by random sampling.
If this enrichment is significant, the expression of the pathway is considered to be different in the two compared conditions.
We can also test for enrichment of other types of gene features, like the involvement in diseases, binding of ligands or drugs, genome coordinates, expression in cell types or tissues, and many others.

Transcriptome de novo assembly

When performing an RNA-seq analysis and a genome is not available, it is possible to reconstruct transcripts by doing something similar to assembly algorithms (but for transcripts).
It is possible to do transcriptome de novo assembly quantitavely too but it does not work very well.

It could be also the case that even having a reference genome, de novo assembly is performed in parallel to combine the tools for a better understanding of the transcriptome.

The problem of transcriptome assembly is further complicated by the fact that you are not trying to assemble only one sequence, but each gene is a unit that must be treated separately, and one must try to identify which reads originates from the same gene and then assemble them.
Moreover, a gene can encode for a number of different splicing variants, therefore from the same group of reads one must be able to identify all possible reconstructions, each corresponding to a different splicing variant.

The De Bruijn graph for transcriptome de novo assembly will be ideally one for each RNA. Therefore, they're interpretation is more complicated.
Branching paths can represent different splicing variants encoded by the gene now, so the assembled transcripts resulting from the semplification of the graph will be many considering alternative splicing.

A tool useful for this operation is Trinity. Starting from RNA-seq reads it splits them into k-mers, then build the de-Bruijn graphs and finally outputs the transcripts in a FASTA file.

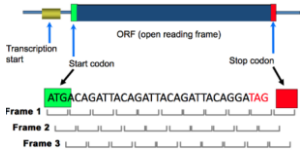
Genome Annotation

Genome annotation has the aim of associating to each nucleotide one or more roles. In its more common form, the major goal is the identification of genes and their accessory sequences.

The annotation process is much more complex in eukaryotes than in prokaryotes, given the more complex eukaryotic gene structure, and the fact that the relative proportion of coding regions in the eukaryotic genome is much smaller.

Since prokaryotic genes are not interrupted by introns, the problem of prokaryotic genome annotation can be reduced to finding ORFs (Open Reading Frames) of sufficient length, and for which a set of known regulatory signals can be found. A prokaryotic genome can be scanned in all the 6 possible reading frames (3 for each strand), each of them then translated into hypothetical protein sequences, following the genetic code rules.

Reading frames are delimited by the start codon and one of the three stop codons, and if the distance between start and stop is significantly longer than what can be expected by chance, the reading frame is called "open".



Since the genetic code contains 3 stop codons (TAA, TAG, TGA) out of the 64 total codons, the probability of finding a stop codon in the same reading frame of a start codon ATG is about 1 over 21. The expected length of an ORF in a random sequence is therefore around 20 amino acids. Real protein sequences are generally much longer than that, therefore, all identified ORFs in a genome can be evaluated for being significantly longer than what is expected by chance, or more simply just by setting a length threshold (e.g. 50 aa).
A possible complication is that often in prokaryotic genomes a single transcriptional unit can contain more than one ORF and lead to the synthesis of more than one protein (these cases are called polycistronic genes or operons).

To check whether the protein found is a real one, it is intuitively to run it on BLAST and see whether it exists or not.

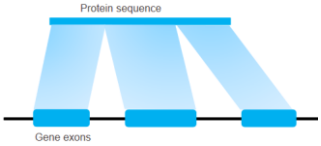
It is also possible to add more features to the ORF, like looking for a promoter and TF binding sites, this procedures is useful to further validate the presence of the ORF.

Since only a relatively small fraction of the proteins encoded in a genome is truly species-specific (i.e. it is encoded only by one or a small set of related species) most real ORFs should encode for proteins that can be found similar in other species.

Given the discontinuous nature of **eukaryotic genes**, and the large genome size, it is not possible to just look for ORFs. The simplest and more effective way for annotating eukaryotic genomes is to align and map full length or partial sequences of known transcripts and proteins on the genome, to identify their gene of origin. When this is not possible, because not enough data are available to annotate the genome of a species, *ab initio preditions* can be used.

It is also a problem the low percentage of Exons that are found in the eukaryotic genome, about 1.5%.

cDNA and protein sequences found in the databases can be used to annotate genomes. If they're from the same species as the target genome (e.g. human proteins aligned to the human genome), **cis alignment** can be performed which require an almost perfect string matching.
Alternatively, it is possible to map on a closely related genome (**trans alignment**), in this case string matching is tolerant to differences.

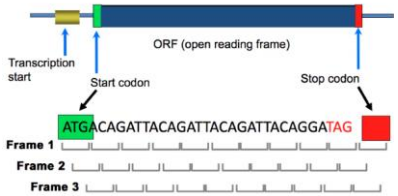


A genome annotation tool that is able to perform alignment is **BLAT** – BLAST-Like Alignemtn Tool.
This tool is able to consider splicing and align with gaps quite effectively.

When performing a trans alignment of an eukaryotic gene, it is also possible to form a sort of **model**, which is not specific on what actually is encoded but on the series of signals that occur in a specific order and at the correct distance.

Emi Lesson 26 24.05.2023

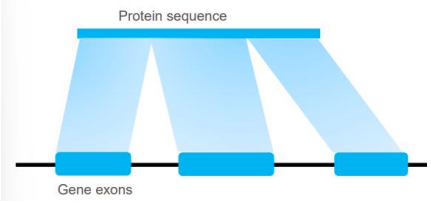
Obtaining a genome sequence is only the first step towards the understanding of the underlying biology processes. Genome annotation is the association of a function to a genome sequence, possibly to each nucleotide in the sequence, and to identify genes and regulatory sequences that may play a role in regulation of gene expression. Genome annotation differs in eukaryotes and prokaryotes since they have different kind of organization of their genome. Prokaryotic genome is compacted and smaller and moreover genes are not interrupted by introns.
The problem in prokaryotes can be reconducted to the problem of finding open reading frames.



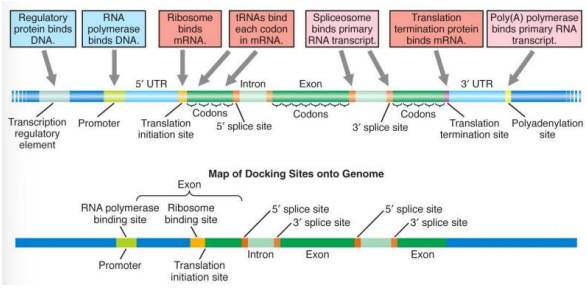
What we do is to scan the genome of all possible strings of length three for both strands. Every time we find a triplet that can be associated to a start codon, we need to search for the closest stop codon and compute the probability of having a start and a stop codon with that specific length. If the distance between start and stop is significantly longer than what can be expected by chance (20 nt), the reading frame is called “open”.
An ORF is a hypothetical gene that can be transcribed and then translated.
When the length of the ORF is enough long we can search for other specific sequences that can validate our hypothesis. These might be sequences characteristic of promoters, or related to the translation (for example the Shine-Dalgarno sequence, for ribosome binding, which is found similar in most prokaryotes and at short distance from the start codon).
If you are convinced that the ORF could be a gene encoding for product, we can take the product of this sequence and, using some tools as BLAST, verify if this could be a real protein (so if we can find some other similar proteins or potential homologous). Since only a relatively small fraction of the proteins encoded in a genome is truly species-specific (i.e. it is encoded only by one or a small set of related species) most real ORFs should encode for proteins that can be found similar in other species.

For eukaryotic genome is much more difficult. The coding part of the genome is very small compared to the rest of it and it’s not possible to just look for ORFs.
One way to annotate the genome is to use sequences that are already known and try to map them on the genome. If protein/cDNA sequences are from the same species as the target genome (e.g. human proteins aligned to the human genome), this is called cis alignment. A perfect or almost perfect string matching is required.
Alternatively, one can try mapping on a genome the available protein/cDNA sequence from a closely related species (e.g. human proteins aligned to the gorilla genome), and this is called trans alignment. String matching in these cases must be more tolerant to differences.

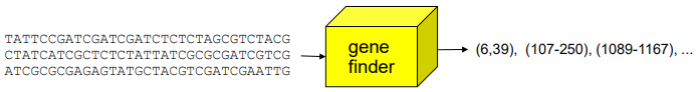
In order to do it there are some specific tools for example BLAT (BLAST like algorithm). BLAT can align protein and RNA/cDNA sequences to a genome, taking into account that the query sequence is the result of splicing events, therefore composed by subsequences that need to be aligned separately, respecting the order and at a reasonable distance.
This is the most accurate way to annotate eukaryotic genome.



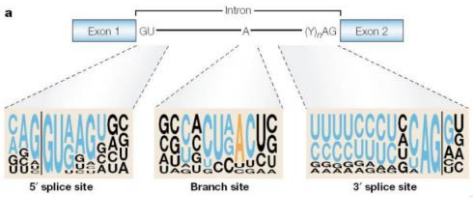
Unfortunately, this is not always possible for lack of proteins and cDNA sequence databases.
In this case we can create models of genes and then scan the genome with this gene models. These must be independent on what are the products of transcription and translation events.
This is possible because genes have standard organization with some signal that are common to all of them and that must occur in a specific order and at the correct distance.

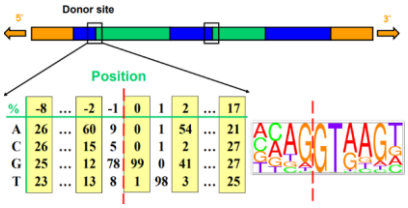


Once we have built this model, with all the signals and features that are found in genes, we can use a gene finder (some algorithm) that, given a genome sequence, can get a prediction on whatever or not this genome could be a gene (with a score and a probability) and how this gene can be organized in the form of a series of ranges. This is called the ab initio genome annotation.



Some of the signals have some characteristic sequences that are almost the same in all the genes.
From the picture we can observe the frequency of the nucleotides reported that is proportional to the height of the letter. For example we can observe that there will always be a sequence “GU” at the end of the 5′ splice site.

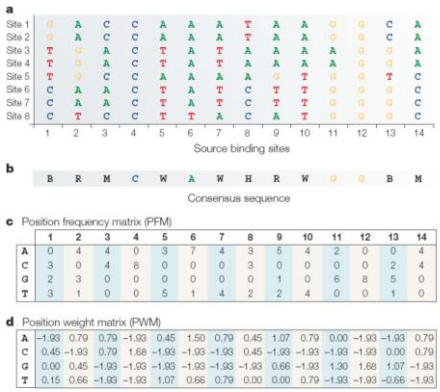




This regularity can be expressed in a numeric form. What we need to do is to collect a set of representative sequences(gene already known) to compute the frequencies position by position. Some features may change species by species, but also they are more or less conserved so we don’t need to already know which gene we are going to annotate to compute this frequency matrix.

These matrices can be manipulated in order to report them in a way that allows for the scanning of the genome and testing of the genome string to see if it can be a match of the features that we are trying to model.
One way could be to count the absolute frequencies and then divide it by the number of sequences to get the relative frequencies that can be approximate to probability.

We can also convert them in different ways, obtaining the Position Frequency Matrix(PFM) and the position weight matrix(PWM).
These matrix can be computed in different ways but we are going to analyse just one of them.



If we start from a set of examples of one genome feature we can calculate the probability to get a nucleotide for each position.
This is done following the formula:

Corrected probability calculation: $$p(b,i) = \frac{f_{b,i} + s(b)}{N + \sum_{b' \in \{A,C,G,T\}} s(b')}$$

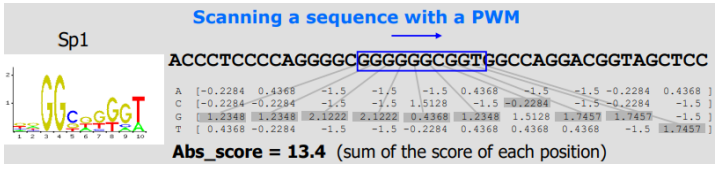
$f_{b,i}$ = counts of base b in position i; N = number of sites;
$s(b)$ = pseudocount function
$p(b,i)$ = corrected probability of base b in position i

In the numerator we need to add a pseudocount function, i.e. a value, to avoid situation in which the count of a given nucleotide is 0. This situation could lead to problems in the next steps.
We can report this probability in a Position Frequency Matrix(PFM).

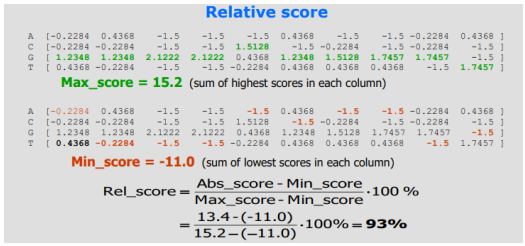
PWM conversion: $$W_{b,i} = \log_2 \frac{p(b,i)}{p(b)}$$

$p(b)$ = background probability of base b;
$p(b,i)$ = corrected probability of base b in position i;
$W_{b,i}$ = PWM vaue of base b in position i

Commonly these probability are converted in a score in the way reported.
This is done because genomes have different contents in nucleotides meaning that some of them could have abundance of a given nucleotide. So if we take a string randomly it might have a good score just because it contains lot's of that nucleotides and not because there is a real match of the feature under study. To minimize it we can get a score(W) given by the ratio of the probability for a position divided by the background probability, taken as the logarithm. These scores are reported in the position weight matrix(PWM).
The matrix is built for a given genome feature so I can then use it to understand if a piece of the genome meet this feature.
Given this matrix if I want to score a piece of the genome I need to fetch to this matrix the corresponding score and sum them. This means that I need to take a piece of genome as big as my matrix and then fetch is nucleotide to the score in the matrix. Then I can move the window selected to scan the other part of the genome(usually shifted by one nucleotide).



Usually we can establish a threshold over which we accept the hypothesis. It's difficult to set this trheshold so we usually compare it with the maximum score that I can get form the matrix.



The closer it's my score to the maximum score the more I'm convinced that this may be truly the feature I try to model.
I can also compute the minimum value I the opposite way, to understand the worst scenario. We can then understand how my score is placed between these values.

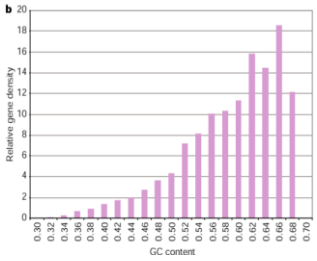
Nevertheless, not all signals in the genome can be described as specific sequence patterns encoded by

a weighted matrix. Some other signals are more elusive, and are defined by specific sequence compositions that cannot be represented by a simple sequence pattern.
For example, exons and introns have sequence characteristics that might differentiate them, but to capture these differences we need more complex models. Also the extension (i.e. the sequence length) of these genome features might follow some rules (described by length distributions) that can help in finding them.

I can take introns from genes already annotate and try to understand what they have in common but that's not easy. For example exons are not the same but four different kinds of exons can be defined:

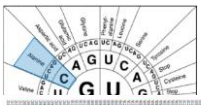
- Initial exons, from the transcription start site to the first splicing donor site;
- Internal exons, from a splicing acceptor site to the next splicing donor site;
- Terminal exons, from the last splicing acceptor site to the polyadenylation site;
- Single exons, from the transcription start site to the polyadenylation site, in mono-exonic genes (genes with no introns)

Let's say that we want to get a length distribution of exons. We need to keep separate the different kinds of exons because the length distributions will be different, with different average, different tails etc..

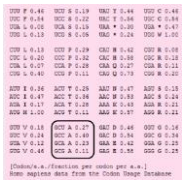


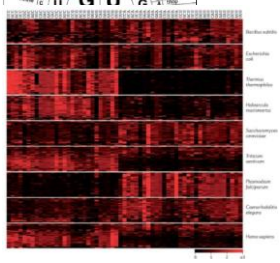
Genes are not uniformed distributed along the genome and the probability to find them in a particular place may depend on some features of the genome. Genes are more frequently found in parts of the genome having a specific CG content. This is a plot of gene found at given GC content.

For example a feature that we can find in intons is the length of genes that depends, for unknown reasons, on the GC content. When the GC content increases, the length of introns decreases. For exons this feature doesn't apply.
Exon sequences are under strong evolutionary pressure that shapes their sequence composition, while introns might evolve much more freely. One characteristic is that not all the codons, encoding for the same amino acid, are present with the same frequency.



In the example we can see that some codons encoding for alanine are more frequent than other.GCC will be more frequent than the other. This is called the codon usage bias.





The codon usage bias is specie's specific.
Codon usage:
▪ Not all codons are used with the same frequency within a species
▪ Different species can have different codon preferences
▪ Different genes within the same genome can use a different set of codons for the same amino acid

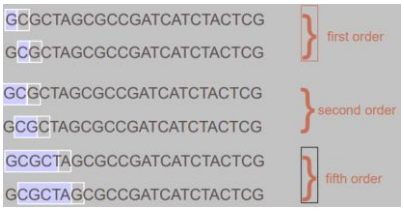
To summarize, while some genomic signals can be defined and identified by means of specific sequence patterns, other signals are more complex and at the same time more difficult to define.
All sequences are shaped by some evolutionary pressure, but the effect can be different depending on the role of the sequence.
To identify these less defined sequence characteristics we need a better way to model sequences, the most commonly employed is by means of **Markov chains** and **hidden Markov models**.

A Markov chain, given a sequence of events, estimates the probability of having a particular event in a particular position depending on the events coming before in the sequence.
So given a sequence S (of numbers, characters, etc.): $S = [s_1, s_2, s_3, \ldots, s_n]$
it is a **Markov chain** of order k if the probability of having an element x in the i^{th} position of S depends on the k

elements right before x
▪ if k=0, each element is independent
▪ if k=1, each element depends on the one right before it, and the probabilities can be modeled by counting frequencies of dimers (dinucleotides or dipeptides)
▪ and so on for higher values of k

How can we use that?

We can model sequences based on the dependence there is between characters in sequence. We can calculate the probability to find a given nucleotide, given that the previous k character are know.



Markov chain rule depends on some probability concepts.

Conditional Probability: probability of A, given that B occurred → $P(A \mid B) = P(A, B)/P(B)$

Independent Events: A and B are independent if knowing the outcome of B gives no information on the outcome of A → $P(A \cap B) = P(A)\,P(B)$

Markov chain rule: $P(A, B, C) = P(A)P(B \mid A)\,P(C \mid A, B)$

A Markov chain is a model for the modelling and stochastic generation of sequential events
The order of the Markov chain indicates the dependencies, i.e. on how many previous events a given event is dependent on. In the case of a biological sequence, the order indicates the number of upstream characters on which the current position depends.

$$s = s_1 s_2 s_3 s_4 \cdots \longrightarrow s = ttacggt$$

0th-order

$$P_0(s) = \mathrm{p}(s_1)\cdot \mathrm{p}(s_2)\cdot \mathrm{p}(s_3)\cdots = \prod_{i=1}^{N} \mathrm{p}(s_i)$$

$$P_0(s) = \mathrm{p}(t)\cdot \mathrm{p}(t)\cdot \mathrm{p}(a)\cdot \mathrm{p}(c)\cdot \mathrm{p}(g)\cdots = \prod_{i=1}^{N} \mathrm{p}(s_i)$$

1st-order

$$P_1(s) = \mathrm{p}(s_1)\cdot \mathrm{p}(s_2 \mid s_1)\cdot \mathrm{p}(s_3 \mid s_2)\cdots = \mathrm{p}(s_1)\cdot \prod_{i=2}^{N} \mathrm{p}(s_i \mid s_{i-1})$$

$$P_1(s) = \mathrm{p}(t)\cdot \mathrm{p}(t \mid t)\cdot \mathrm{p}(a \mid t)\cdot \mathrm{p}(c \mid a)\cdots = \mathrm{p}(s_1)\cdot \prod_{i=2}^{N} \mathrm{p}(s_i \mid s_{i-1})$$

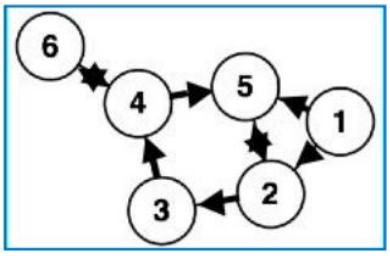
In a 0th-order chain, each character is independent and the probability to get that given sequence is the product of the probabilities of each nucleotide.

In a 1st-order chain, each character depends only on the previous one, and so on.

A first order Markov chain can also be described as an oriented graph D = (V, A), where V is the set of vertices (*states*) and A is the set of oriented edges. Each node in V corresponds to an element in the sequence of events S, and edges connect consecutive elements in S.

Each conditional probability can be used to describe the edges linking the nodes.

$$s = s_1 s_2 s_3 s_4 \cdots \longrightarrow s = ttacggt$$

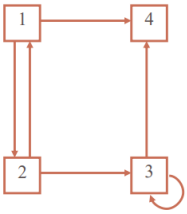


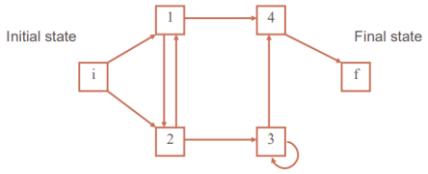
Graphs describing first order Markov chains do not need to be linear, and can still be described by an oriented graph D = (V, A), in which edges in A can connect any pair of nodes in V.
V:={1,2,3,4,5,6}
A:={{1,2},{1,5},{2,3}, {2,5},{3,4},{4,5},{4,6}, {5,2},{6,4}}

In a graph depicting a Markov chain, edges express the dependencies among the system states. The idea is that now we can ask the probability to be at a particular node. It depends on at

which the path is in the previous step. For example I can't reach node 5 from nodes 3 and 6. Also the probability to get to the node 5 from nodes 1, 2, and 4 is different.

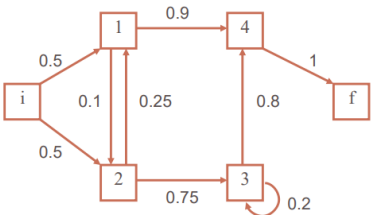
Let's see another example:
this is an oriented graph composed by four nodes (states) and their dependencies (edges):
Since these graphs are generally used to model sequential chains of events, an initial and final node are often added.



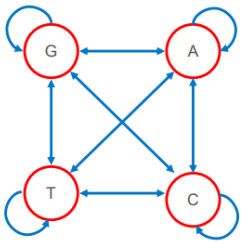


There are several possible paths in the graph from the initial to the final state: 1234; 234; 14; 12121214; 21234, etc…
Anyway the layout of the graph imposes constraints on the possible paths: the last state in the path is always state 4, the first can be state 1 or 2, etc

Let's now add to each edge a probability of being traversed.
Movements from one state to another following an edge are called transitions.
The probability of reaching a specific state depends only on which state we are currently in.
The sum of the probabilities of the edges outgoing from a state must always be 1. For example, for state 1, we have probability 0.9 of transitioning to state 4 and 0.1 of going to state 2.
Instead, the sum of the probabilities of incoming edges can be ≠ 1.



Therefore, we can model paths in the graph as series of conditional probabilities and we can compute the probability of any path in the graph by multiplying the probability of each transition in the path. In thi s way we can also compute which is the most probable path in the graph.



We can create a model for the nucleotides, in which each state correspond to a nucleotide. All the states are linked with each other and each transition has a given probability.

If the states of the model represent the 4 nucleotides, we can use the model to:
1) Generate a nucleotide sequence as a stochastic path in the graph
2) Compute the probability of a sequence given the model

Each edge (i.e. each transition between states) is modelled as the conditional probability P(X|Y); Such probabilities can be computed as the frequency of all possible dinucleotides given a reference sequence.
For example, given this table of probabilities:

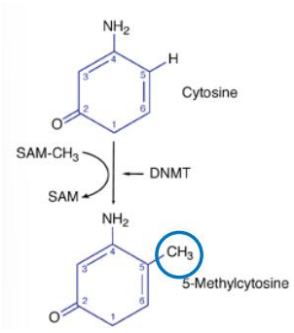
	G	A	T	C
G	0.35	0.22	0.12	0.31
A	0.14	0.32	0.30	0.24
T	0.23	0.28	0.10	0.40
C	0.18	0.19	0.30	0.33

The probability of ATTGC = P(T|A)P(T|T)P(G|T)P(C|G) = 0.3*0.1*0.23*0.31 = 0.002139

Emi Lesson 27 26.05.2023

We can use this value as an indication whether or not this sequence can be another member of this family(if it meets the feature).
Setting a threshold is not easy because we get it multiplying lots of tiny numbers, so the probability will be very small.
To set the threshold what we can do is to create a model that ca be used as a reference.
Let's see an example of how to use the model to describe CpG islands.

Let's consider the case of DNA methylation, and epigenetic signal in which cytosines, often followed by a guanine, are covalently modified by the addition of a methyl group in the 5 position of the ring.

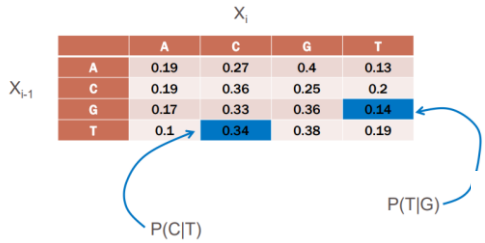


C methylation often (but not always) occurs on several CG dimers in the so called CpG islands that are found upstream (but not always) to many eukaryotic genes.

C methylation is a repressory signal (but not always) that inhibits the transcription of the closest downstream (or upstream) gene by changing the conformation of chromatin.
It's important to study methylation patterns since they can be different in different cell types.
Cg methylation is useful to map where CpG islands are.

We would like a way to identify CpG islands by scanning the genome using a model that computes the probability that a tract of the genome is a CpG island.
Our goal: a strategy for scoring a genome k-mer according to how confident we are it belongs to a CpG island.
If we have a reference set, i.e. a set of known CpG islands (possibly from the same genome we want to annotate), how can we compute the model parameters given the known examples?

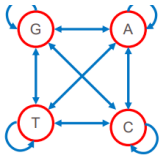
For simplicity, let's consider all k-mers of length 2 (dimers) in the known islands, and compute their frequency (that we are going to consider as conditional probabilities, modelling the sequence as a Markov chain of order 1).



X_{i-1} \ X_i	A	C	G	T
A	0.19	0.27	0.4	0.13
C	0.19	0.36	0.25	0.2
G	0.17	0.33	0.36	0.14
T	0.1	0.34	0.38	0.19

In the following way we can estimate the probability of having C given that the previous nt is a G.
P(G|C) = # of times CG occurs / # times CX occurs;
where X is any nucleotide
We can do the same for all 16 possible dimers p(Xi |Xi-1) and put the values in a table.

	A	C	G	T
A	0.19	0.27	0.4	0.13
C	0.19	0.36	0.25	0.2
G	0.17	0.33	0.36	0.14
T	0.1	0.34	0.38	0.19



We can now plug these inside a CpG frequencies in the model graph and compute the probability that a given sequence X is part of a CpG island.
The only problem is the first nucleotide and to solve it we need another table reporting the relative frequencies of the 4 nucleotides, or we can just set it to 0.25.

If *X*=GATC, then P(X)=P(G)P(A|G)P(T|A)P(C|T) = 0.25*0.17*0.13*0.34 = 0.002

Probability of G, for simplicity set to 0.25

Inside CpG

	A	C	G	T
A	0.19	0.27	0.4	0.13
C	0.19	0.36	0.25	0.2
G	0.17	0.33	0.36	0.14
T	0.1	0.34	0.38	0.19

Outside CpG

	A	C	G	T
A	0.34	0.18	0.23	0.25
C	0.38	0.27	0.04	0.33
G	0.3	0.2	0.25	0.25
T	0.22	0.21	0.26	0.31

Once we got the probability, how can we determine if it is sufficient to s
ay that X is part of a CpG island?
We need to compare P(X) to something, used as a background, which could be estimated as the probability of X not being part of a CpG island.
This can be done in different ways but what we usually do is to take strings, of the same length, randomly from the genome and compute the probabilities.
We can notice how P(G|C) is much smaller outside CpG islands than inside them.
We can now use these outside frequencies and compute the probability that X is not part of a CpG island.
In the example before it will be:

If *X*=GATC, then P(X)=P(G)P(A|G)P(T|A)P(C|T) = 0.25*0.3*0.25*0.21 = 0.004

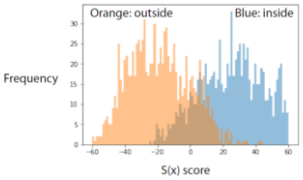
From the two probabilities we got, we can calculate a score that will tell us if it's more probable that our sequence it's contained in a CpG island or not.

$$S(x) = log \frac{P(x)inside\ CpG}{P(x)outside\ CpG}$$

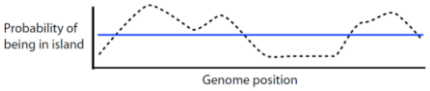
If inside is more probable than outside, then the log ratio S(x) > 0, otherwise S(x) < 0 (or = 0 if the two probabilities are equal). In our case S(X) = -0.69 so the sequence is not in a CpG island.

We still need to set a threshold to accept the sequence as part of a CpG island in order to reduce the false positive.
We can test this simple model on real CpG island to understand how well it works.

This have been done and we can see that, on average, sequence that are from inside have a greater S(X) score than the other.



Since it seems to work fairly well, we can use the model to scan a genome looking for potential CpG islands.

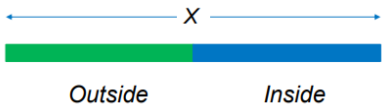


We can use a sliding window of length k (we could use the average length of a real CpG island) to test each genome k-mer from one end of a chromosome to the other end. In this way we can see at which position CpG islands could be.

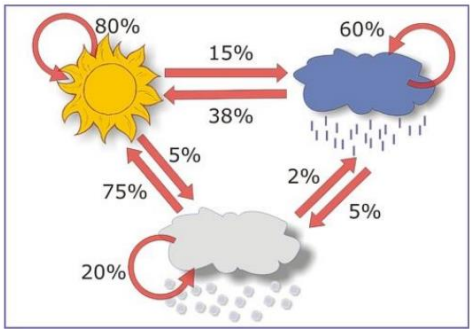
The choice of k is crucial:
- If k is too small, each real island could be identified but just as a set of small islands
- If k is too large, we might miss entire small islands

Moreover there is another problem. Sequences can be classified only as *inside* or *outside*. Consider the case of a sequence corresponding to the beginning of a CpG island and of the part of genome that precedes it.

Neither the *inside* nor the *outside* model is going to assign to this sequence a high probability. We can say that our model works properly only if the input sequence is entirely inside or outside.



The problem cannot be solved from our models. In this case our goal would be to create a model that is able to unify both models, or at least to have a way to switch among them to model sequences entering or exiting a CpG island. However, finding a way for switching between the two models is not easy.
When different models are needed to describe a system, and for systems composed by a large number of states, we need something more complex. Let's now define a hidden Markov model.



This is a model with three states (nodes). Each of one is labelled to one weather condition. Let's assume that a day can have just one weather condition. States are connected by edges and each one have a transition probability. We can consider them as relative frequencies. Since now that's exactly what we have seen for Markov models. OBJOBJ

Resuming the components of the model, we have:

- Set of states $\{S_1, S_2, \ldots, S_N\}$
- Transition probabilities (transition matrix)
  $A_{ij} = P(q_{t+1} = S_i \mid q_t = S_j)$
- Initial states probabilities $\pi_i = P(q_1 = S_i)$

In particular we can get the transition matrix and the initial states probabilities.

From the model we can calculate the probability to get a particular sequence of events.

$P(S_{sun}) \times P(S_{rain} \mid S_{sun}) \times P(S_{rain} \mid S_{rain}) \times P(S_{rain} \mid S_{rain}) \times P(S_{snow} \mid S_{rain}) \times P(S_{snow} \mid S_{snow}) = 0.7 \times 0.15 \times 0.6 \times 0.6 \times 0.02 \times 0.2 = 0.0001512$

Weather model components:

- Set of states
  $\{S_{sun}, S_{rain}, S_{snow}\}$
- Transition probabilities (transition matrix)

$A =$

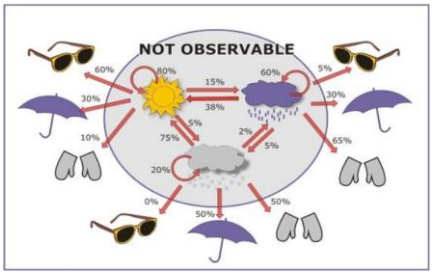
	Sun	Rain	Snow
Sun	.8	.15	.05
Rain	.38	.6	.02
Snow	.75	.05	.2

Rows = outgoing edges
Cols = incoming edges

- Initial states probabilities
  $\pi_i = (P(S_{sun})=0.7, P(S_{rain})=0.25, P(S_{snow})=0.05)$

Given a model of a system described as a first order Markov chain, and all its parameters, we can easily compute the probability of any sequence of events, and there are efficient algorithms for computing the most probable series of events. We can compute the probability of a series of events because we know the path in the graph corresponding to that series of events, therefore it is easy to fetch the corresponding transition probabilities.
What if the sequence of events is unknown? What if we only known some observable features which depend on the series of events, but that can be generated by different sequences of states?

Let's imagine that we don't know the weather of the day but we know what the person is wearing.
This means that the model and the states are hidden.
What we know is called observation.
Of course each observation have a probability depending on in which state we are.



	0.1	0.3	0.6
	0.65	0.3	0.05
	0.5	0.5	0

Observation (emission) probability matrix

Our model now contains some other components:

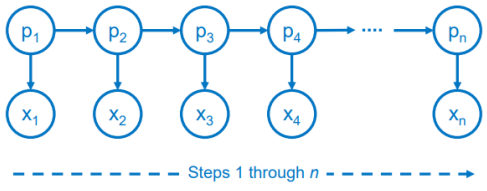
- Observations: $\{O_1, O_2, \ldots, O_M\}$
- Observation probabilities: $B_j(k) = P(v_t = O_k \mid q_t = S_j)$

Observations are sometimes also called emission.

The difference between a hidden Markov model (HMM) and a Markov chain is that in a HMM we only know the sequence of observations, and we don't know the series of events that led to the specific series of observations (which is therefore hidden). Nevertheless, the series of observation depends on the series of events by means of the observation probability matrix, reporting the probability of emissions for all possible states of the model. Different series of events can lead to the same series of observations, but we can compute their probability and identify the most probable one.

Let's imagine that we follow a person for a week and we have 7 different observations.

How can we determine which is the most likely path along the model states that could have generated it?
The number of possible paths is very large but I want to find the most probable. In order to do that we need a systematic approach.

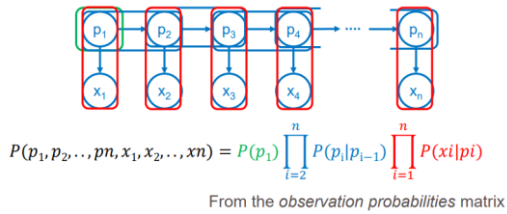


p ={p_1 ,p_2 ,..,p_n } is a sequence of states (i.e. a path in the model). Each p_i is part of the set S of states. We cannot observe p, which is hidden to us
x ={x_1 ,x_2 ,..,x_n } is a sequence of observations (or emissions) emitted by each p_i. Each x_i is part of the set observations O for each state.

Edges are conditional probabilities:

- xi is conditionally dependent on p_i and independent on anything else
- p_i is conditionally dependent on p_{i-1} and independent on anything else

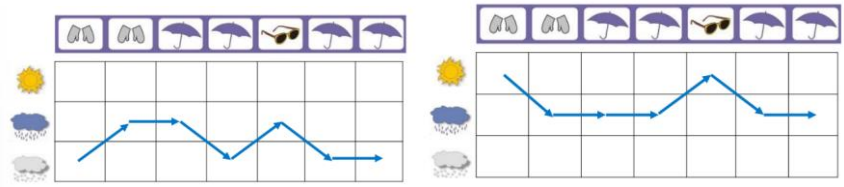
We can compute the probability of a path in the following way:



$$P(p_1, p_2, .., pn, x_1, x_2, .., xn)$$

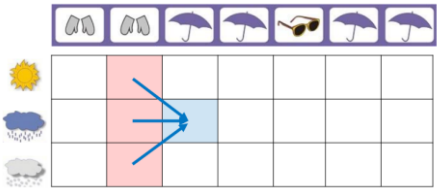
This is possible for each path.
If we calculate the probabilities for each path we can understand which is the most probable. The problem is that the number of paths can be very large.
This can be solved using an algorithm of the dynamic programming class algorithm.

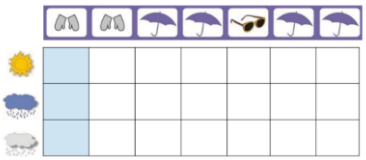
We can represent the sequence of observations as a matrix, in which columns are the observations and rows are the states that might have emitted them. Any path in this matrix corresponds to a specific series of states that could have generated the series of observations.



For each path we can compute its probability. We would like to compute the optimal path without having to exhaustively evaluate each one of them.
What we can do is to fill the matrix with all the probabilities and then, starting from the last one and tracing back, we can calculate the most probable path. Therefore, the problem is quite similar to the alignment of two sequences, and we can solve it by using variants of the dynamic programming algorithms that we described for pairwise sequence alignment.
The **Viterbi algorithm** is used for decoding a series of observations. Given the model parameters (i.e. the model structure and all initial, transition and emission probabilities) and a sequence of observations, the algorithm computes the most probable sequence of states that can lead to that sequence of observations.
Let's see how this algorithm works.

Each cell in the matrix can be reached from each cell of the previous column.
A cell in position t in the series of observations can be filled with its optimal value by computing the probability of reaching the cell from the states at positions t-1 that have a transition probability with it > 0, and then choosing the direction having the maximum value (i.e. the maximum probability).

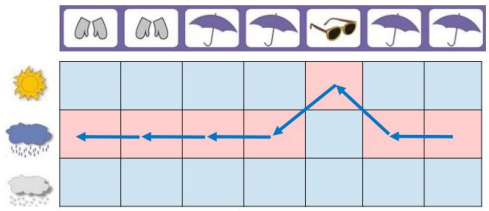




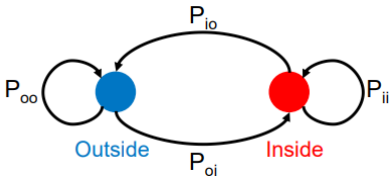
The matrix is first initialized by using the initial state probability matrix and the associated emission probabilities for each state.

To fill the whole matrix we need to compute the transition probability from one state to another, taking in account also the observation probability. For each cell we will have 3 different values (calculated starting from the three possible cells of the previous column) and we need to choose the greatest value. Of course we also need to store the direction from which we calculated the probability chosen.

Once the entire matrix is filled, we can identify in the last column which is the cell having maximum score, and from there reconstruct backwards which is the best path leading to it. A traceback is therefore used to reconstruct the optimal sequence of states that could have generated the query series of observations, starting from the cell in the last column having the highest probability.



We now want to learn how to apply this algorithm to our problem of finding CpG islands.



We have a model with only two node, one for being outside the CpG island, the other one for being inside.
For each node we have two possible transition, one for staying in the same node, the other one for moving to the other node.

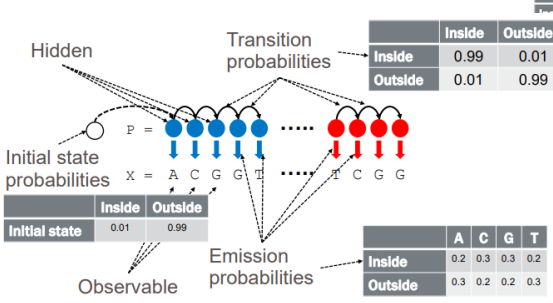
Given a sequence X, we can consider the nucleotides in X as the observations, and what is hidden is the path within the system states. We need to find the most likely path P in this model, in this way assigning a label to each nucleotide in X depending on which of the two states (Inside or Outside) is more likely to have emitted it.

Our model is therefore composed by:
- Two states S = {inside, outside}
- Initial state probabilities for states in S
- Transitions from/to each state
- Four observables for each state: O = {A, C, G, T}

```
X = CGATGTAGCTGCACTGTAGC.....CGATGCTAGTG
P = OOOOOOOOOOOOOOOOOOOO.....IIIIIIIIIII
```

Emi Lesson 28 29.05.2023

In a more explicit way we can express our model with matrices.



	Inside	Outside
Inside	0.99	0.01
Outside	0.01	0.99

	Inside	Outside
Inside	P_{ii}	P_{io}
Outside	P_{oi}	P_{oo}

Transition Matrix

	A	C	G	T
Inside				
Outside				

Emission Matrix

	Inside	Outside
Initial state	0.01	0.99

	Inside	Outside
Initial state	P_i	P_o

Initial state matrix

	A	C	G	T
Inside	0.2	0.3	0.3	0.2
Outside	0.3	0.2	0.2	0.3

If we evaluate the values on real data these are some real values that we could get.

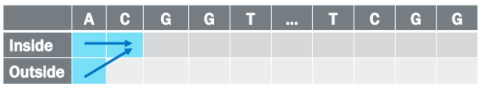
Given the observations, we can solve the problem using the Viterbi algorithm.

	A	C	G	G	T	...	T	C	G	G
Inside										
Outside										

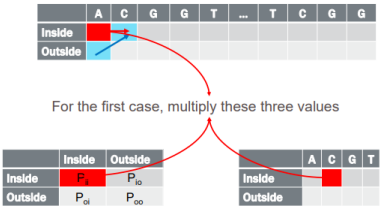
The matrix that needs to be filled contains on one axis the two possible states, and on the other axis the sequence of observations. After initialization with initial state probabilities, the matrix is filled cell by cell up to the end (on the bottom right corner). Then, one can traceback, from the cell with highest probability in the last column, up to the first column to get the optimal path.
As before:

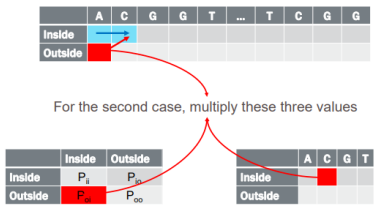
- Each cell can be reached from any cell corresponding to the previous state, with which the state associated with the cell has a transition probability > 0.
- The score of each cell is computed on the probability of the observation at that step, conditional to the state, and on the content of the previous cell
- To fill the first column we need to plug in the initial state probabilities
- All remaining cells can be filled by computing the optimal way to reach a cell from all previous states for which the transition probability is not 0 and for which the emission probability of the observed emission is not 0

In this case, to fill this cell we have two possibilities:



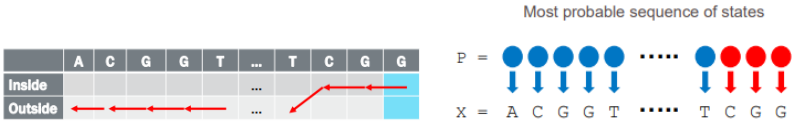
case i) remaining Inside and emitting a C from Inside;
case ii) moving from Outside to Inside and emitting a C from Inside





- Then, take the max of the two cases and store the direction
- When the matrix is complete, the traceback starts from the cell in the last column having the highest probability, and proceeds following the optimal path until reaching a cell in the first column

The traceback determines the most probable sequence of states P, therefore labelling each nucleotide in X with one of the two states (I or O)



Reassuming the problem in a more schematic way, we can say that:
for a series T of observation O, and a model having N nodes we need to follow these steps:

1) Initialization — To fill the first column of the matrix

2) Recursion — To fill the whole matrix

3) Termination — The matrix it's already complete and we need to take the maximum value.

$$P^* = \max_{1 \le i \le N} v_T(i)$$

The last step is the traceback

Max probability in the last column of matrix v

Transition probability from state i to state j

TIME COMPLEXITY OF THE VITERBI ALGORITHM

The dynamic programming matrix contains T*N cells. For each cell we need to perform at most N operations, for computing the probability of reaching a cell from any of the N states.
Overall complexity is then **O(TN^2).**
The complexity increases with the square of the number of states, i.e. the nodes in the graph.
In the case of CpG island prediction, T might be long, but the number of states N is small, therefore the algorithm can run in a short amount of time. More complex models can have instead slow runtimes.

Many algorithms have been developed for hidden Markov models. All algorithms share the same basic principles, and can be classified for their purpose in:

- Scoring: probability of a sequence X given the model and its parameters
- Decoding: find the best possible path for X (e.g. Viterbi algorithm)
- Learning: from a set of reference examples you compute all the probabilities required in the model (unsupervised learning, e.g. Baum-Welch algorithm, Viterbi training algorithm

By determining the most likely state that have emitted each character in the query string, the string classification is not just a label associate to the entire string (e.g. inside or outside) like we did using simple Markov chains models, but each nucleotide is instead labelled by the state of the model it is associated to:

```
X: ATTATAGATATATTATATAAAACGGAGCGCACGCGCGCGG
P: OOOOOOOOOOOOOOOOOOOOOOIIIIIIIIIIIIIIIII

X: TTATATAAAACGGAGCGCACGCGCGCGGATTATAGCGTAT
P: OOOOOOOOOOIIIIIIIIIIIIIIIIOOOOOOOOOOOOO
```

I = inside
O = outside

Lesson 29 30.05.2023

Hidden Markov Models

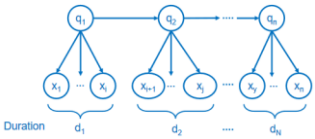
The main problem for a hidden Markov model for the description of genomic sequences is that it does not explicitly take into account that genomic features often have characteristic lengths.

For example, the following path of states for the CpG island problem could have the best probability, but it is biologically unlikely.

```
X: CTCACTCACTCACACACACTCGCACTCACACACACTCTCT
P: IOIOIOIOIOIOIOIOIOIOIOIOIOIOIOIOIOIOIOIO
```

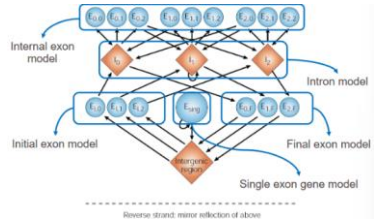
To force the algorithm in having the correct biological relevance, it could be a solution to make it take in consideration the length of the feature of the analysis. In order to do that, known distributions of length of these genome features are plugged in the algorithm.

In the Hidden Markov Model, if the probability of remaining in the inside state is P_{ii}, then the probability of having L consecutive inside residues is $P(L) = P_{ii}^{L}$. Which is an exponentially decaying distribution, which is not the distribution of known CpG island lengths for example.
To better model a CpG island, we need to force the search of the best path in the model to stay in each state for a number of steps that is sampled from the known length distribution of the feature that the model is describing.
Therefore, we need to add parameters specifying the **duration** of how long the path stays in each state.



Duration is based on the length distribution estimated on real example of a feature. And the model derived by it is called explicit state duration HMM.

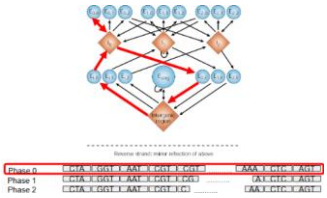
Once the models for specific features are created, it is possible to concatenate them forming General HMM.



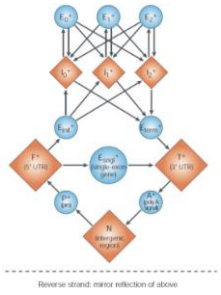
Generalized Hidden Markov Model are formed by states which are models, that sometimes can be complex because they are an agglomerate of modules (e.g., an intron model state could have one model for finding the intron-exon junctions and one for the branching site, one for the sequence of the intron itself, etc.).
These states (models) are therefore still connected to each others with some transition probability and duration.

Models tend to be complex because there are some characteristics of these genome features that are not easily implemented in a statistical model. (e.g., recognize the sequence of exons and introns, which can have different characteristics depending on themselves).

Paths in these models can predict genes if there's something that can be considered as one, which can be decided based on the probability of the optimal pathway and setting a threshold.
Of course, finding genes is difficult because of the presence of many different splicing isoforms.



Gene finding tools



Genescan

Has been developed for drosophila because human genomes were not available.
It includes: introns, exons, promoters, polyadenylation sites, etc.
It can identify whole genes, or just fragments, multiple genes separated by intergenic regions, genes on both strands.
An important feature is that it requires training data from genes already known in order to have a way to weight these parameters.

Twinscan

It is based on Genescan but employs some notion of sequence conservation to facilitate the gene scanning. It is pretty accurate.

```
(Target) Human :    ACGGCGA-GTGCACGT
(Support) Mouse :   ACTGTGACGTGCACTT
Alignment :         ||:|:||.|||||:|
```

The algorithm of Genescan works on protein coding genes, not other kind of transcribed genes and it is an over predictive model (many false positives).

In order to avoid this issue, Twinscan embedded a model of **evolutionary conservation**.
Given a target genome, that needs to be annotated, and a support genome (informant sequences), sequence alignment is used to identify similar regions between the two genomes.
Each target genome nucleotide is then encoded differently depending on how it aligns with the support genome: gap (.), mismatch (:), or match (|).

```
(Target) Human :    ACGGCGA-GTGCACGT
(Support) Mouse :   ACTGTGACGTGCACTT
Alignment :         ||:|:||.|||||:|
```

A new 12-character alphabet is adopted to describe simultaneously a sequence and its conservation:

{A., A:, A|, C., C:, C|, G., G:, G|, T., T:, T| }

Therefore, the point of the model is that specific conservation patterns can be used for discriminating among features.

i.e., Let's say that we are testing a query sequence, and that it might be an exon or an intron. Using a support genome, we can identify the conserved nucleotides in the query sequence when aligned to the support sequence.
At this point, it is possible to check for specific patterns of conservation to recognize introns and exons, since exon sequences have pattern of conservation which allow a mismatch between the third nts of the codons (which do not generally encode for a different amino acid).

N-SCAN implemented this idea of using conservation to predict the gene, applying it with multiple sequence alignment which is more efficient since it is comparing more than just two genomes.

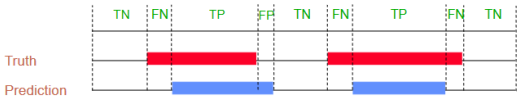
Contrast

It's based on another statistical model called Conditional Random Field (CRF). It had been considered the best accurate for years. (just to say know that there are also different approaches to solve this problem)

Since different algorithms for gene finding were proposed in the scientific literature, it is important to assess their performance by running the gene finder on a benchmark set of genes that are already annotated (for example by mapping protein or mRNA sequence over the genome sequence).
This evaluation can be done in different ways:
It can consider a gene correctly predicted if all (or a certain fraction) of its nucleotides are correctly predicted, or if all (or a certain fraction) of its exons are guessed correctly.
Or it might be ok if the gene finder correctly predicts where the gene is, even if some exons are missed, or exon boundaries are not precise.

In the example illustrated here, the real gene (Truth) has two exons, and the Prediction returned by a gene finder guessed correctly both of them, but the first exon is shifted with respect to the correct on and the second is shorter.
How could we evaluate this case?

1. Assessing performance at **nucleotide level** defining which of the nucleotides are:
   - True positive TP
   - True negative TN
   - False positive FP
   - False negative FN



It is therefore possible to plot these information using different matrices like the confusion matrix.
To assess **sensitivity**: Sn = TP / (TP + FN)
To assess **specificity**: Sp = TN / (TN + FP)

2. Assessing performance at **exon level**
   In the example, the prediction is going to be considered accurate but not perfect, since there are several false positive and false negative nucleotides. But if we focus on exons, we might say that, with some of tolerance, the prediction is correct, since both exons are identified (even if not perfectly).

There's no specific definition of what a true negative is. But it is possible to identify as true positives all exons that the gene finder identified, with some tolerance, as false positives all predicted exons not having any correspondence with real ones, and as false negatives all real exons that the algorithm completely missed (or when the overlap with the prediction is below the tolerance threshold).

$$\text{Sensitivity} \quad S_n = \frac{\text{Number of correctly predicted exons}}{\text{Total number of exons in the dataset}}$$

$$\text{Specificity} \quad S_p = \frac{\text{Number of correctly predicted exons}}{\text{Total number of predicted exons}}$$

3. Assessing performance at **gene level**

Depend on the tolerance threshold, a gene is considered correctly predicted only is all its exons (or a sufficient fraction of them) are correctly predicted.

$$\text{Sensitivity} \quad S_n = \frac{\text{Number of correctly predicted genes}}{\text{Total number of genes in the dataset}}$$

$$\text{Specificity} \quad S_p = \frac{\text{Number of correctly predicted genes}}{\text{Total number of predicted genes}}$$

Full genome annotation

Usually these tools are used when there are no other options to map genes.
The pipeline:

1. Cis-alignment mapping of protein
2. RNA-seq or RNA libraries
   mapping them will extend the protein alignments adding promoter regions, UTRs, etc.
3. Trans-alignment alignment of sequences or proteins from some other close species
4. Use these info to train a gene predictor
   if the gene number is too low, it is possible to use the few found to compute models which will be used to detect the rest of them.

Other options used by some pipelines are using comparative genomics, or filters for pseudogenes, etc.
The limit of gene finding algorithms are that:

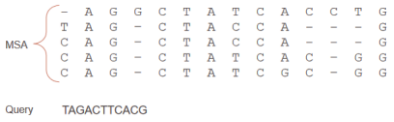
- They cannot identify:
  - Overlapping genes
  - Nested genes
  - Frame-shifts
  - Alternative start or stop codons
  - Non-canonical splicing junctions
  - Alternative splicing
- Are species-specific
- Are designed only for protein-coding genes
  Non-coding genes might not have the same features and therefore might be missed.

Profile Hidden Markov Models

Hidden Markov Models are quite powerful tools which have many applications in Bioinformatics.
One or the most successful applications of HMMs in bioinformatics is for describing **protein or gene families**. Also known as profile HMM, since the result is a sort of profile, a model of features of a specific family.

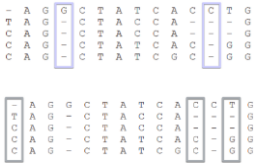
The starting point is a set of sequences that are clear homologs, and that can be aligned obtaining a multiple sequence alignment (MSA). A model of the family is built using the MSA characteristics, and then the model can be used for computing the probability that a query sequence is part of the family, and to get its alignment to the other family members.

Starting from genomes of the same family, MSA is computed and then it is possible to determine whether the query sequence could be member of that family.



Since it is still a HMM, it still have nodes, transitions, emissions and all the rest. It has an organization such that each state in the model describe different column in a multiple alignment of member of the same family.

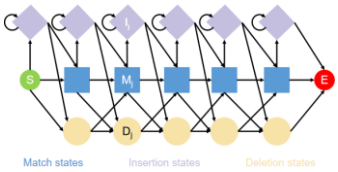
Some columns of the MSA are well conserved and have no gaps, suggesting that these positions are under stronger selective pressure.
Other columns contain gaps, suggesting that in those positions there is more tolerance for variability since the selective pressure on these positions is less.



In some of these columns containing gaps, only one or few sequences might show residues and all the others have a gap, and it is reasonable to assume that the sequences carrying a residue at these positions were subject to an insertion with respect to the ancestral sequence.

In some other columns containing gaps, most sequences show residues and only few have a gap, and it is reasonable to assume that the sequences lacking residues at these positions were subject to a **deletion** with respect to the ancestral sequence.

At this point the objective is to create a Hidden Markov model based on these information.



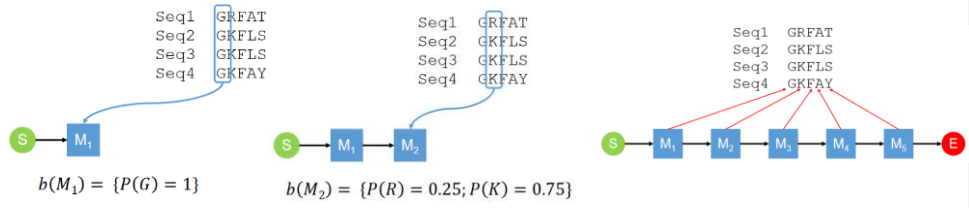
The blue squares are called match states (no gaps observed), the purple and the yellow one instead, model for insertion and deletions.

This is going to be a model of the entire family that is going to be used to label each nucleotide or amino acid of the input sequence with one of these states. In this way each query sequence is aligned with the other members of the family. Together with the alignment, also the overall probability of the query which can be used to decide whether it is a new member of the family or not.

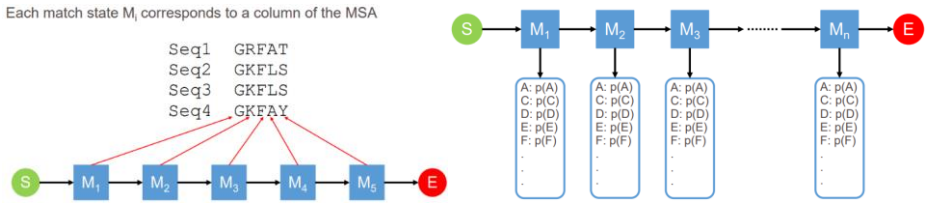
Start form a set of member of these family for which there's a clear homology.

e.g.
```
Seq1  GRFAT
Seq2  GKFLS
Seq3  GKFLS
Seq4  GKFAY
```

Consider each column of the alignment as a different state Mj, and the relative frequency of each amnio acid in each column as the state emission probabilities.
For each multiple alignment column Mj, it is possible to compute the absolute and relative frequency of residues, to be used as emission frequencies of each state Mi.

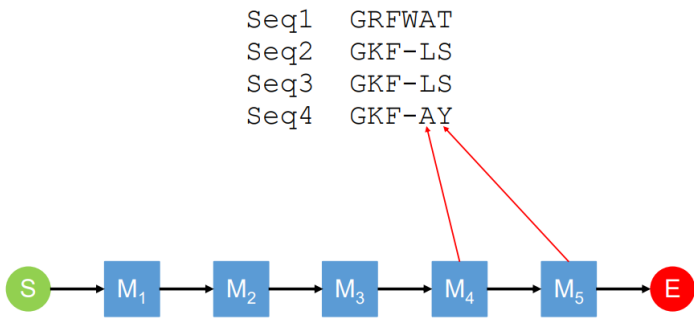


Multiple sequence alignment can be converted into states equal to the number of columns. These states are usually called match states (=no gaps | can be mismatch).
They are connected – transition to one column to the next.
Each match state is characterized by its specific emission probability, computed as relative frequency of different residues that you observe in this multiple alignment.



The states M1 ,..,Mn are called match states. This description of a multiple alignment by means of a HMM is similar to a position weighted matrix, in which the residue relative frequencies of each column of the alignment are used as emission probabilities. Edges connect match states corresponding to consecutive columns of the multiple alignment, with transition probabilities P(a(Mi),a(Mi+1)) that at this point can be all set to 1.

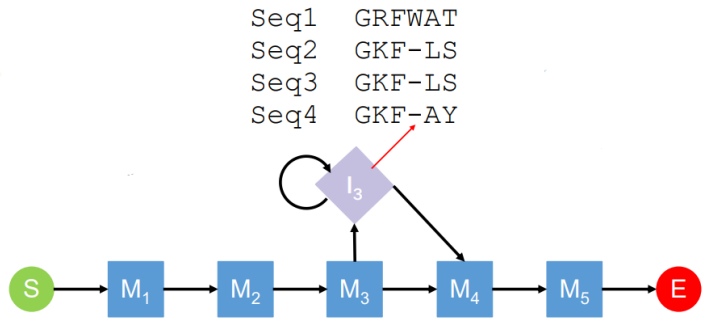
What if we have gaps?
If the multiple alignment contains columns in which one or few sequences have an insertion, we could model these columns in the same way as the match states, but it would be better to treat them differently in order to consider that they correspond to protein sequence sites that could be subject to a different selection.



In this example, the first 3 columns of the MSA can be modelled as usual with a match state Mi.

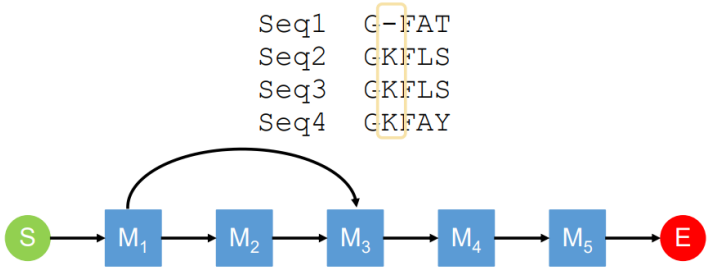
Same for the last two columns.

What to do with the fourth column, containing several gaps? Let's add a specific insertion state Ii , different from a standard match state Mi , to handle this case. This state is connected to the match states corresponding to the MSA columns right before and after the column with gaps.



The true path for a sequence with a gap would just follow the linear sequence of match states. The path for a gapless sequence, i.e. a sequence containing an insertion, will move from a match state to an insertion state, and then back to the next match state.
An insertion states describe alignment columns with multiple gaps. Their emission probabilities can be modeled as frequencies of the residues carried by those sequences in the MSA not having gaps at that position, or (more often) by background distributions

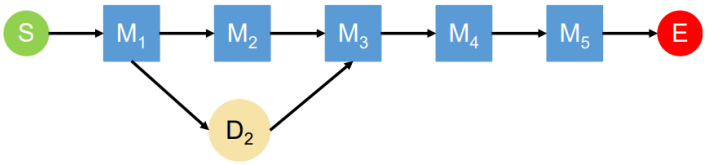
If one or few sequences in the alignment contain positions in which they have a deletion, we could model this by adding edges to the model skipping the deleted positions.



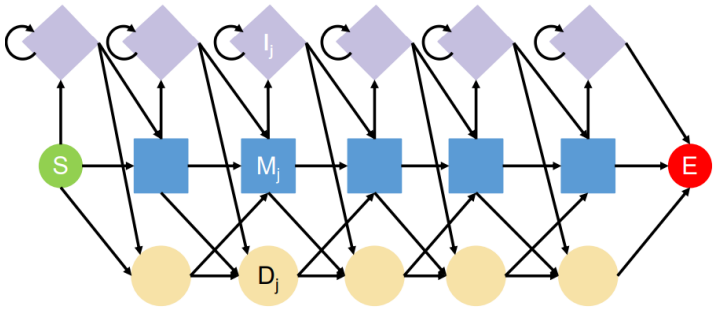
Using this approach, to allow for deletions of any length we should draw edges connecting each pair of states, which would result in a more complex graph and would also imply the computation of several transition probabilities that could be hard to estimate.

We could solve this problem by using states corresponding to deletions, that can be used to skip one or more match states. These deletion states are silent, meaning that they have no emission probabilities. The true path for a gapless sequence would just follow the linear sequence of match states. For a sequence containing a deletion, passing through a deletion state can be used to skip one match state.

```
Seq1   G-FAT
Seq2   GKFLS
Seq3   GKFLS
Seq4   GKFAY
```



Consecutive deletion states are connected to each other, allowing for deletions of any length. Probabilities of transitions between deletion states are estimated by the deletions length in the multiple alignment sequences. Putting all together, we have a profile hidden Markov model.



How to build the model:

```
S1   VGA--HAGEY
S2   V----NVDEV
S3   VEA--DVAGH
S4   VKG------D
S5   VYS--TYETS
S6   FNA--NIPKH
S7   IAGADNGAGV
     MMMIIMMMMM
```

Usually, positions with more than 50% gaps are modeled as insertions states (I), the remaining positions as match states (M)

Frequencies of adjacent positions in M and I states are used to model transition probabilities from match to insertion states.

Amino acid frequencies in M states are used for emission probabilities in each M state

Deletion states are added at each position of the model

One problem could emerge in the emission probability of match states. The model parameters are computed from the sequences chosen for representing the family, used as a training set, but a useful model should be able to recognize other family members not present in the training set, from which they might be different in small or more significant ways. Parameters therefore must be smoothed, such that the model is not too specific and able to recognize divergent family members. Hence, one must avoid overtraining and overfitting and the model must be more general.

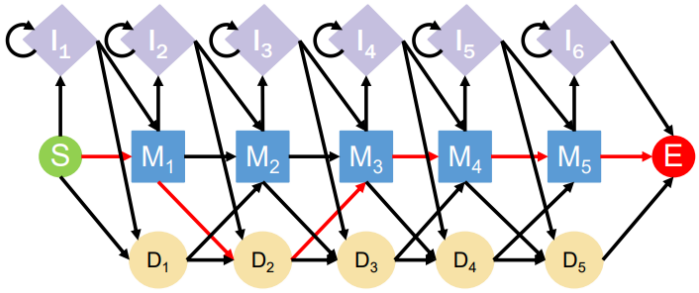
To avoid having model parameters too dependent only on the sequences in the multiple alignment, therefore not capable of correctly handling more divergent sequences (overfitting), pseudocounts are added to the absolute frequencies of emissions and transitions. This way, the model should be better at generalizing the protein family.

$$a_{k\to l} = \frac{|k\to l| + |B| \cdot q_{k\to l}}{|B| + \sum_{l'} |k\to l'|}$$

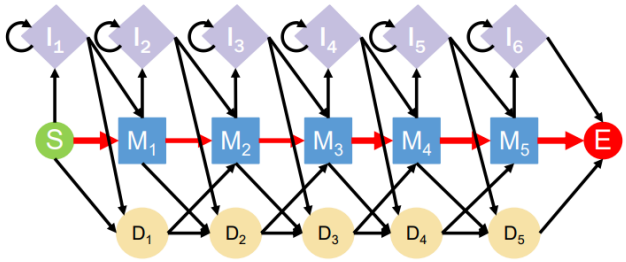
$$e_k(b) = \frac{|k(a)| + |B| \cdot q_{k(b)}}{|B| + \sum_{b'} |k(b')|}$$

A count B is added to the absolute counts of the transitions $a_{k,l}$ between match and insertion states and the emissions e_k (B), even for transitions not seen in the multiple alignment or amino acids not seen in the columns of the alignment. In this way, the model becomes able to allow insertions in each point of the multiple alignment, and to allow the handling of sequences that contain differences from those used to build the initial multiple alignment.

A pseudocount of 1 is automatically added for all possible 20 amino acids, and the emission probabilities of the state M1 are computed on all the absolute frequencies plus the pseudocount for all possible 20 amino acids. Let's also add a pseudocount of 1 for transition frequencies between M1 and all other states in the model.

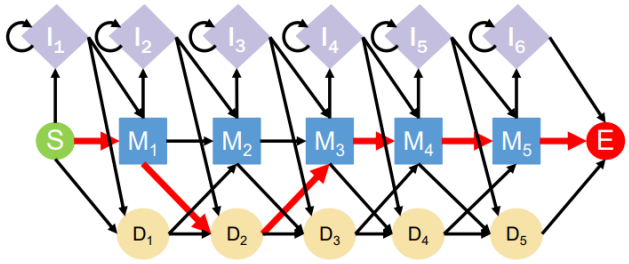


Once the model and its parameters is complete, to each sequence in the starting multiple alignment corresponds to a specific path across the model states.



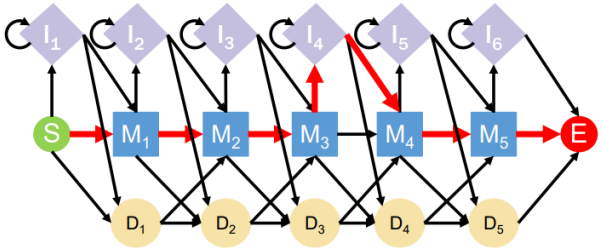

```
Seq1   G-FAT
Seq2   GKFLS
Seq3   GKFLS
Seq4   GKFAY
```

S → M_1 → M_2 → M_3 → M_4 → M_5 → E



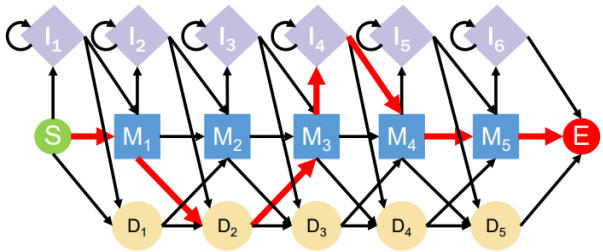

```
Seq1   G-FAT
Seq2   GKFLS
Seq3   GKFLS
Seq4   GKFAY
```

S → M_1 → D_2 → M_3 → M_4 → M_5 → E




```
Seq1   GRFWAT
Seq2   GKF-LS
Seq3   GKF-LS
Seq4   GKF-AY
```

S → M_1 → M_2 → M_3 → I_4 → M_4 → M_5 → E



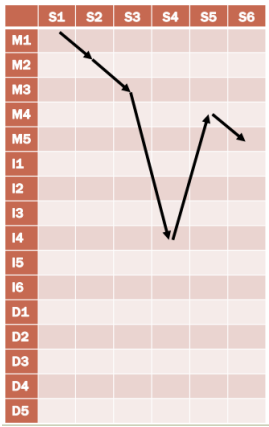

```
Seq1   G-FWAT
Seq2   GKF-LS
Seq3   GKF-LS
Seq4   GKF-AY
```

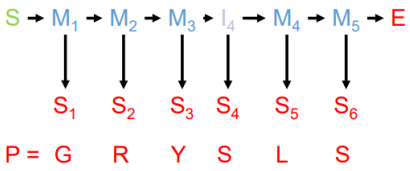
S → M_1 → D_2 → M_3 → I_4 → M_4 → M_5 → E

In the same way, given any query sequence P, not included in the multiple alignment used to build the model, we can use the Viterbi and other algorithms to find the most probable path of states that could be associated to P.

We have to build a matrix having as one dimension all the model states (M, I, D), and as the other dimension all the residues in a query sequence S. Any path in this matrix corresponds to a different alignment of the query with the family.

Using Viterbi and Forward algorithms, we can get the probability of P given the model, and the most likely sequence of states. We can use the probability to assess whether P might be a member of the family.





Moreover, the most probable path corresponds to the alignment of P to the multiple alignment of the members of the model.

The Viterbi algorithm for these models must be slightly modified to account for the different meaning of the 3 types of states:

Recursion:

$$v_{M2}(i) = eM_2(x_i) max \begin{cases} a_{I2M2} v_{I2(i-1)} \\ a_{M1M2} v_{M1(i-1)} \\ a_{D1M2} v_{D1(i-1)} \end{cases}$$

$$v_{I2}(i) = eI_2(x_i) a_{M1I2} v_{m1(i-1)}$$

$$v_{D2}(i) = eD_2(-) a_{M1D2} v_{M1(i)}$$

recursion is slightly different as we can see. Here we have states that might have different nature, for example, the deletion state doesn't have an emission probability. so, when we compute the optimal score, we should take it into account.
No emission -> we only have transition probability
Then we have initialization and termination (check the last column)

Profile HMM advantages:

- They can produce high quality MSA
- Are based on solid statistical background
- No need for arbitrary gap penalties or scoring schemes

Limits:

- A large number of sequences is needed to build a good model (≥ 50)
- Finding parameters can be computationally heavy

SOF	Lesson 6	14.03.2023	fatta
IO	Lesson 7	16.03.2023	fatta
EMI	Lesson 8	17.03.2023	fatta
EMI	Lesson 9	23.03.2023	fatta
IO	Lesson 10	24.03.2023	fatta
EMI	Lesson 11	30.03.2023	fatta
SOF	Lesson 12	31.03.2023	fatta
IO	Lesson 13	13.04.2023	fatta
SOF	Lesson 14	17.04.2023	fatta
SOF	Lesson 15	04.05.2023	fatta
EMI	Lesson 16	05.05.2023	fatta
IO	Lesson 17	08.05.2023	fatta
SOF	Lesson 18	09.05.2023	fatta
EMI	Lesson 19	11.05.2023	fatta
EMI	Lesson 20	12.05.2023	fatta
SOF	Lesson 21	17.05.2023	fatta
IO	Lesson 22	18.05.2023	fatta
SOF	Lesson 23	19.05.2023	fatta
SOF	Lesson 24	22.05.2023	fatta
IO	Lesson 25	23.05.2023	fatta
EMI	Lesson 26	24.05.2023	fatta
EMI	Lesson 27	26.05.2023	fatta
EMI	Lesson 28	29.05.2023	fatta
IO	Lesson 29	30.05.2023	fatta
SOF	Lesson 30	31.05.2023	fatta

SOF 9 EMI 9 IO 7 + ordinare il file