NOTES FROM THE BIOINFORMATICS COURSE

ROBERTA PRENDIN

Settembre 2015 – version 1.0

1

MOLECULAR BIOLOGY AND BIOINFORMATICS

In Computer Science we can easily distinguish between **software** (programs, data and informations) and **hardware** (computers and machineries capable of reading the software). In a living being a similar difference can be made between **DNA**, a macromolecule containing the data and the commands needed to interpret it, and **cells**, components capable of interpreting the DNA and translate it into specific behaviours. These components are not clearly distinguishable though.

1.1 THE DNA

Deoxyribonucleic acid (or DNA) is the macromolecule that **carries most of the genetic instructions** used by all known living organisms and viruses. Along proteins and carbohydrates DNA is a *nucleic acid* essential for all known forms of life, and is present in every cell of our bodies.

1.1.1 *History of the DNA discovery*

Up to 1943 genetic information was thought to be carried by proteins. Despite knowing that DNA was constituted by four different nucleobases scientists believed that their purpose was merely structural. In 1943, however, Avery, McLeod and McCarty discovered that the DNA of *Pneumococcus Pneumoniae* contained the information to make otherwise non-virulent bacteria patogenous. Their result was based on another experiment performed by **Griffith** in 1928. Griffith identified that some ***transforming principle*** could transform non-virulent strands of pneumococcal bacteria into patogenous ones. *Pneumococcus Pneumoniae* is in fact characterized by **two different strains**:

- Smooth strain, with a polysaccharide capsule. Very virulent and responsible of pneumonia infections;
- Rough strain, with no capsule. Not virulent.

Griffith noticed that the transformation occurred only when dead bacteria of a virulent type and live bacteria of a non-virulent type were mixed together and injected in a mice: the mice would develop a fatal infection caused only by live bacteria of the virulent type.

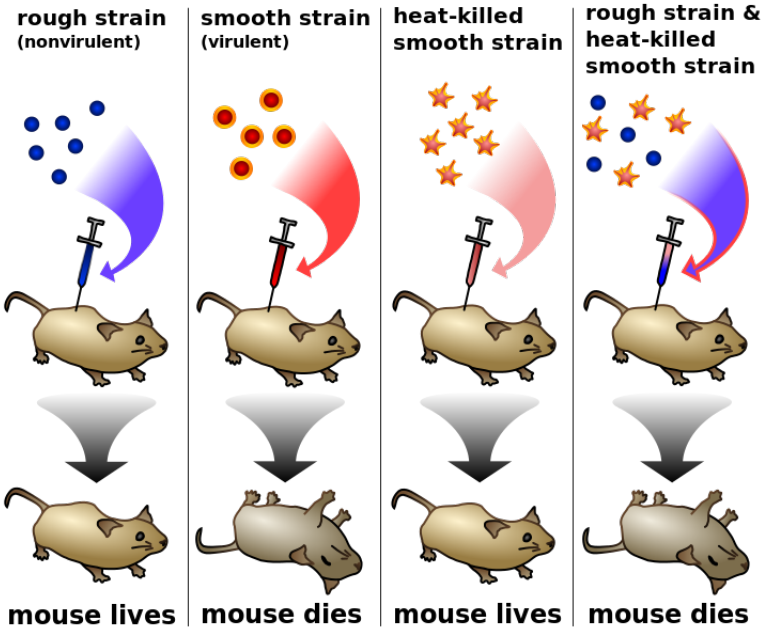


Figure 1: Griffith's experiment (1928).

In 1943 Avery, McLeod and McCarty further developed Griffith's experiment performing a series of biochemical tests, showing that the transformation principle was indeed DNA rather than proteins or some other cell component. The result was that the non-virulent bacteria was capable of **incorporating the DNA of the dead virulent strains**, obtaining the means to produce the capsule that made the other strain virulent.
As for the discovery of the **inner structure of the DNA molecule**, the double helix was first theorized in two historical papers both published on *Nature* in 1953. In the first Rosalind Franklin purported **X rays data** that later allowed Watson and Crick to propose the double helix structure of DNA.

1.1.2 *Structure and components of the DNA molecule*

DNA molecules are composed by **two strands forming a double helix**. Each strand contains millions of simpler units called **nucleotides**, which are composed by three different elements: a nucleobase (**cytosine**

C, guanine G, adenine A or thymine T), a sugar (**deoxyribose**) and a **phosphate group**. Nucleobases are usually in the inner portions of the DNA structure but can be accessed through the **major and minor grooves**.

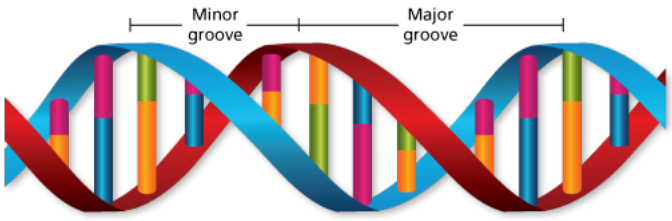


Figure 2: Major and minor grooves of a DNA molecule.

From a **chemical point of view** adenine and guanine are *purines*, containing a pair of connected rings; cytosine and thymine (or uracil, which substitutes thymine in RNA) are instead *pyrimidines* are they are composed by a single ring.

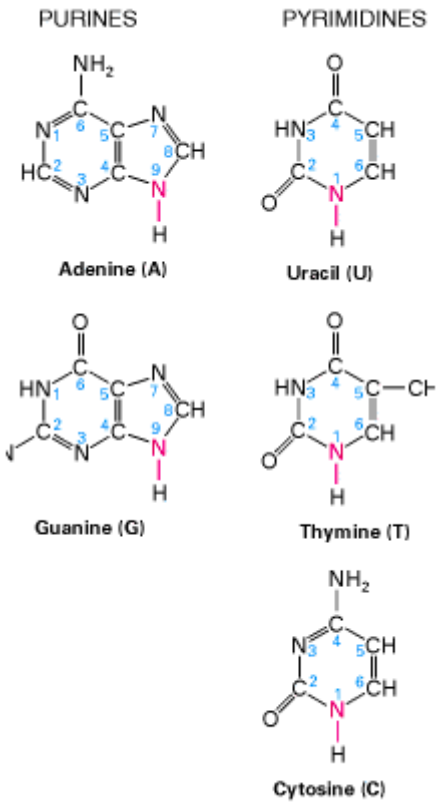


Figure 3: Chemical structures of nucleobases.

Genetic information is encoded as sequences of the four nucleobases. As for the structure itself, nucleotides of each strand are binded together in a chain that alternates the sugar of one nucleotide

and the phospate of a next. The two strands are instead joined by **hydrogen bonds between nucleobases** that can be **broken up and rejoined** with relative ease. These bonds follow a simple rule: **A with T, C with G** – which means that if we know one strain, we can **deduce the other**.

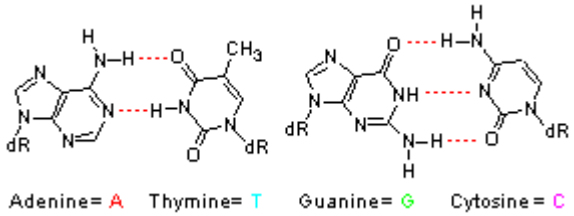


Figure 4: Hydrogen bonds between nucleobases.

Note that both strands of the DNA store the same genetic information but are **anti-parallel**, meaning that each strand run in opposite direction to the other. The ends of the DNA strands are called the 5′ **and** 3′ **ends**, with the 5′ end (start) having a terminal phosphate group and the 3′ end (stop) a terminal hydroxyl group.

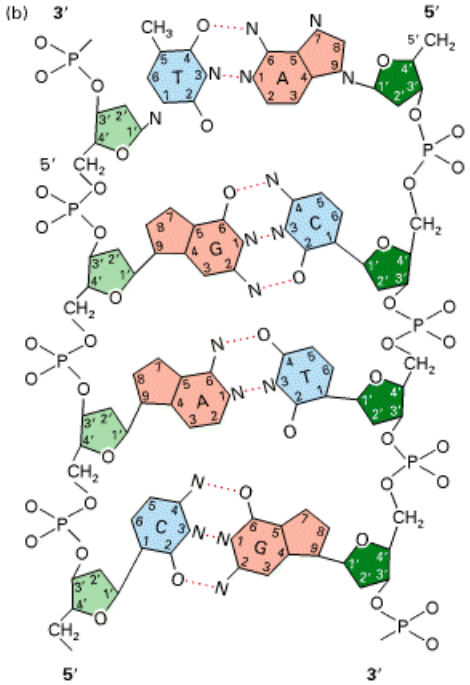


Figure 5: Anti-parallel structure of the double helix.

It's interesting to note that more than the 98% of the macromolecule is ***non-coding***, meaning that these sections do not serve as patterns for protein sequences.

1.2 GENOME AND GENES

Within cells, DNA is organized into long structures called **chromosomes**. Eukaryotic organisms (animals, plants, fungi, and protists) store most of their DNA inside the cell *nucleus* and some of their DNA in organelles (mitochondria or chloroplasts, called *linear chromosomes*); prokaryotes (bacteria and archaea) instead store their DNA directly in the cytoplasm (*circular chromosomes*). The set of chromosomes in a cell makes up its genome. The human genome has approximately 3 billion base pairs of DNA (3×10^9) arranged into 23 chromosome pairs.

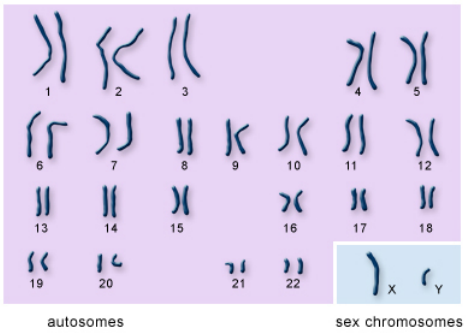


Figure 6: Genome of a human being.

The genetic information in a genome is held within genes, which are **portions of DNA**. Genes are **units of heredity** and influence particular features of organisms; the whole set of genes in an organism is called its ***genotype*** while its translation into physical features is called its ***phenotype***. A gene is associated to an ***open reading frame*** (ORF) that can be transcribed, as well as regulatory sequences such as promoters and enhancers, which control the transcription of the open reading frame.

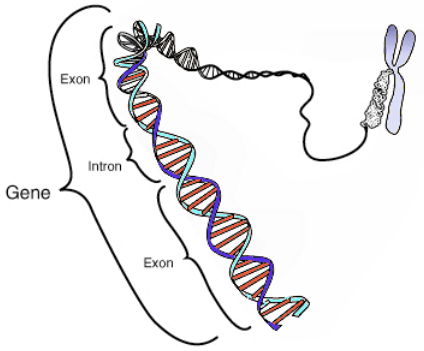


Figure 7: Human chromosomes.

Each gene **encodes the sequence of aminoacids** composing a particular proteine. DNA also contains regions for determining how and when each gene has to be expressed; it may in fact depend on the type of cell, on the stage of the organism development or on an internal or external stimulum.

1.2.1 *Replication of Genetic information*

Because of its double helix structure, genetic information can be **duplicated** to obtain two identical replicas of the original DNA molecule. Once the double helix is split into its two strands, a complementary sequence of nucleobases is created on each stread, producing **two exact copies of the original DNA**. These copies are composed by **an old and a new strand**.

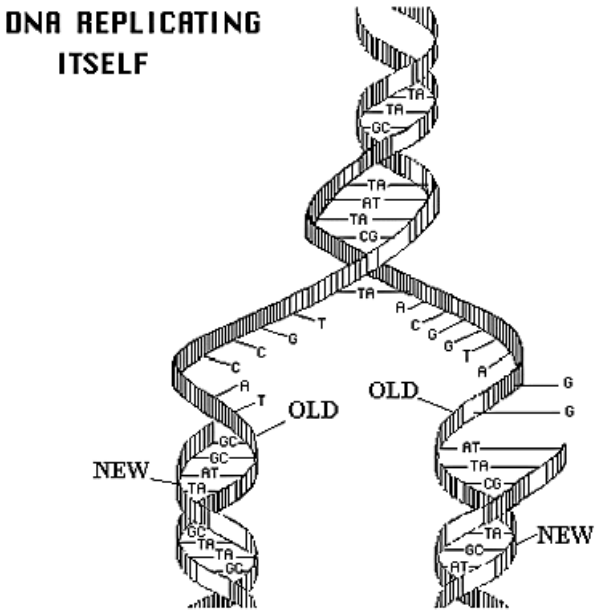


Figure 8: Example of DNA replication.

More in details, DNA replication consists of the following steps:

- The process starts at several particular sequences of nucleobases called *origin points*. These points are **rich in adenine and thymine bases** as their hydrogen bonds are easier to break up. The origin points are found and targeted by **initiator proteins**.
- Once an origin points has been found, a **complex of several types of proteins** called **helycases** unzips the double-stranded

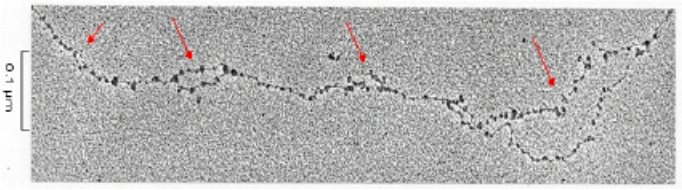


Figure 9: Origin points in a DNA molecule.

DNA, thus obtaining a **replication fork** composed by two branches, the *leading strand template* and the *lagging strand template*.

- The replication itself is carried out by **DNA polymerase**, a family of enzymes that reads the original strand (from 3′ to 5′) and creates a new strand (from 5′ to 3′) adding new nucleotides matched to the template strand. To begin synthesis, a **primer** (short fragment of RNA) is also created and paired with the old strand.

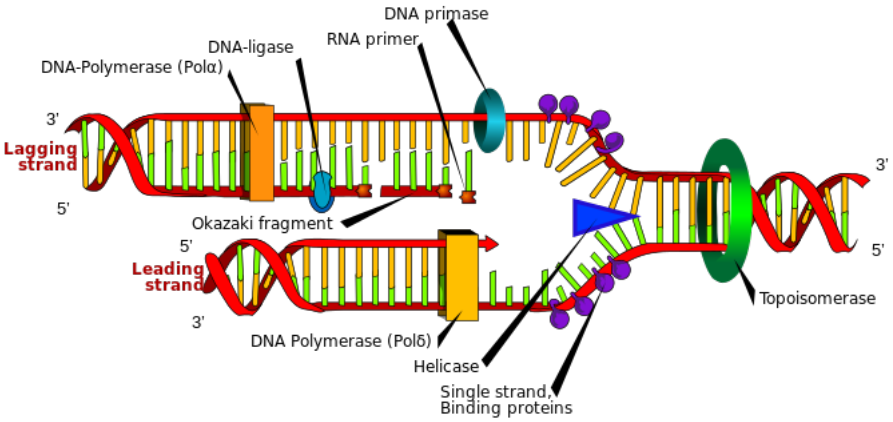


Figure 10: DNA replications and its proteins.

Note that the leading and lagging strands are replicated in different ways. The leading strand is read in the same direction of the growing replication fork: the DNA polymerase reads the leading strand template and adds complementary nucleotides to the nascent leading strand **continuously**. The direction of the lagging strand is instead opposite to that of the growing replication fork: because of this, the lagging strand is synthesized in **short, separated segments** which are then joined together by **DNA ligase**.

1.3 CELLS

The cell is the **structural unit of most living organisms** and can be described as a *chemical factory* enclosed by a semi-permeable membrane. Cells can be either **eukaryotic or prokaryotic** depending on their internal organization: eukaryotic cells are compartmentalized (with a nucleus and organelles with specific functions) while prokaryotic cells are simpler precursors of eukaryotic cells, containing only the cytosol and a plasma membrane. Prokaryotes are single-celled organisms bacteria and archaea) while eukaryotes can be either single-celled or multicellular.

1.3.1 *Prokaryotic cells*

Prokaryotic cells are simpler and smaller, and lack organelles such as the nucleus. DNA is a **single chromosome in direct contact with the cytoplasm**.

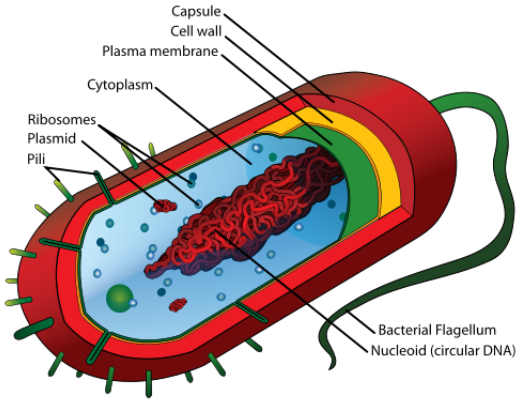


Figure 11: Typical structure of a prokaryotic cell.

A prokaryotic cell has **three regions**:

- On the outside, **flagella and pili** made of proteins facilitate movement and communication between cells;
- Enclosing the cell is the **cell envelope** – usually a plasma membrane plus optional layers (a capsule, for example). It gives rigidity to the cell and separates the cell from the external environment.
- Inside there is the **cytoplasmic region** containing the genetic material, which is freely found in the cytoplasm. Though not

forming a nucleus, the DNA is condensed in a **nucleoid**. Plasmids encode additional genes, such as antibiotic resistance genes.

1.3.2 *Eukaryotic cells*

Typical of animals, plants, fungi, algae and protozoa, eukaryotic cells are usually wider and greater in volume than a typical prokaryote. Their main feature is compartmentalization – the presence of organelles in which specific functions take place.

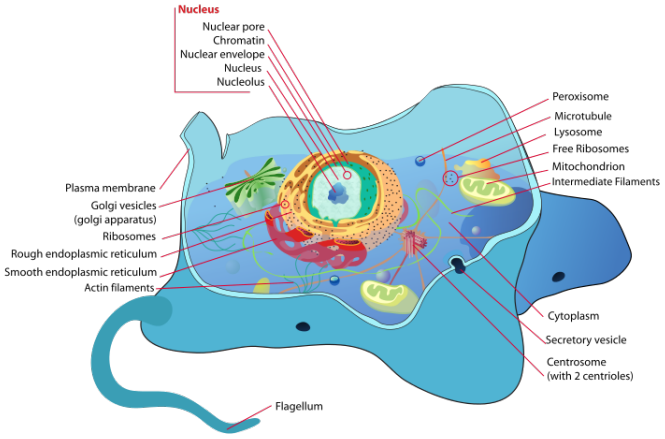


Figure 12: Structure of an animal cell.

The most important organelles are:

- The nucleus, containing the DNA molecule. Note that since space is limited[1] the DNA double helix is packed into highly-organized structures called **chromosomes**. To obtain a chromosome, the double helix is first wrapped around proteins called **histones**, obtaining the nucleosomes; nucleosomes are grouped together to form **chromatin fibres**, which are folded to form large structures called **chromosomes**. Chromosomes are usually composed by two **chromatides**, which are joined together by a **centromere**. Many eukaryotic cells contain pairs of chromosomes and are called diploid. Other cells contain single chromosomes and are called haploid. The human genome consists of 23 pairs of chromosomes.
- The **ribosomes**, responsible for protein synthesis;
- The **mitochrondria**, responsible for energy production (energy is always necessary for reactions).

1 Human DNA is 3m long, a cell nucleus may be 0.006mm!

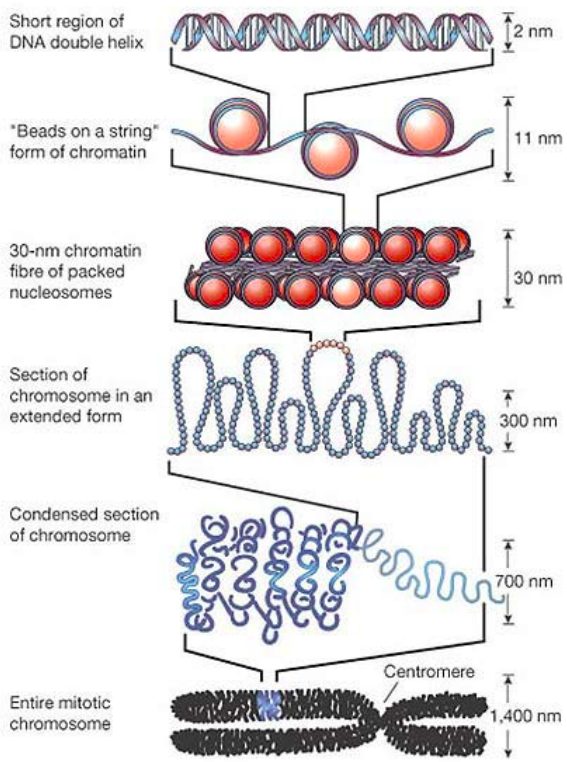


Figure 13: How a chromosome is obtained.

1.4 PROTEINS

Proteins are large macromolecules composed by a **sequence of 20 amino acids**, which are linked together by **peptide bonds**. The average protein is more or less 200 amino acids long but some may contain thousands of them. Proteins perform a **large number of functions** within living organisms, including catalyzing metabolic reactions (enzymes), performing DNA replication, responding to stimuli, transporting molecules from one location to another (hemoglobin), offering structural means and movement (collagen and myosin respectively). They also perform defense and regulation operations (toxins and hormones respectively). The function of a protein depends on **number and sequence of amino acids**; furthermore, because of the chemical and physical properties of the molecules, each sequence organizes itself into a **different tridimensional structure** which determines the specific function of the protein.

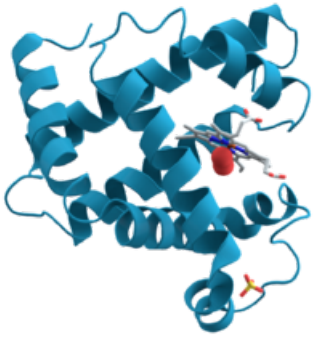
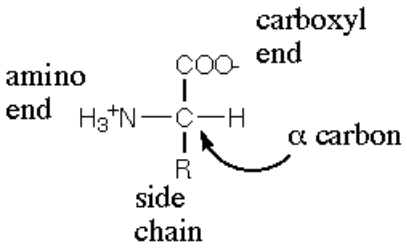


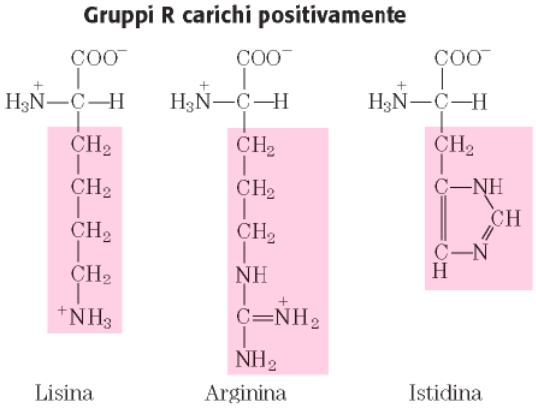
Figure 14: 3-dimensional structure of a myoglobine molecule.

1.4.1 *The amino acids*

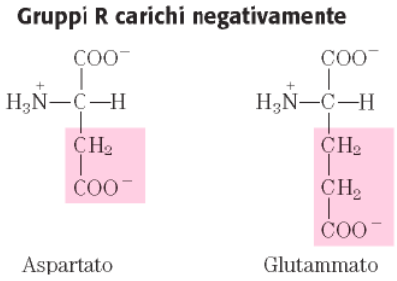
Amino acids consist of a central carbon atom (called the **alpha carbon**) connected to an amino group, a carboxyl group, a hydrogen atom and a side chain. The side chain differs between the different amino acids but the rest is the same:



Depending on the **shapes and chemical groups of the side chain**, amino acids have different properties (hydrophobic/hydrophilic, acidic/basic/neutral, aliphaptic/ aromatic, ...). For example, amino acids with basic lateral chains are:



Amino acids with acid lateral chains are instead:



1.4.2 *Peptide bonds*

Amino acids can be bound by a covalent bond between the amino and carboxyl groups. An example of peptide bond is the following:

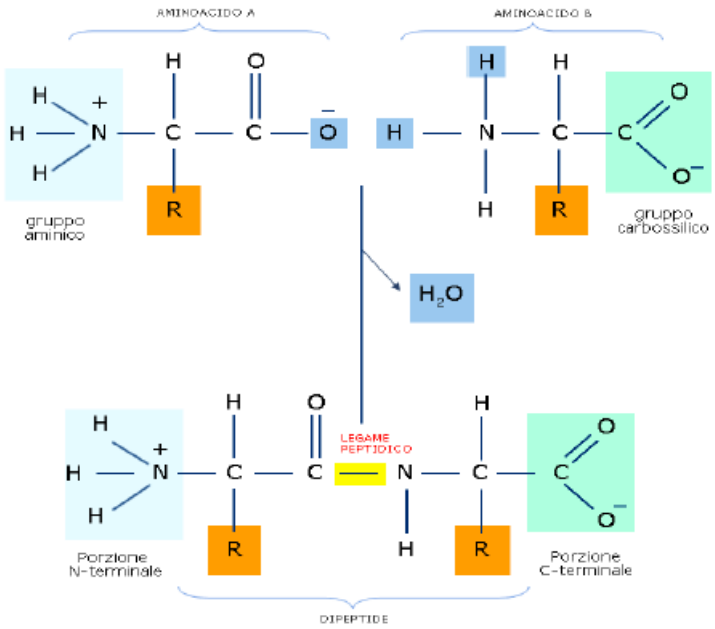


Figure 15: A peptide bond between two amino acids.

1.5 PROTEIN STRUCTURE

Proteins have such a complex structure that it can be divided into **four structural levels**.

1.5.1 *Primary structure*

The primary structure of a protein is simply the **linear sequence of amino acid** in its polypeptide chain. Note that proteins are written in

order **from the amino-teminal end to the carboxy-terminal end**, so `Ala-Cys-Phe` is different from `Phe-Cys-Ala`. Fir example, the first 100 (of 457) amino acids in *hexokinase* are:

> A A S X D X S L V E V H X X V F I V P P X I L Q A V V
> S I A T T R X D D X D S A A A S I P M V P G W V L K Q
> V X G S Q A G S F L A I V M G G G D L E V I L I X L A
> G Y Q E S S I X A S R S L A A S M X T ...

This sequence contains all the information required to determine the higher levels of structure. The linear polypeptide chain folds in a particular arrangement, giving a three-dimensional structure, the information on how to fold is contained in the sequence.

1.5.2 *Secondary structure*

It consists of the local local **spatial arrangement of the protein**; short stretches of the chain fold up to form alpha-helices, beta-sheets and loop. Alpha-helix is a tight rodlike helix formed out of the polypeptide chain.

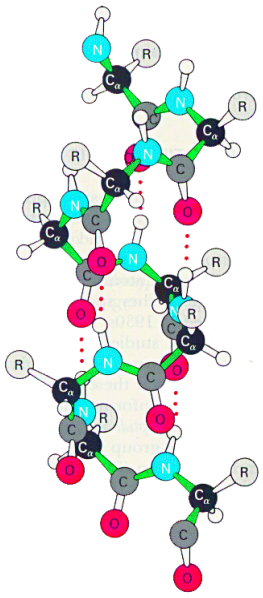


Figure 16: Alpha-helix secondary structure.

The central structure is formed by the polypeptide main chain while the side chains extend out and away from the helix. The CO group of one amino acid (n) is hydrogen bonded to the NH group of the amino acid four residues away ($n+4$). Alpha helices are most commonly made up of hydrophobic amino acids because hydrogen

bonds are generally the **strongest attraction possible** between such amino acids. Another possibile type of secondary structure is the **beta sheet**. They can be parallel or antiparallel, and are formed by bond between amino acids n and $n+4$.

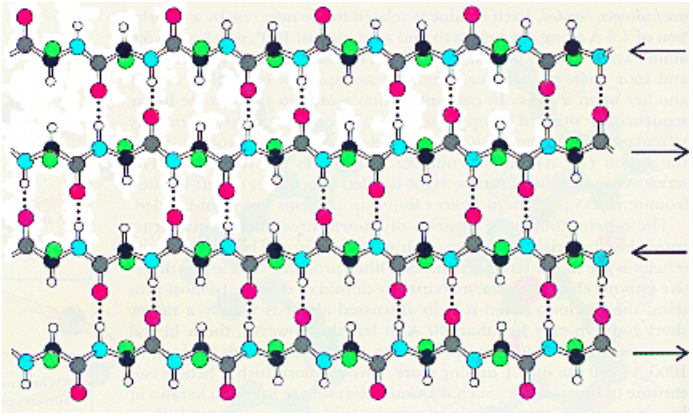


Figure 17: Anti-parallel beta sheet.

Between alpha helices and beta sheets in the protein we find **loop regions**. They are less regular although they may still have some structure. Loops tend to end up on the outside of the proteins, when the protein folds up to form its full 3d structure (tertiary structure), so they are exposed to water, the loop regions thus tend to be **rich in hydrophilic residues** (amino acid side chains). Loops often are **binding sites for other molecules**.
As for their representation, secondary structures are usually represented in **ribbon diagrams** where a coiled ribbon represents an alpha helix, an arrow ribbon a beta sheep and a thin string a loop.

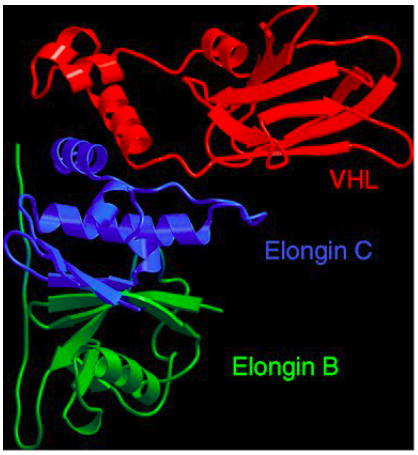


Figure 18: Secondary structures and ribbon representation.

1.5.3 *Tertiary structure*

The tertiary structure is the way in which the **secondary structure elements fit together** in the full 3d structure. Proteins, in fact, fold up so that amino acids that are distant in the linear sequence may be very close in space, thus forming a domain.

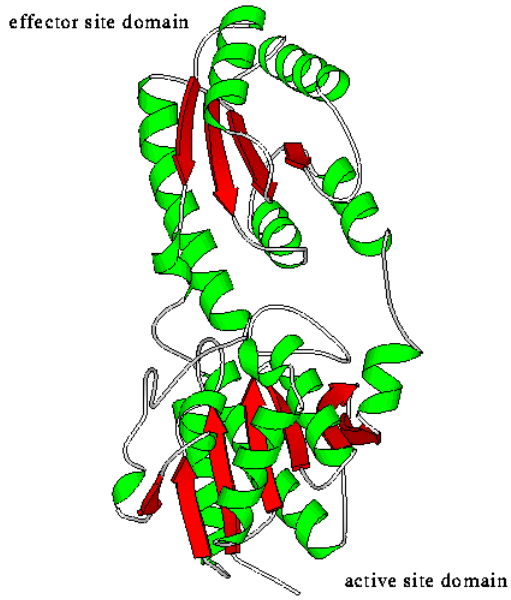


Figure 19: Secondary structures and ribbon representation.

1.5.4 *Quaternary structure*

A protein may consist of more than one linear chain molecule; the quaternary structure determines how these chains fold around one another. Insulin is a good example of a protein with more than one polypeptide chain:

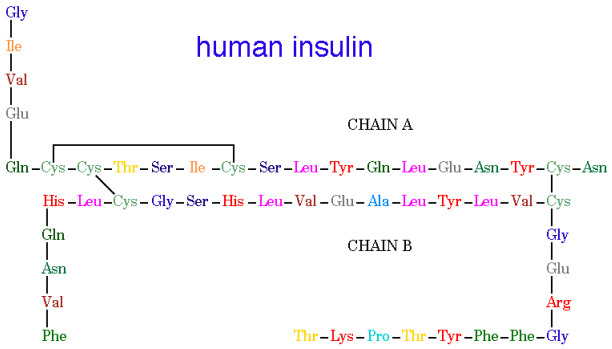


Figure 20: Secondary structures and ribbon representation.

In the quaternary structure, the B chain is wrapped around the A chain, which forms a compact central unit. In fact the structure of insulin is more complicated: it forms hexamers with six insulin molecules (A and B chain) around two zinc ions and 6 water molecules. Another example is *hemoglobin*, composed by 4 sub-units.

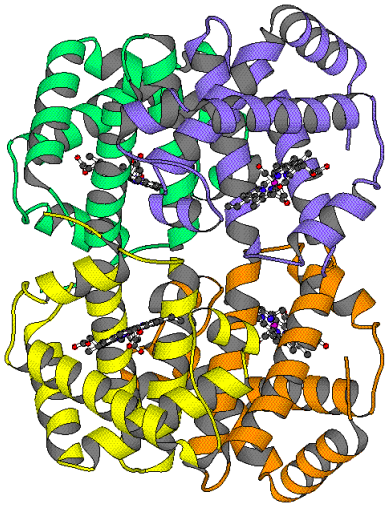


Figure 21: Secondary structures and ribbon representation.

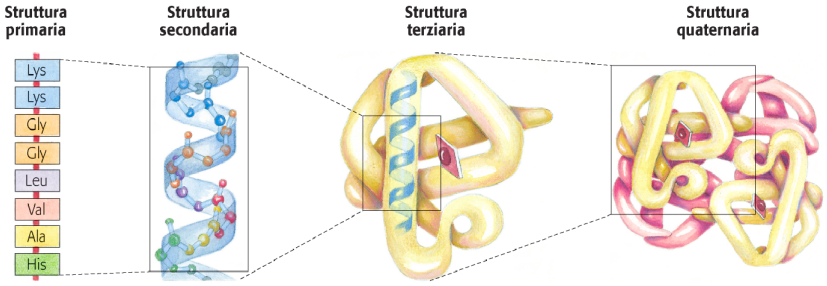


Figure 22: Possible organizations of proteins.

Note that each amino acid has a three-letter code and a single letter code:

Alanine	Ala A	AsparagiNe	Asn N
Cysteine	Cys C	Proline	Pro P
Aspartic AciD	Asp D	Glutamine	Gln Q
Glutamic Acid	Glu E	ARginine	Arg R
Phenylalanine	Phe F	Serine	Ser S
Glycine	Gly G	Threonine	Thr T
Histidine	His H	Valine	Val V
Isoleucine	Ile I	Tryptophan	Trp W
Lysine	Lys K	Tyrosine	Tyr Y
Leucine	Leu L		
Methionine	Met M		

1.6 THE CENTRAL DOGMA OF MOLECULAR BIOLOGY

The central dogma of molecular biology is an explanation of the **flow of genetic information** within a biological system. In an oversimplied version, it states that '**DNA makes RNA (*transcription*) and RNA makes protein (*translation*)**': the genetic information is first transcripted in RNA and then translated into proteins or, in other words, the amino acid sequence of proteins is determined by the nucleotide sequence of the DNA molecule.
We've said before that a gene is a portion of DNA that control a certain hereditary feature: it usually corresponds to a single mRNA, which is later translated into a specific protein. In eukaryotes, the genes have their coding sequences (exons) interrupted by non-coding sequences (introns). In humans, genes consistute only about 2-3% of DNA, while the rest is junk (or **non-coding nucleotides**).

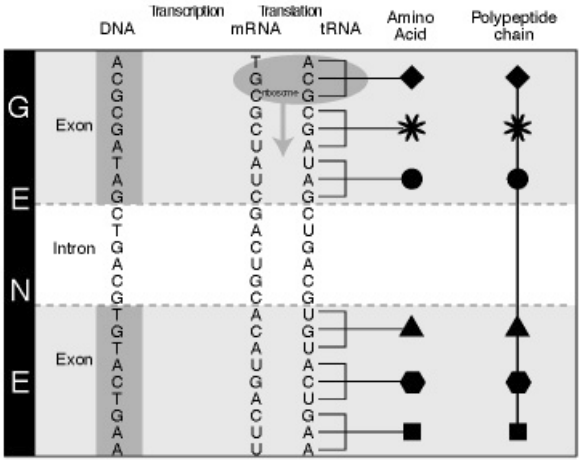


Figure 23: Secondary structures and ribbon representation.

1.7 RNA

RNA is a macromolecule implicated in various biological roles, the main being the **expression of genes**. Its chemical structure is very similar to that of DNA but **differs in three main ways**:

- The sugar-phosphate group has a different sugar, **ribose (R)** instead of deoxyribose (D);
- Nucleotides use **uracil (U)** instead of thymine (T);
- The structure is almost always a **single-stranded molecule** and not a double helix.

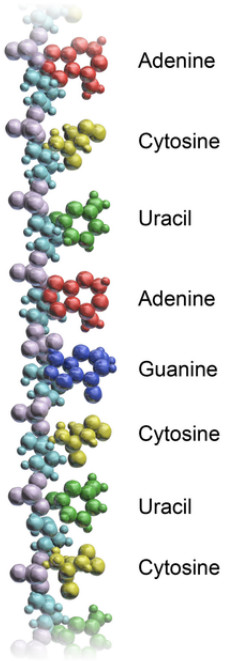


Figure 24: Secondary structures and ribbon representation.

RNA comes in **different forms** including **mRNA** (messenger RNA: it is transcribed from DNA and translated into protein) and **tRNA** (transfer RNA: a functional molecule used during translation).

1.7.1 *Transcription*

Transcription is the first step of gene expression, in which a particular portion of DNA is used to produce a strand of mRNA. It consists of three stages:

1. **Initiation**. The RNA polymerase enzyme binds to a **promoter site** on the DNA and unzips the double helix, breaking up the hydrogen bonds.

2. **Elongation**. RNA polymerase adds matching RNA nucleotides to the complementary nucleotides of one DNA strand (thymine in DNA is replaced by uracil in RNA). The polymerase moves along the DNA in the $3'$ to $5'$ direction, extending the RNA $5'$ to $3'$, exactly as in DNA replication.

3. **Termination**. Specific sequences in the DNA signal that the transcription is terminated. When the RNA polymerase encoun-

ters one of them, the RNA strand is released from the DNA and the double helix is zipped up again.

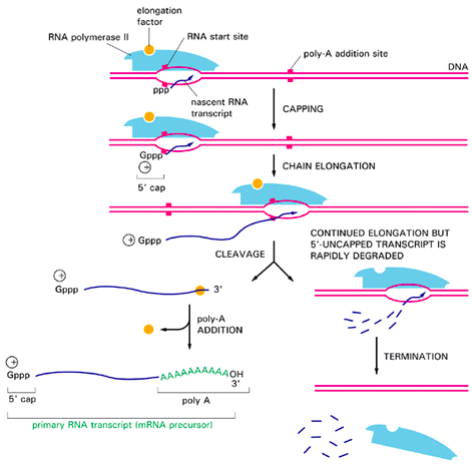


Figure 25: DNA transcription to obtain a RNA strand.

In **eukaryotes**, the original transcript from the DNA is called **heavy nuclear RNA (hnRNA)** and it contains transcripts of both introns and exons. Introns are removed (**splicing**) to produce messenger RNA (mRNA) and the ends of the RNA molecule are processed.

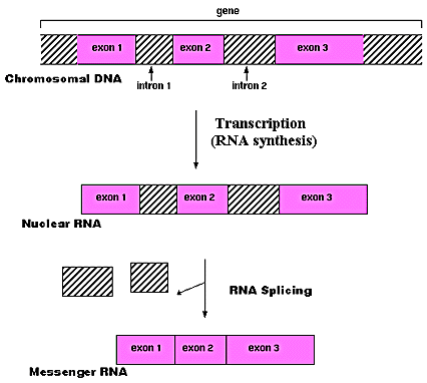
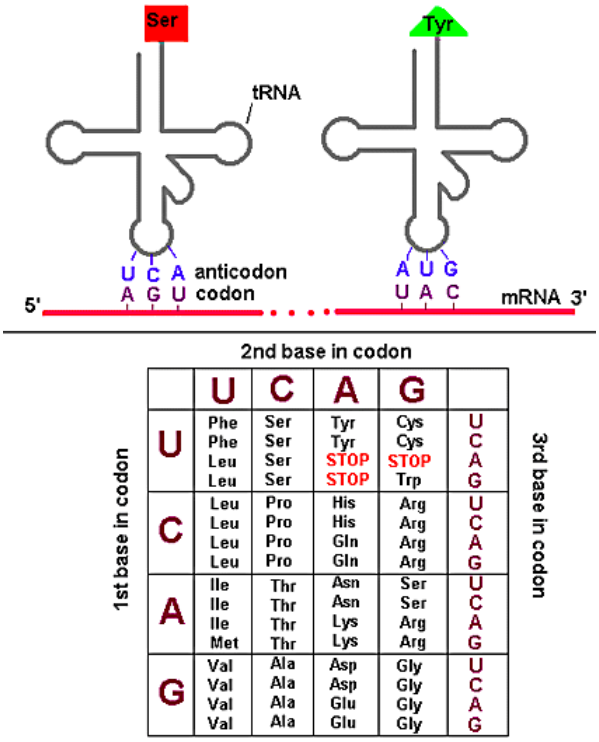


Figure 26: The whole process from DNA transcription to mRNA splicing.

1.7.2 *RNA Translation*

Translation is the biologic process in which **cellular *ribosomes* produce proteins**. In translation, messenger RNA (mRNA)—produced by transcription from DNA—is decoded by a ribosome to produce a specific amino acid chain, or polypeptide. The polypeptide later folds into an active protein and performs its functions in the cell.

Translation is done by using special molecules called **transfer RNAs** (tRNAs): their codon recognizes triplets of nucleotides (codons) and their anti-codon produces the corresponding amino acid. This way, a specific genetic triplet is encoded into the corresponding amino acid.



	U	C	A	G	
U	Phe	Ser	Tyr	Cys	U
	Phe	Ser	Tyr	Cys	C
	Leu	Ser	STOP	STOP	A
	Leu	Ser	STOP	Trp	G
C	Leu	Pro	His	Arg	U
	Leu	Pro	His	Arg	C
	Leu	Pro	Gln	Arg	A
	Leu	Pro	Gln	Arg	G
A	Ile	Thr	Asn	Ser	U
	Ile	Thr	Asn	Ser	C
	Ile	Thr	Lys	Arg	A
	Met	Thr	Lys	Arg	G
G	Val	Ala	Asp	Gly	U
	Val	Ala	Asp	Gly	C
	Val	Ala	Glu	Gly	A
	Val	Ala	Glu	Gly	G

Figure 27: Transfer RNAs at work and the corresponding genetic codes.

More in details, tRNA structure is the following (notice both the codon and the anticodon):

But how is it possible to translate sequences of 4 nucleobases (DNA) into sequences of amino acids? Since DNA is an *alphabet* of 4 simbols, to obtain 20 amino acids we need **strings of at least length 3**. In fact, with length 2 we can code 16 different items (4^2):

AA AC AG AT CA CC CG CT GA GC GG GT TA TC TG TT

With length 3, instead, we can code $4^3 = 64$ different items, more than what we actually need.

AAA AAC AAG AAT ACA ACC ACG ACT ...

The genetic code also has three STOP sequences that are interpreted by the tRNA as the end of the proteine. ATG also represents the beginning of the region coding for the protein.

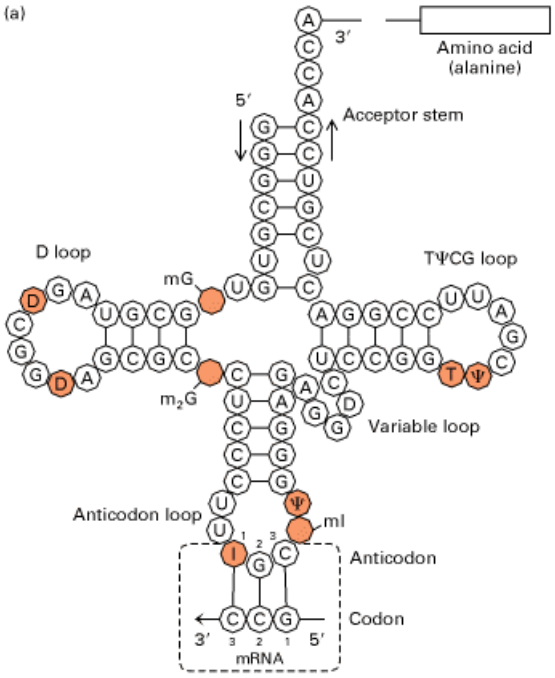


Figure 28: Detailed structure of transfer RNA.

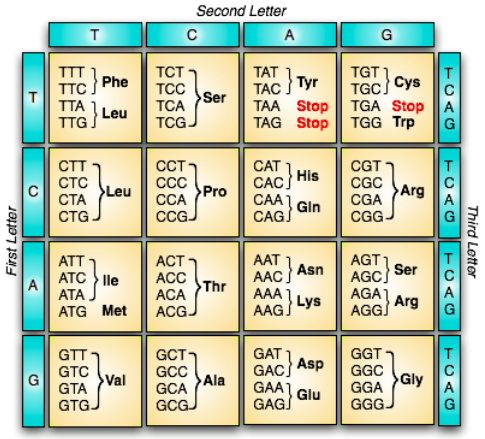


Figure 29: Detailed structure of transfer RNA.

1.8 ORF: OPEN READING FRAME

How many ways can we read a DNA string? Depending on the starting point results may vary, as this example shows:

Translation, therefore, starts when tRNA position itself for reading the genetic message in the mRNA; the first tRNA bonds to a start codon (AUG) on the mRNA, then each tRNA adds an amino acid to a growin protein chain.

Note that transcription and translation are different in **prokaryotes and eukaryotes**. In prokaryotic cells, which have no nuclear com-

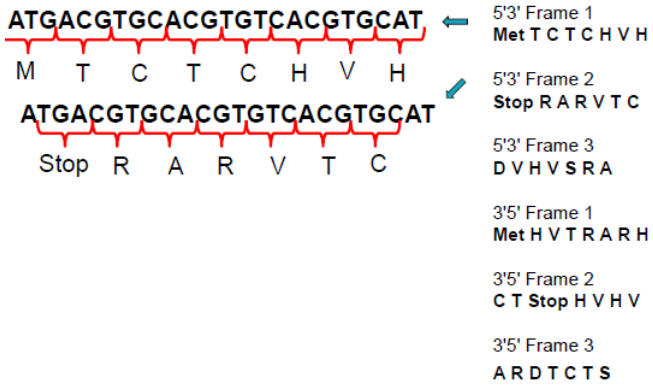


Figure 30: Different ways of reading the same DNA nucleotides sequence.

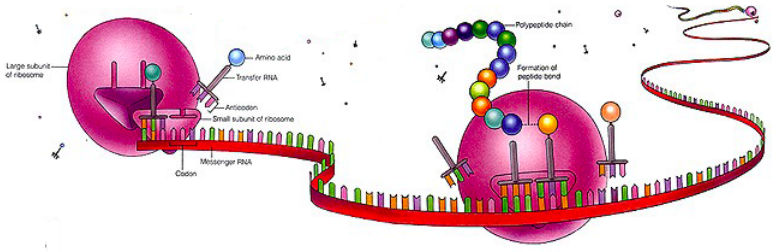


Figure 31: tRNA attaches itself on the `AUG` sequence of the mRNA strand produced during the DNA traduction.

partment, the transcription and translation may be linked together **without clear separation**:

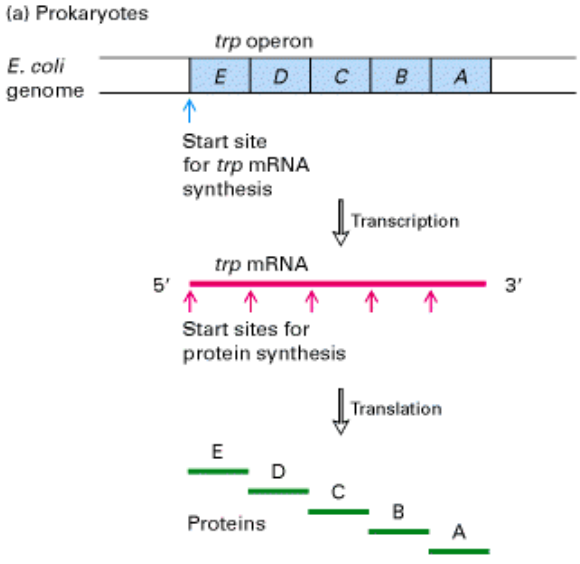


Figure 32: Transcription and translation in a prokaryote cell.

In eukaryotic cells, the site of transcription (the cell nucleus) is usually separated from the site of translation (the cytoplasm), so the mRNA must be transported out of the nucleus into the cytoplasm, where it can be bound by ribosomes.

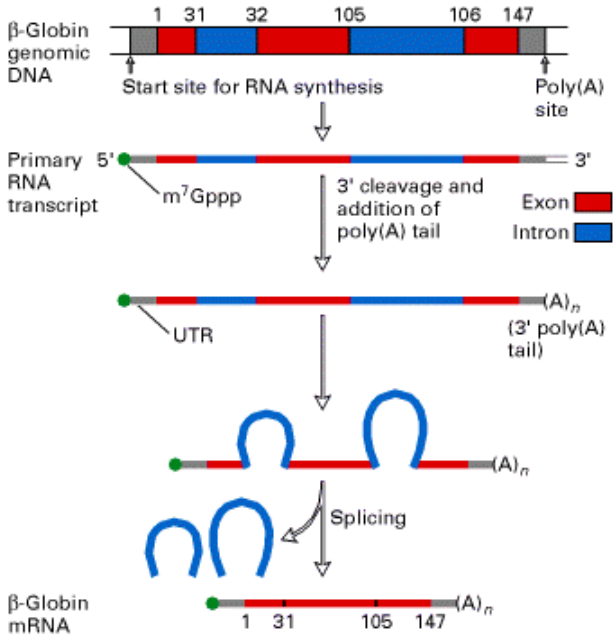


Figure 33: Transcription and translation in a eukaryote cell.

1.8.1 *Eukaryote alternative splicing*

Alternative splicing is a process where the exons of the RNA produced by transcription of a gene (pre-mRNA) are reconnected in multiple ways during RNA splicing. The resulting different mRNAs may be translated into different proteins; thus, **a single gene may code for multiple proteins**. This greatly increases the diversity of proteins that can be encoded by the genome.

Finally, remember that all cells in our bodies have the same DNA. How then do different cells have very different characteristics? The answer is that not all the genes in the genome are being transcribed and translated into proteins in every cell. We say that genes which are transcribed and translated are **expressed in the cells**. Gene expression is controlled in different cells via functional protein molecules.

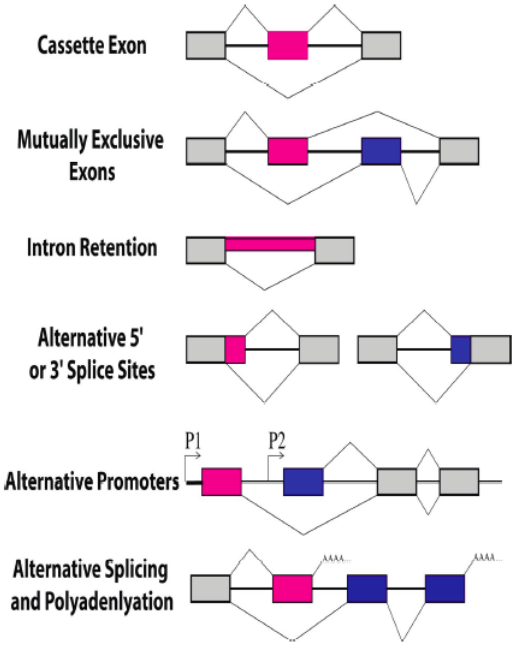


Figure 34: Alternate splicing.

1.9 WHY BIOINFORMATICS?

The first reason is that sequencing produces an enormous amount of data on genomes (man, rat, yeast, ...); Proteomics is very important for new medicines and it also produces great amounts of data. Need of DBs and tools to store, organize and query biological data.

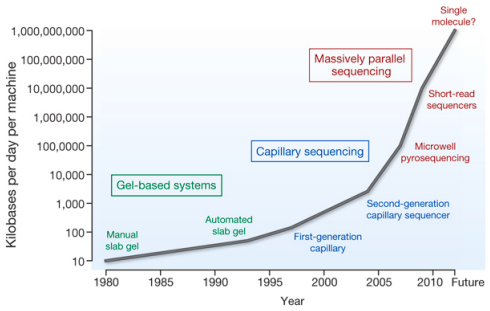


Figure 35: Sequencing data growth.

Furthermore, DNA and proteins can be seen as **strings of symbols**: since deseases may depend on genes or can be cured by proteins, we need algorithms and tools to search, compare, analyse and manipulate strings.

Interestingly, the replication of DNA is **accurate but not error free**: occasionally **mutations take place**.

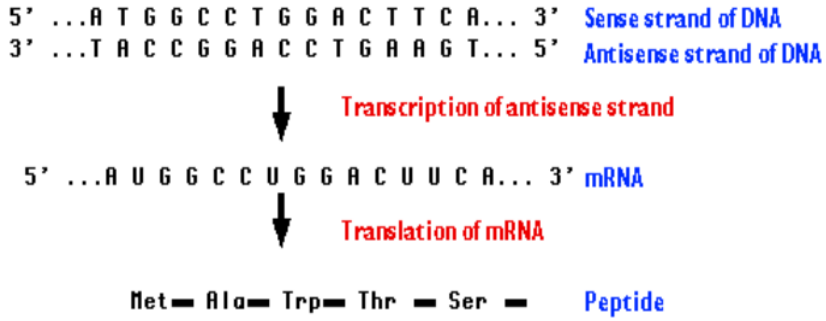


Figure 36: Translation and transcription as processes on strings.

```
ATCGGGCCATATCGAAATGG
ATCGCGCCATTTCGAAATGG
```

In humans, on average we have 3 mutations per genome per cell division. This error is not always a problem though, as it's what allows evolution to take place, since it leads to the generation of novel genotypes. Evolution is in fact based on heredity, mutation and selection and can be studied by comparing and relating genes or even whole genomes in different organism. Again, we need algorithm and tools to manipulate trees and graphs. Algorithms are also needed to simulate folding and to visualize and analyse proteins in 3D:

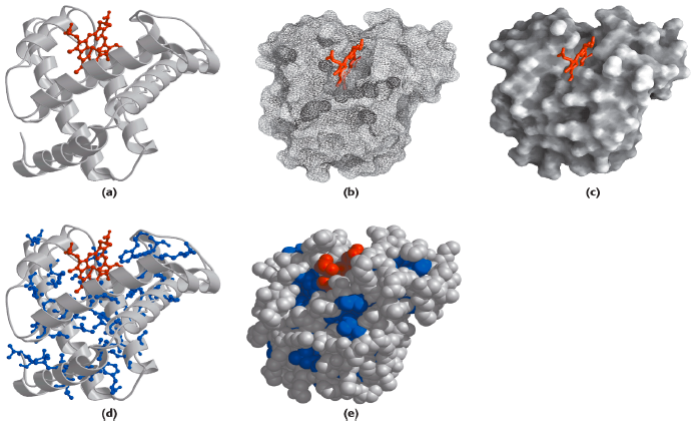


Figure 37: 3D representation of a protein.

Finally, we need techniques and tools to model biological systems in an interactive way – for example to represent dinamically metabolic pathways such as the following:

Formally, bioinformatics is defined as the application of computer science and information technology to the field of biology and medicine. Its purpose is generating new knowledge of biology and medicine, and improving and discovering new models of computation (e.g. DNA

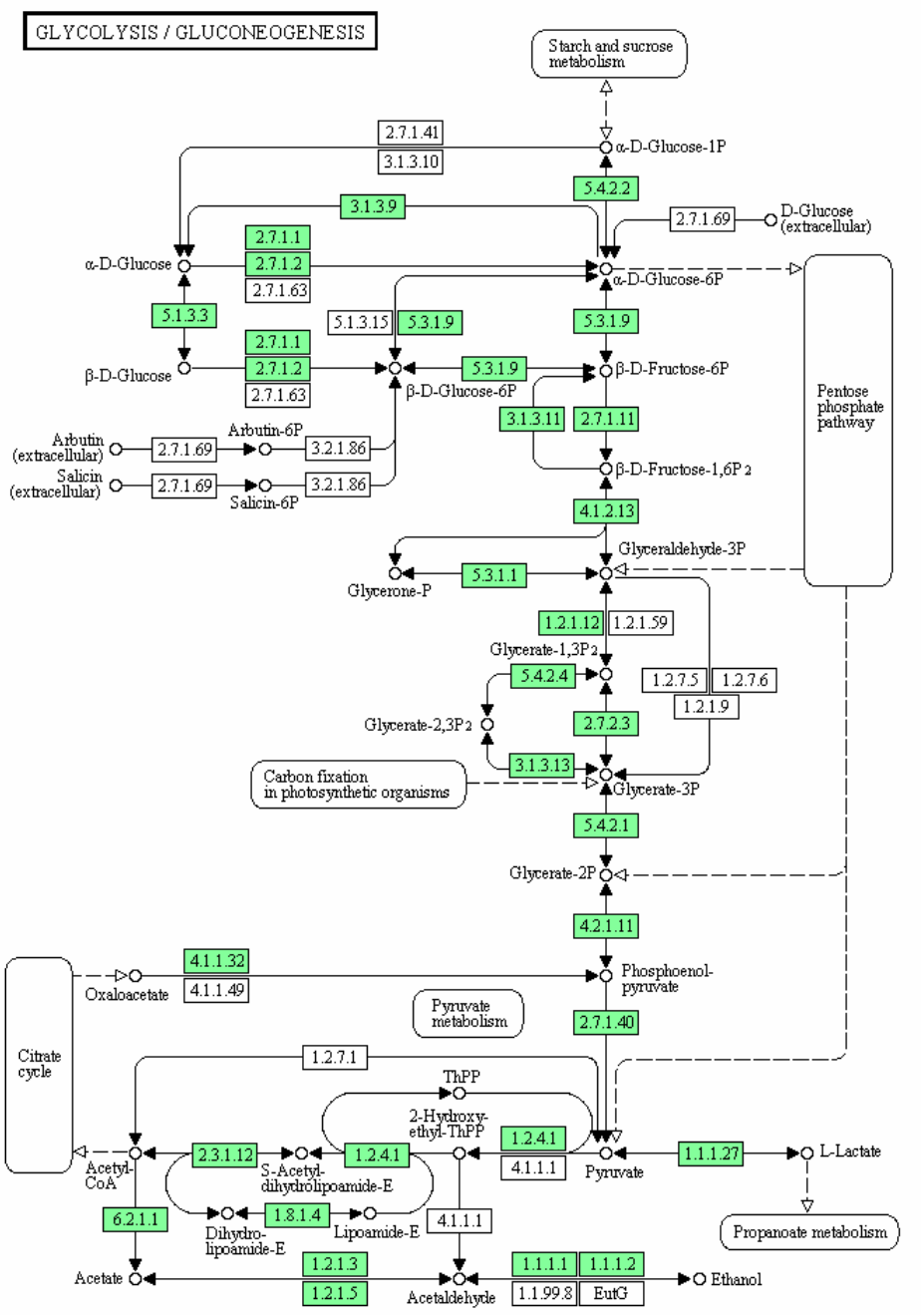


Figure 38: Interactive scheme of metabolic pathways.

computing, neural computing, evolutionary computing, immunocomputing, swarm-computing, cellular-computing, ...). It deals with many other fields: algorithms, databases, data mining, artificial intelligence, discrete mathematics, statistics, modeling and simulation, ... Note that bioinformatic applications generally have to deal with enormous amount of data with redundant parts, errors and incomplete infor-

mation; rules usually have many exceptions and heuristics are also applied. It is necessary to have a deep knowledge of the problem and work together with with biologists, statisticians, ...

2

EXACT PATTERN MATCHING

Exact pattern matching is a problem of highly practical importance, as it arises in many applications such as word processors, browsers, operative systems and, of course, bioinformatics. But what does it mean to solve an exact pattern matching problem? Suppose we have a text T and a pattern P: solving an exact pattern matching problem means **finding all occurrences of** P **in** T. For example, if $P = aba$ and $T = bbabaxababay$ then P occurs in T starting at location 3, 7, 9, with overlapping between 7 and 9.

2.1 WHAT IS A STRING?

A **string** S of length n is a **sequence of** n **symbols**. Given S, we can tell:

- **Prefixes and suffixes**: $S[1..h]$ represents the prefix of h symbols, $S[h..k]$ the suffix of $k - h + 1$ symbols.
- **Substrings**: $S[i..j]$, and contains $j - i + 1$ symbols.
- **Symbols**: $S(h)$ represents the h-th symbol in S (notice the different parentheses).

For example, *california* is a string, *lifo* is a substring, *cali* is a prefix, *fornia* is a suffix, *c* is the first symbol.

2.2 EXACT PATTERN MATCHING: NAIVE APPROACH

How can we find all the occurrences of the pattern inside a text? The first method naively aligns the left end of P with the left end of T and compares the symbols of T with the symbols of P: if no match is found P is **shifted one place to the right** and the comparison starts again. For example:

```
abcabaabcabac
abaa <- no match, shift
 abaa <- no match, shift
```

```
abaa <- no match, shift
  abaa <- match found!
```

Formally (n is the text length, m is the pattern length):

```
for s:= 0 to n-m do
   if P[1..m] = T[s+1..s+m] then
   print "pattern P occurs from position s+1";
```

Despite being very simple this approach cannot be used with long texts and patterns, as its **complexity becomes exponential** ($O(n^2)$).

2.3 EXACT PATTERN MATCHING: FINITE AUTOMATA

To speed up the naive methods we can **pre-process the pattern** by building its finite automata. Using the pre-processed pattern we **strongly reduce the work done while searching for occurrences of** P **in** T: in fact, each symbol in T is inspected only once, reducing the overall complexity of the algorithm to $O(n)$. The complexity of the pre-processing phase is instead $O(m|\epsilon|)$.
To understand how this approach works, suppose we want to find all the occurrences of $P = nano$ in the text $T = bananano$. The steps are to preprocess the pattern are:

1. **Defining the states**. Since we must recognize if we're in the middle of a match, we need to know how much of $nano$ we've already seen: the finite automata is thus composed by **as many states as the prefixes of** $nano$**, trivial cases included** (empty string, full pattern).

2. **Defining transitions**. Suppose we've already seen nan: if we find an o we get to state $nano$, if we get an a we get to the state na, if we get a n we get to the state n, if we get anything else we get to the state $empty$. Similar reasoning is performed for every state we have. Note that there's always a "linear" transition between the states.

Once P is pre-processed, T is searched: if $T = bananano$ we get the sequence of states empty, empty, empty, "n", "na", "nan", "na", "nan", "nano", "nano", "nano". Since we end in state "nano", we have a match.

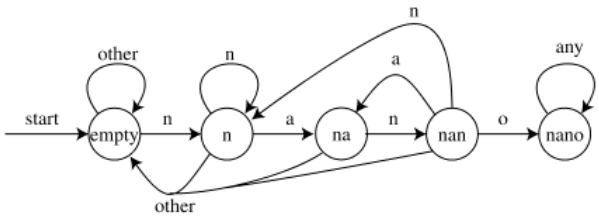


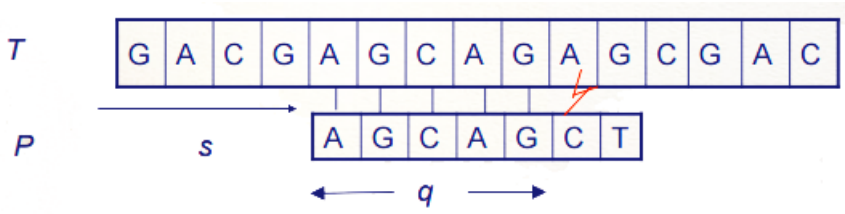
Figure 39: Finite automata for nano.

2.4 EXACT PATTERN MATCHING: KNUTH-MORRIS-PRATT ALGORITHM

The most famous and efficient algorithm for exact pattern matching was developed by Knuth, Morris and Pratt. Much like with finite automata, it's **based on two steps**:

1. **Pre-processing** P. Pre-process the pattern ($O(m)$) to obtain useful information for the second step;
2. **Scanning** T. Scan the text ($O(n)$; in the worst case $O(n+m)$) using the previous information.

For example, suppose we're at a certain point of the comparison between the following pattern and text:



The KMP algorithm differs from the naive approach on a crucial point: what happens when we have a mismatch. Consider the example: up to the second G everything matches but C is not matched with A. Now:

- The naive approach shifts the pattern one position to the right and compares again P with T: however, there's no guarantee that this costly comparison will produce a match between P and T.
- KMP instead suggests shifting P of several positions to the right in order to **minimize the number of comparisons** and reduce the cost of the algorithm. More in details, since we already

know that P and T matches up to G, we can determine that +1 and +2 shifts are useless.

As we can see, to determine the shifting value we need to explore the matches between T and the prefixes of P: for each position q of T we find the longest prefix of P matching up to q and then shift by q (the greatest shift we can safely perform). To ease this approach we need to use the *prefix function*.

2.4.1 *The Prefix function*

Given a pattern P of size m, the prefix function π receives in input a value i between 1 and m and returns the **length of the longest non-trivial prefix of** P **which is also a suffix of** P_i.

$$\pi : \{1, \ldots, m\} \to \{0, \ldots, m-1\} \tag{1}$$

...where $\pi[q] = \max\{k : k < q \& P_k] P_q\}$.
Formally, the prefix function is:

```
Compute-Prefix-Function(P):
    m  ← length[P]
     π[1]  ← 0
    k  ← 0 prefix length
    for q  ← 2 to m
        do while k>0 e P[k+1] != P[q]
            do k  ←  π[k]
        if P[k+1] = P[q]
            then k  ← k+1
         π[q]  ← k
    return  π
```

As we can see, in the initialization phase we keep track of the length of P, m and we initialize the first prefix value, $\pi[1]$. Next, starting from $q = 2$ and up to $q = m$, we compute the prefix value for each prefix of P, storing the result into the array π. The rules to compute the prefix value are simple: if k is non-trivial but we cannot extend it, we inductively reuse the content of the table that we've already stored, otherwise we extend k.

2.4.2 *Application of the algorithm*

Suppose we have the patter `P = bbba`. Let's initialize the values: $\pi[1] = 0$, $k = 0$, $q = 2$, $m = \mathtt{length}(P) = 4$.

Step 1. $q = 2, k = 0$. Since k is not > 0 the internal `while` cycle is not considered and we simply try to extend the value of k by evaluating of $P[k+1] = P[q]$. Since $P[k+1] = P[1] = b$ and $P[q = 2] = b$, then k is incremented to 1 and $\pi[q = 2] = 1$.

Step 2. $q = 3, k = 1$. This time $k > 0$ so we need to evaluate if $P[k+1] \neq P[q]$. Since $P[k+1] = P[2] = b$ and $P[q = 3] = b$ the previous condition is false and we exit the `while` cycle. The `if` condition is true so we increment k to 2 and set $\pi[3] = 2$.

Step 3. $q = 4, k = 2$. Again, $k > 0$ so we need to evaluate if $P[k+1] \neq P[q]$. Since $P[k+1] = P[3] = b$ and $P[q = 4] = a$ the previous condition is true: we stay inside the `while` cycle and decrement k to 0. Therefore $\pi[q = 4] = 0$.

P	b	b	b	a
i	1	2	3	4
$\pi[i]$	0	1	2	0

2.4.3 *Example of computation of prefix function*

Suppose we want to compute the prefix values of $P = ATATATATCA$, with $m = 10$. First we initialize an array with 10 locations:

```
   A T A T A T A T C A
i: 1 2 3 4 5 6 7 8 9 10
```

Next, we compute $\pi[i]$: it's the length of the longest, non-trivial prefix of P which is also a suffix of p_i. The results are the following:

- $\pi[1] = 0$ since the table is always initialized this way.
- $\pi[2] = 0$ since the longest prefix of P which is also a suffix of $P_2 = AT$ does not exist.
- $\pi[3] = 1$ since the longest prefix of P which is also a suffix of $P_3 = ATA$ is `A` (length 1).

- $\pi[4] = 2$ since the longest prefix of P which is also a suffix of $P_4 = ATAT$ is AT (length 2).
- $\pi[5] = 3$ since the longest prefix of P which is also a suffix of $P_5 = ATATA$ is ATA (length 3).
- $\pi[6] = 4$ since the longest prefix of P which is also a suffix of $P_6 = ATATAT$ is ATAT (length 4).
- $\pi[7] = 5$ since the longest prefix of P which is also a suffix of $P_7 = ATATATA$ is ATATA (length 5).
- $\pi[8] = 6$ since the longest prefix of P which is also a suffix of $P_8 = ATATATAT$ is ATATAT (length 6).
- $\pi[9] = 0$ since the longest prefix of P which is also a suffix of $P_9 = ATATATATC$ does not exist.
- $\pi[10] = 1$ since the longest prefix of P which is also a suffix of $P_{10} = ATATATATCA$ is A (length 1).

P	A	T	A	T	A	T	A	T	C	A
i	1	2	3	4	5	6	7	8	9	10
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

To understand how the algorithm works, let's consider a particular case, $\pi[8] = 6$. What does it mean that when i is 8 its prefix function is 6? Interestingly, it means that if in the next iteration we have a mismatch then 6 is the length of the maximum prefix without mismatches. Of course, each value is also inductively linked to all the previous prefixes, which are **valid but not maximal**. Therefore, this algorithm is particularly useful with patterns with many repetitions (σ is small), exactly as in DNA strands. This is exactly why we have an internal cycle in the algorithm: if k is non-trivial ($k > 0$) but I cannot extend the match ($P[k+1] \neq P[q]$), I can reuse the previous largest prefix thus "decrementing" the value of k. If everything goes well and matches are found, instead, k is "incremented". This behaviour is clear if we consider the previous table: $\pi[i]$ is incremented and then decremented.

2.4.4 *Complexity*

As for the complexity of the function, we have:

```
Compute-Prefix-Function(P):
    m  ← length[P]
     π[1]  ← 0
    k  ← 0 prefix length
    for q  ← 2 to m    -> O(m)
        do while k>0 e P[k+1] != P[q]     -----.
            do k  ←  π[k]                     |
        if P[k+1] = P[q]                      | -> O(1)
            then k  ← k+1                     |
         π[q]  ← k     ----------------------'
    return  π
```

The operations withing the `for` cycle have a constant cost, while the `for` cycle itself inspects m values of `q`, thus a linear complexity of $O(m)$. Space-wise, we need to store an array with m cells so complexity is again $O(m)$.

2.4.5 *KMP algorithm*

As said before, KMP uses the prefix function to pre-process the pattern and **maximixe the shift of the pattern along the text**: this way, T is read without backtracking (online algorithm) and P is not re-read from the beginning at each shift. The algorithm is very similar to that of the prefix function:

```
KMP-Matcher(T,P)
n ←  lenght[T]
m ←  lenght[P]
π ←   COMPUTE-PREFIX-FUNCTION(P)
q ←  0 number of matches
for i ←  1 to n
     do while q>0 e P[q+1] ≠  T[i] mismatch
         do q ←   π [q]
     if P[q+ 1]=T[i] match
         then q ←  q+ 1
     if q= m
         then print "The pattern occurs with shift" i-m
              q ←  π [q]
```

As we can see, in the **initialization phase** we store the lengths of the text (n) and the pattern (m); next, we compute the prefix function and set the number of matches `q` to 0. Then for each element of

the text we evaluate if we have a mismatch or a match between the current symbols:

- If there's a match the "match counter" q is incremented;
- If theres a mismatch ($P[q+1] \neq T[i]$), we inductively **extract the length of the largest reusable prefix of the pattern** from the π matrix;

When the number of matches is equal to the length of the pattern m we print the shift, $i - m$.

2.4.6 *Complexity*

As for the complexity, we have the cost of COMPUTE-PREFIX-FUNCTION plus the cost of the `for` cycle: $O(n+m)$. The operations within the `for` cycle have an ammortized cost of $O(1)$.

2.4.7 *Example (#2)*

Suppose we have the following T and P:

```
T: bbbabbbba (length n=9)
P: bbba      (length m=4)
```

At first we initialize several values: $m = 4, n = 9, q = 0$, the prefix table is that of the previous exercise:

P	b	b	b	a
i	1	2	3	4
$\pi[i]$	0	1	2	0

Next, we apply the algorithm:

- **Step 1, q=0**. We enter the `for` loop. Since $q = 0$ we skip the while and check if $P[q+1] = T[1]$. Since they're the same q is incremented to 1.
- **Step 2, q=1**. Since $q > 0$ and $P[q+1] = T[2]$, we increment q to 3.
- **Step 3, q=2**. Since $q > 0$ and $P[q+1] = T[3]$, we increment q to 4.

- **Step 4, q=3**. Since $q > 0$ and $P[q+1] = T[3]$ we increment q to 4. But the length of the pattern is 4 so the pattern occurs with shift $i - m = 4 - 4 = 0$. q is then returned to $\pi[q] = 0$.

- **Step , q=0**. We enter the `for` loop. Since $q = 0$ we simply check if $P[q+1] = T[4]$: since it is so q is increased to 1.

- **Step 6, q=1**. Since $q > 0$ and $P[q+1] = T[i]$ q is increased to 2.

- **Step 7, q=2**. Since $q > 2$ and $P[q+1] = T[i]$ then q is increased to 3.

- **Step 8, q=3**. Since $q > 3$ and $P[q+1] \neq T[i]$ we iteratively decrease the value of q from 3 to $\pi[q] = 2$. Now, $q = 2 > 0$ but $P[2+1] = T[7]$: since there's a match q is increased to 3.

- **Step 9, q=3** Since $q > 0$ and $P[q+1] = T[i]$, q is increased to 4. But $q = m$ so the shift is $9 - 4 = 5$.

3

SUFFIX TREES AND GENERALIZED SUFFIX TREES

A suffix tree is a data structure capable of **representing a string and its internal suffixes**. In pattern matching suffix trees are used to **preprocess the text** and search patterns in it. The first definition of suffix tree and the algorithm to build it was introduced by Weiner in 1973, but it went unnoticed for many years due to its complexity. In 1995 **Ukkonen** defined a simpler algorithm to build suffix trees in linear time with respect to the string length n. Thanks to their simplicity, suffix trees have been used in many contexts.

3.1 WHAT IS A SUFFIX TREE?

Suppose we have a string S with lenght n. Its corresponding suffix tree τ has:

- as many leaves as suffixes is S, n;
- non-trivial internal nodes (they must have at least two children);
- labels on the branches. Each branch has a unique label (on the path from the root to a leaf we can read a suffix of S). Note that the branches starting from a node must start with different symbols.

An example of suffix tree for $S = mammao$ is the following:

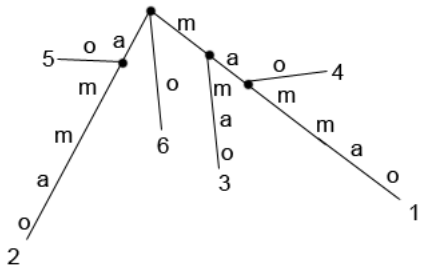
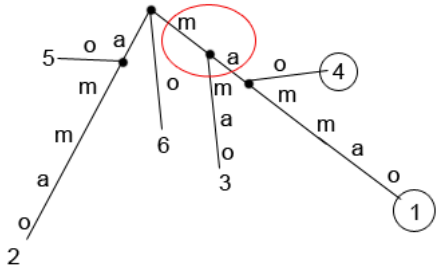


Figure 40: Suffix tree of `S=mammao`.

Suffix trees are **used to solve *exact* pattern matching problems**: to find a pattern P in a string S it is enough to find all the suffixes of S starting with pattern P. For example, if $S = mammao$ and

$P = ma$, since there are two suffixes of S starting with ma then we have two matches. Incidentally, the positions of these suffixes are also the positions of the pattern in the string, 1 and 4. To sum up, pattern matching with suffix trees works as follows:

1. Given a text T and a pattern P, apply Ukkonen algorithm to build the suffix tree of the text T;
2. Find every unique path in T containing P (if no path is found then there is no match);
3. The positions of these paths correspond to the positions of an occurrence of P in T.



3.1.1 *Complexity and difficult suffix trees*

As for the complexity of finding matching patterns with suffix trees, it **depends on the problem** at hand:

- If we're only interested in the existence of at least a match (yes/no question) then the complexity is $O(m)$, where m is the size of the pattern.
- If we want to find the number and the position of the matches, then we need to traverse the suffix tree k times, where k is the number of occurrences of P in T. Therefore complexity is $O(k)$.
- If we need to reach the leaves of the tree, we apply any traverse algorithm (for example, **depth-first**) with complexity $O(n)$.

To sum up, the **overall complexity of preprocessing and searching is** $O(m+n+k)$[1]. Note that **there may be situations where we**

1 A way to reduce this complexity **when looking for a single occurrence of P in T** is to associate to each node v the corresponding number of a leaf of the subset rooted in v. This way, complexity is $O(n+m)$

can't build a suffix tree: for example, if a suffix S_1 is the prefix of another suffix S_2 the suffix tree cannot be constructed because S_1 is completely overlapped by S_2. This usually happens when the last symbol of a string appears elsewhere in it: to avoid situations such as this one it is enough to add a **terminator symbol** \$ at the end of S_2.

3.2 BUILDING SUFFIX TREES: NAIVE ALGORITHM

How can we build a suffix tree for a certain text S? The naive approach produces a suffix tree in an exponential time, $O(n^2)$, where n is the length of S. The algorithm is:

1. Enumerate each suffix (longest $\rightarrow$ shortest) starting from the whole text T.
2. Build the first arc with the longest suffix $S[1..n]\$$ and label the leaf with 1.
3. Add the remaining branches. Remember: if we already have a branch starting with symbol "a" we cannot have another branch starting with the very same symbol; note also that the final symbol, \$, is **always a new branch**.

For example, the suffix tree of `mamma$` is:

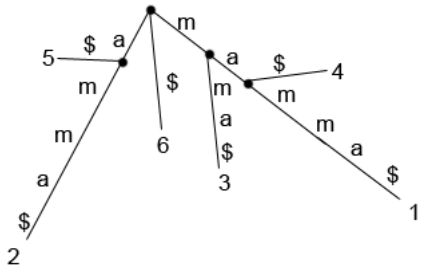


Figure 41: Suffix tree of `S=mamma$`.

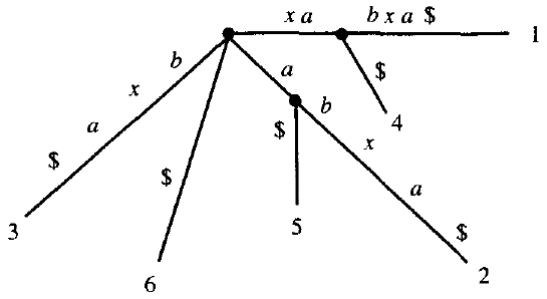
As we can see the algorithm **loops on the suffixes** of S, from 2 (since the first suffix is always the first to be used) to n – a fact that explains **why complexity is quadratic**. At each step we extend the current branch and/or create a new branch from the root and/or add an internal node plus a branch. As for spatial complexity, it depends on the number of nodes of the suffix tree: in the worst case we have n leaves + $n-1$ internal nodes.

3.3 BUILDING SUFFIX TREES: UKKONEN ALGORITHM ($O(n^3)$)

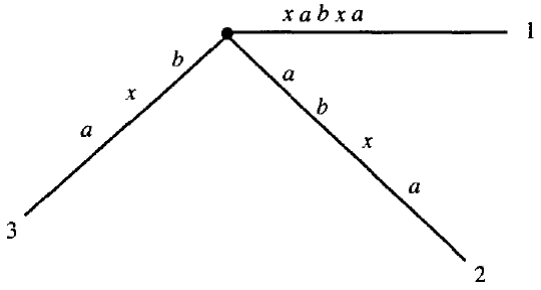
The first algorithm proposed by Ukkonen is actually worse than the naive one, at least complexity-wise. Still, it can be optimized later on. The idea behind the Ukkonen's algorithm is straight-forward: build a series of partial suffix trees for the prefixes of S, one on top of the other; the last one is then transformed into the actual suffix tree of S.

> If $S[1..n]$ then build $\tau_1, \ldots, \tau_n$ where τ_1 is for $s[1..1]$, τ_2 is for $S[1..2]$, ...

Note that partial suffix trees are suffix trees with no terminator symbol. They can be obtained from a suffix tree by removing \$, removing arcs with no labels, removing nodes with less than two children. As an example, if the suffix tree of `xabxa` is...



...then the partiaul suffix tree of the same string is:



Generally speaking, Ukkonen's algorithm is divided into m phases: in phase $i+1$ the tree τ_{i+1} is constructed from the previous τ_i. Each phase is characterized by $i+1$ extensions, one for each suffix of the current string $S[1..i+1]$. In extension j of phase $i+1$ the algorithm extends the current branches of the partial suffix tree plus adds new internal nodes with additional branches, following rules that we will see in the following pages. The algorithm is:

```
build Ti : build an arc labelled S(1) and associate the
```

```
number 1 to the leaf;

for i from 1 to n-1 do
   begin phase i+1: build Ti+1
   for j from 1 to i+1 do
      begin extension j
      search the end of the path with label S[j..i] in the
      tree τ_i from the root; if necessary, extend
      the path with the symbol S(i+1) so that S[j..i+1]
      is in the tree;
end;
```

Note that the algirhtm starts by building the first partial suffix tree, τ_1, which is always a tree with a single symbol in one branch):

Figure 42: τ_1 for the text $T = mammamia\$$.

Then, for the remaining symbols of the text, we consider all the suffixes (from j to $i+1$) and perform the extension, adding the new symbol $i+1$ to the present partial tree with label $S[j..i]$, thus having $S[j..i+1]$ in the tree. Note that we have **two cycles**: the first produces the n partial suffix trees, the second extends each partial suffix tree. This explains why complexity is $O(n^3)$: producing each partial suffix tree costs n times the overall cost of each extension, n^2. All in all, τ_n is built in $O(n^3)$. But what does it mean to "extend"?

3.3.1 *Rules for extension*

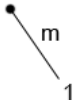
Suppose we have $S[j..i]$, a suffix of $S[1..i]$. During the extension phase the algorithm finds the end of $S[j..i]$ in the current tree and extends it to be sure the suffix $S[j..i+1]$ is in the tree. Extension is done according to **one of the following rules**:

1. **Actual extension of the label**. If $S[j..i]$ is a path from the root to **leaf** i, add the symbol $S(i+1)$ in the last arc of the path as a new leaf.
2. **Do nothing**. If $S[j..i]$ is inside another path in the tree and the symbol after $S(i)$ is $S(i+1)$, the $S[j..i+1]$ is already in the current tree so we do nothing.

3. **Create a new branch**. Suppose we find S[j..i] inside the tree, but the next symbol is not $S(i+1)$. In this case a new leaf edge starting from the end of S[j..i] must be created and labeled with character $S(i+1)$. A new node will also have to be created there if S[j..i] ends inside an edge. The leaf at the end of the new leaf edge is given the number j.

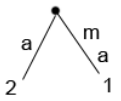
3.3.2 *Example: how to build a suffix tree using Ukkonen's algorithm*

Suppose we have the string $S = mammamia\$$. The steps are: **Phase 1**. The first partial suffix tree is τ_1, for the prefix $S[1..1] = m$. We have only one suffix, m itself, so τ_1 contains a root plus a branch labelled m and numbered 1.



Phase 2. We compute τ_2 for the prefix $S[1..2] = ma$. It has two suffixes:

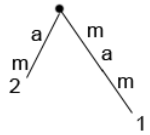
- ma: since there's already a branch containing m we extend it with a a new leaf, a;
- a: since there's no branch starting from the root with an a, we add a new branch to the tree. Since a is in position 2, the branch is numbered 2.



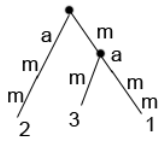
Phase 3. We compute τ_3 for the prefix $S[1..3] = mam$. We have three suffixes:

- mam: since there's already a branch containing ma, we extend it with a new leaf, m;
- am: since there's already a branch containing a, we extend it with a new leaf, m;
- m: since there's already a branch containing m there's nothing to do.

Phase 4. We compute τ_4 for the prefix of $S[1..4] = mamm$. We have four suffixes:

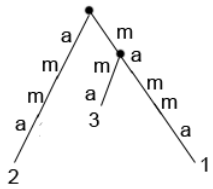


- mamm: since there's already a branch containing mam we extend it with another leaf, m;
- amm: since there's already a branch containing am we extend it with another leaf, m;
- mm: this suffix is partially contained in mam, therefore we add a node after the first m and a leaf m. This branch is enumerated with 3.
- m: this suffix is contained within mamm so there's nothing to do.



Phase 5. We compute τ_5 for the prefix $S[1..5] = mamma$. We have five suffixes:

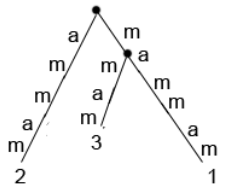
- mamma: we extend the existing mamm with a new leaf, a;
- amma: we extend the already existing amm with a;
- mma: we extend the already existing mm with a;
- ma: contained within mamma, there's nothing to do;
- a: contained within amma, there's nothing to do.



Phase 6. We compute τ_6 for the prefix $S[1..6] = mammam$. We have six suffixes:

- mammam: we extend the already existing mamma;

- ammam: we extend the already existing amma;
- mmam: we extend the already existing mma;
- mam: completely overlapping with mammam, there's nothing to do;
- am: overlapping with amma, there's nothing to do;
- m: overlapping with mammam, there's nothing to do.

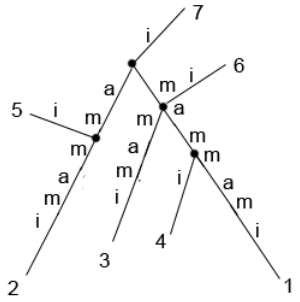


Phase 7. We compute τ_7 for the prefix $S[1..7] = mammami$. We have seven suffixes:

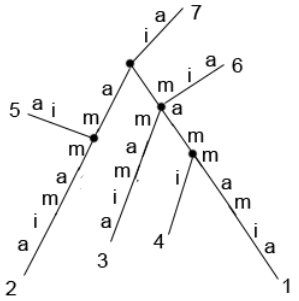
- mammami: we simply extend the already existing mammam;
- ammami: we simply extend the already existing ammam;
- mmami: we simply extend the already existing mmam;
- mami: this suffix does not appear in any branch of the tree so we add a new node after mam and a new branch with leaf i. This leaf has number 4.
- ami: this suffix does not appear in any branch of the tree so we add a new node after am and a new branch with leaf i. This leaf has number 5.
- mi: this suffix does not appear in any branch of the tree so we add a new node after m and a new branch with leaf i. This leaf has number 6.
- i: this suffix does not appear anywhere so we add another branch to the root of the tree. This branch, with leaf i, is numbered 7.

Phase 8.. We compute τ_7 for the prefix $S[1..7] = mammamia$. We have eight suffixes:

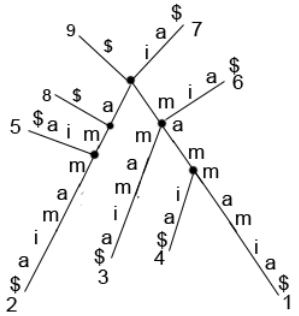
- mammamia: we simply extend the already existing mammami;
- ammamia: we simply extend the already existing ammami;



- mmamia: we simply extend the already existing mmami;
- mamia: we simply extend the already existing mami;
- amia: we simply extend the already existing ami;
- mia: we simply extend the already existing mi;
- ia: we simply extend the already existing i;.
- a: overlapping with ammamia so there's nothing to do.



The last step consists in simply **adding the** $ **symbol** after each leaf, plus a new branch with the label $, numbered 9.



3.4 FIRST OPTIMIZATION: SUFFIX LINK

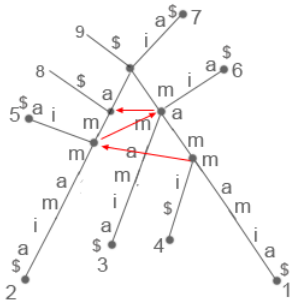
As we've said before, Ukkonen's algorithm has complexity $O(n^3)$:

```
build T1
for i=1 to n-1                          O(n)
     for j = 1 to i+1                   O(n)
          search for prefix T[1..j]     O(n)
          extend it with T(i)           constant
```

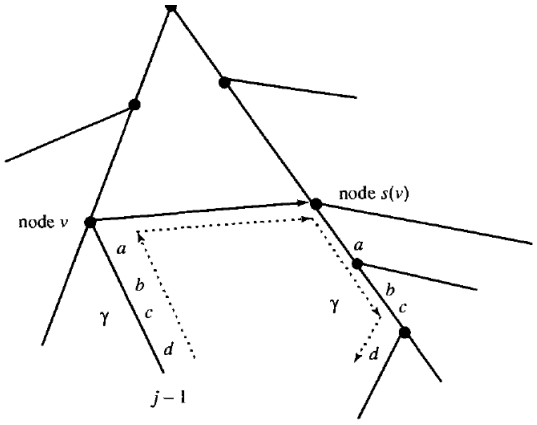
The first optimization **reduces this complexity to** $O(n^2)$. The idea behind starts from a simple consideration: traversing the whole tree to find the suffixes we need to extend is a costly operation. We could, instead, traverse the tree following its border, starting from the latest symbol we've added to the tree. To do this, every internal node of the suffix tree must be **equipped with a suffix link**, created in a **costant time** while building the suffix tree. To understand **how nodes are linked**, suppose we have:

- a node v whose label is $x\alpha$ (x is a single character and α is a string, which can also be empty);
- a node whose label is simply α, called $s(v)$.

Then there's a suffix link from v to $s(v)$ (referred as $(v, s(v))$). When α is the empty string ϵ then the suffix link goes from v to the root. Now, suffix links create a **path within the suffix tree** called **suffix chain**:



In the Ukkonen algorithm every new internal node created during an extension phase has a suffix link to the next extension. To update the suffix tree with a new extension, then, we **follow the suffix chain** without going up to the root of the suffix tree. For example, suppose we want to extend $S[j..i]$ to $S[j..i+1]$: we start at the end of the string $S[j-1..i]$ in the current tree, walk up at most one node to either the root or a node v that has a suffix link from it; if γ is the label of that edge and v is not a root, we traverse the suffix link from v to $s(v)$, then we walk down the tree from $s(v)$ following a path labelled γ to



the end of S[j..i]; there, we extend the suffix to S[j..i + 1] according to the extension rules.

Overall, this optimization has the following pros and cons:

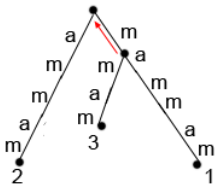
- **Pros**: complexity is reduced to $O(n^2)$ since we're moving from the first extension to the next without traversing the whole suffix tree;
- **Cons**: building an inner node is more comples, since a suffix link is always created.

3.5 SECOND OPTIMIZATION: THE SKIP/COUNT TREE

Using the suffix chain we walk down from node $s(v)$ along a path labelled γ; this walk along γ takes time proportional to $|\gamma|$, the number of characters of the path. A simple trick, called *skip/count*, can reduce the traversal time to make it**proportional to the number of nodes on the path**, reducing complexity to $O(n)$ for each phase of the Ukkonen algorithm.
The algorithm works by jumping from an internal node to another, keeping track only of:

- the size of the skipped suffix;
- the first symbol of the suffix itself, for branching reasons.

This way it's possible to avoid the whole traversing of the labels: during the next extension we will easily find the the correct branch to extend (using the first symbol) and the exact point where to add the symbol (right after the size of the skipped part). More in details, suppose we have the following structure with a suffix link and appropriate labels:



The algorithm works as follows:

- Start from the first linked node $s(v)$, in this case the root;
- Compare the first symbol of the suffix we want to extend with the first symbol of the branches starting from $s(v)$;
- Select the correct branch and jump to its first internal node;
- Select again the current branch comparing the k-th symbol of the suffix we want to extend and the k-th symbols of the suffixes starting from the internal node; select the correct branch.

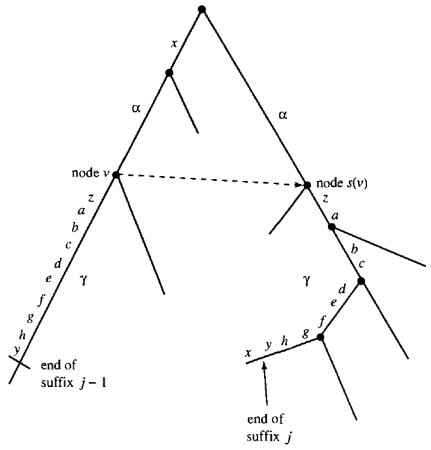
Once we've reached the correct branch, we perform the extension.

```
h = |γ|;
γ' = label of a single outgoing arc from s(v)
(the one labelled with the first simbol of |γ|);
if γ' < h then begin k = |γ|' + 1;
    jump to the next node; skip
    follow the arc starting with the k-th symbol in γ; end;
γ' = remaining string to be traversed;
while | γ'| < h do begin jump to the next node; skip
    modify k, h and |γ|' ; end;
| γ '| ⩾ h
reach the h-th symbol in |γ|';
```

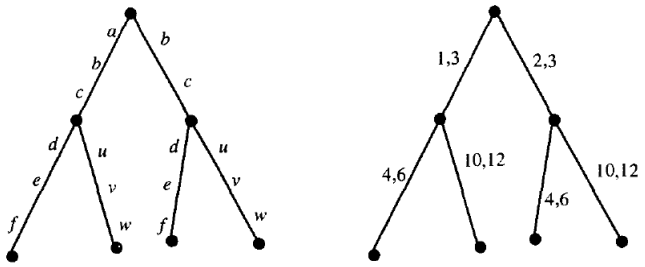
Skip/count moves from a node to another in constant time, so time depends from the number of traversed nodes, not from the number of symbols. Note that if the depth of a node v is the number of nodes in the path from the root to v, the if there's a suffix link from v to $s(v)$ then the depth of v is at most 1+ the depth of $s(v)$.

3.6 THIRD OPTIMIZATION: LABEL COMPRESSION

To further reduce the complexity of Ukkonen's algorithm, we need to recognize that any algorithm dealing with a *labelled* suffix trees will never be less than quadratic (because the suffix tree is represented



with space complexity $O(n^2)$). To reduce time, it is **necessary to reduce space** and compress the representation of the tree. Instead of using substrings for the labels, we can use pair of indexes: $S[i..j]$ becomes (i, j):



Note that this **labeling is not unique**: the edge with label $2, 3$ could also have been labeled $8, 9$. This way, the representation of the suffix tree is linear and **does not depend on the size of the text**.

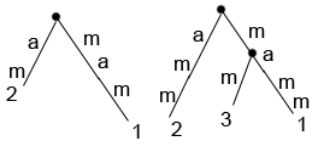
3.7 FOURTH OPTIMIZATION: MODIFICATION OF PHASE i + 1

The fourth optimization exploits the **extension regularities**. It can be proved, in fact, that there are some **correlations between extensions**.

3.7.1 *Correlation between extensions 1 and 2*

The first regularity is between extensions 1 and 2: during the creation of the partial suffix trees there is always a sequence of type 1 extensions (addition of a new leaf to the current branch) followed by a sequence of type 2 extensions (creation of internal nodes). This is clear if we consider what happens in the costruction of `mammamia`: we

extend branch 1 with a type 1 extension and create a new internal node with a type 2 extension; etc.



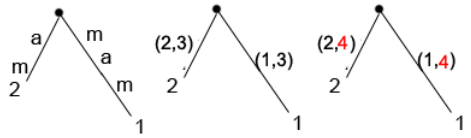
3.7.2 *Chains of extensions 3*

Interestingly, if we perform an extension 3 (do nothing), the same extension will be applied until the end of the phase. We can see it in the sixth phase: after extension 3 (mam overlaps with $mammam$ so there's nothing we can do) we only have extensions 3 (am overlaps with $amma$ and m overlaps with $mammam$).

To sum up, **extensions have a fixed order**:

1. One or more extensions 1 (addition of a leaf); the *last* extension 1 is the actual starting point of each phase.
2. One or more extensions 2 (changing the structure of the suffix tree by adding an inner node plus a branch); it's always inbetween extensions 1 and 3.
3. At least one extension 3 (overlapping suffixes so there's nothing we can do). Only extension 3 from then on. The *first* extension 3 is the ending point of each phase: once we find an extension 3 there's nothing else we need to do.

Now, since we're working with a succint representation of the suffix tree (see the third optimization) every extension 1 can be **performed implicitly** (modified phase $i+1$): it is enough to represent to increase an index:



Therefore Ukkonen's algorithm must keep track only of two indexes:

- the **index** i of the **latest extension 1** performed in the previous phase; it's the starting point for any following extensions;

- the index $*$ corresponding to the **first extension** 3.

The only extensions explicitly performed – extensions 2 – are between these two indexes, from $j_i + 1$ up to the first extension 3 $j*$; if we don't have extensions 3 then the phase will perform extensions 2 up to j_{i+1}. This way, the algorithm **performs no repetitions of work**, hence a linear algorithm.

Using suffix link, skip/count, label compression and this modification, Ukkonen's algorithm has a **limited amount of actual work to perform** and therefore requires only $O(n)$ to build $\tau_1..\tau_n$. Finally, τ_n is transformed into the suffix tree for the string S in time $O(n)$ by adding the new symbol \$, therefore performing another phase of Ukkonen's algorithm to build τ_{n+1}.

3.8 APPLICATIONS OF SUFFIX TREES

Suffix trees are useful not only for ***exact*** **pattern matching** where we're interested in finding a specific strings. They can also be used in other contexts:

- **Finding different strings of a certain length**. How can we find how many different sequences of a certain size k are present in my text? Suppose, for example, that $k = 10$: it is enough to find every path (sized k) starting from the root or, in other words, to traverse the suffix tree labels starting from the k^{th} symbol. To get where are each specific substring is in the text, we can traverse again the suffix tree and get all its occurrences.

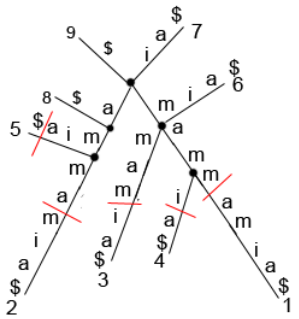
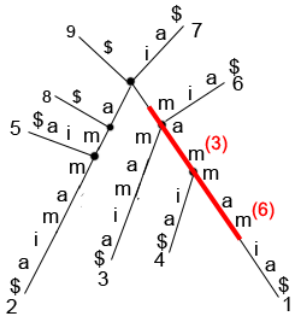


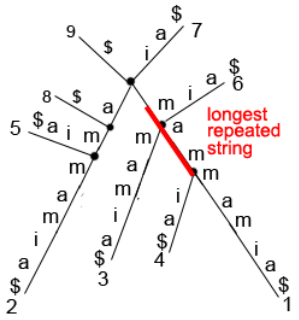
Figure 43: Every string with length $k = 4$.

- **Tandem *repeats* and tandem *arrays***. Suppose we have a string α: how can we find adjacent repetitions α^i, where i can be either 2 (tandem repeats) or > 2 (tandem array)? This is a very

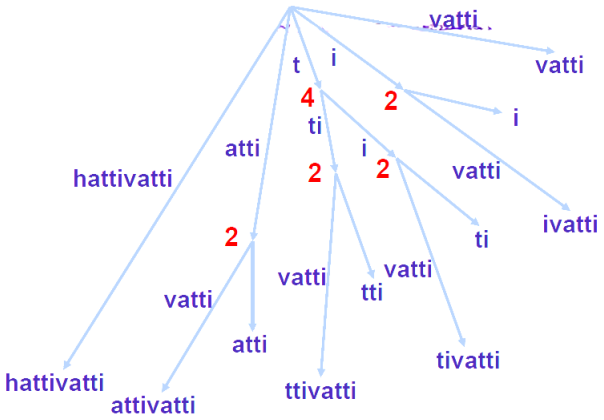
important problem in DNA analysis since an adjacent repetition of nucleotides may represent the starting point of a gene. Suppose we know the length h of the repeated string: it means that the occurrences of this string should be repeated at distance h. To solve the problem we look for a string containing many repetitions of the string we're studying: if the index associated to each repeated substring is a multiple of h then we've found a tandem repeat or array depending on the number of repetitions.



- **Longest repeated substring**. Interestigly, the longest repeated substring always ends at the internal node which is farthest from the root (the **deepest node in the tree**). This means that to find the longest repeated substring we need to find the the deepest node in suffix tree and then get the **path label from root to that deepest internal node**.



Suffix trees can also be used for *approximate* pattern matching problems, where a small number of errors – i.e. mutations – are allowed. There are also techniques for **discovering motifs** – short substrings with an associated property – statistically by exploring the suffix tree. The DNA analysis often focuses on searching of motifs, unusual strings that can be more frequent or rarer than their usual distribution in the DNA. In this case a suffix tree is used to count these motifs, just like in this example:


vatti
vatti
t
i
4
2
i
ti
atti
i
2
2
vatti
hattivatti
ivatti
2
ti
vatti
vatti
tti
vatti
atti
tivatti
hattivatti
attivatti
ttivatti

4

EXTENSIONS OF SUFFIX TREE

Suffix trees can be extended:

- A **generalized suffix tree** is used to represent and compare many texts at a time (for example different DNA strands);
- **Suffix arrays** are a more concise representation of suffix trees, requiring less space.

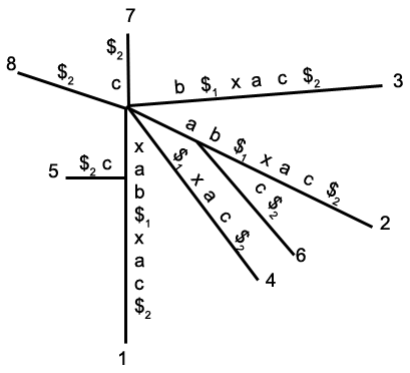
4.1 GENERALIZED SUFFIX TREE

A generalized suffix tree is used to represent all the substrings in a set of texts $T_1 ... T_k$. It's built in a **linear time with respect to the total length of the strings**:

$$O(\sum_i n_i) \text{ where } n_i = |T_i|$$

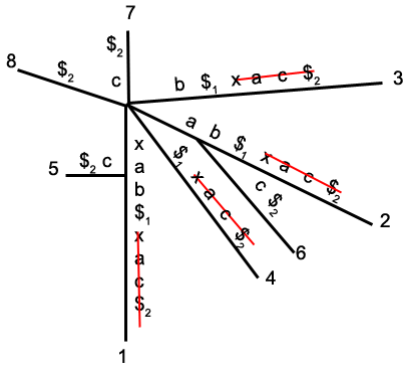
We have **two construction techniques**:

1. **Naive approach**. We simply **concatenate the texts** using different termination symbols ($\$_1, \$_2, \ldots, \$_k$), obtaining a new unique text. Then, we apply Ukkonen's algorithm to the text we've just produced, obtaining a suffix tree. For example, given the texts $T_1 = xab$ and $T_2 = xac$, we concatenate each string obtaining $xab\$_1 xac\$_2$. The suffix tree is:

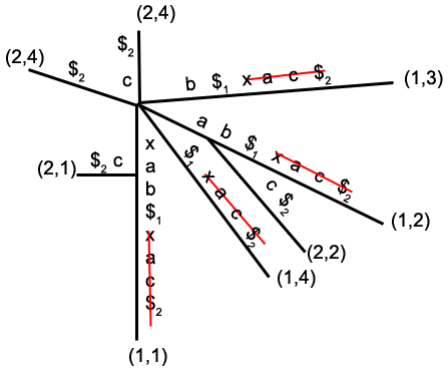


This approach is simple but **has some limitations**. First of all it **contains some junk** – suffixes of the unique strings containing

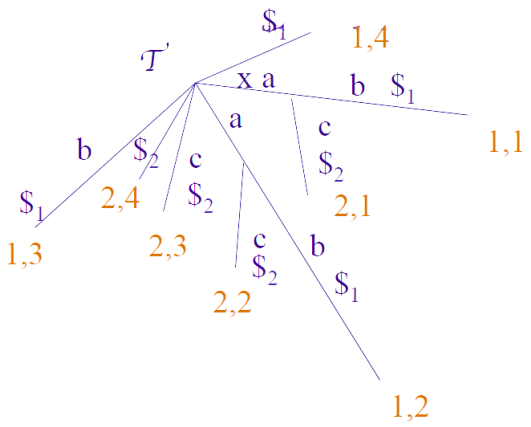
portions **not belonging to one single text** but inbetween two different strings. Therefore, we need to traverse the suffix tree to clean it, deleting every part after the first S_i. we find.



Furthermore, **leaves must be enumerated correctly**, not with a single value but with a couple where the **first value is the text number** (deducted from the terminator of each branch) and the second value is the **starting point of the text** (we consider each text singularly!). The result is:



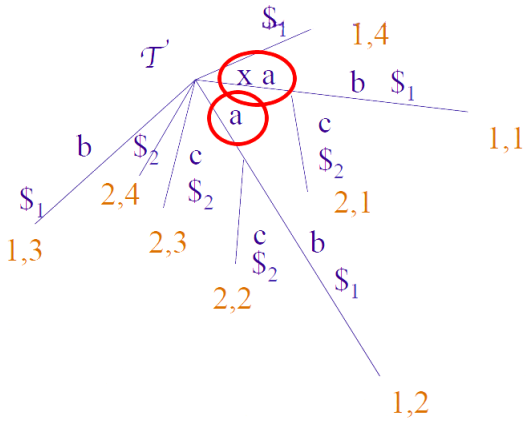
2. **Simpler approach**. The cleaner approach is still rather simple but produces a **suffix tree with no junk to remove**. It applies Ukkonen's algorithm to build the suffix trees one on top of the other: τ_3 for T_3 is built over τ_2 for T_2, which is built over τ_1 for T_1, etc. Again, each suffix tree is computed in time $O(n_i)$, where n_i is the number of symbols of the i-th text. The overall complexity is therefore $O(\sum_i n_i)$. To understand how this works let's consider again the previous texts $T_1 = xab$ and $T_2 = xac$ and apply this algorithms. The result is:



4.2 APPLICATIONS OF GENERALIZED SUFFIX TREES

Generalized suffix trees are used to work with sets of different sequences; therefore, they can be used to reply to more complex questions such as *are there common strings between different texts?* or *which is the longest common substring?* In fact:

- **Common strings**. It is enough to look at paths starting from the root and having different terminator symbols. For example these are common strings:



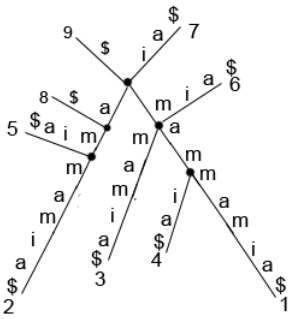
- **Longest common substring**. As before, it is enough to search the deepest internal node of a branch that ends with different terminator symbols. With respect to the previous example the longest common substring is xa.

- **Palindromic structures inside a single text**. To see whether there are palindromic structures in a single text we can build the generalized suffix tree of **my text** T **and its reversal** T^r. Then, we search for the common substrings. More in details, given T

and a pivot in position q, we consider the suffix $q+1$ in T and the suffix $(n-q)+1$ in T^r – like checking the substrings before and after the pivot. If there's a palyndrome substring in T then we should find the same string in T^r. For example, given the text $T = cattac$ with pivot in position q between the two ts, by building the generalized suffix tree of T and its reverse I can find a common string, cat, which respects the previous pivot rule. Therefore we have a palindromic substring. Of course the situation is more complex if the distance between the strings is larger (if there are symbols inbetween).

4.3 SAVING MEMORY WITH SUFFIX ARRAYS

Given a text T of size n, its suffix trees occupies a space as big as $O(n|\epsilon|)$, where ϵ is the alphabet of T[1]. If both n and ϵ are extremely large – for example if ϵ is the chinese alphabet with thousands of symbols or if we're performing a DNA mapping – the size of the suffix tree in the memory becomes **problematic**. There is a solution: suffix trees can be transformed in suffix arrays, very concise structures where the exact matching problem can be solved **as efficiently as with suffix trees**. Given a string T of length n, its suffix array Pos_T is an array of size n where cells are numbered $0, ..., n-1$ or $1, ..., n$; the content of each cell is the positions of the n suffixes of T in lexicographic order (note that \$ is always in the first position). For example, if $T = mammamia\$$, given its suffix tree...



...we create an array with 9 positions, numbered from 0 to 8, where the terminator symbol \$ in the first position. The 9 suffixes are inserted in **lexicographic order**: since we have $a\$$, $amia\$$, $ammamia\$$, $ia\$$, $mamia\$$, $mammamia\$$, $mia\$$ and $mmamia\$$ the array is:

1 Each node of the suffix tree can potentially be associated to as many symbols as there are in its alphabet.

0	9	$
1	8	a$
2	5	amia$
3	2	ammamia$
4	7	ia$
5	4	mamia$
6	1	mammamia$
7	6	mia$
8	3	mmamia$

4.3.1 *Cost of building suffix arrays*

Despite our previous example, remember that suffix arrays **contain only integers**: no information about the alphabet of the text is maintained. It follows that the **space requirement of suffix arrays are very modest**: if T has size n then the space needed for its suffix array is $O(n)$. In some situations, however, we could have enough space to build a suffix tree for T but not to save it permanently in the memory; to build its suffix array we have the following steps:

1. Build a temporary suffix tree of T;
2. Build the corresponding suffix array, traversing the tree in lexical order ($O(n)$). Note that in suffix trees the children of each node are **already stored in lexical order** so we need a simple depth-first traversing.
3. Get rid of the temporary suffix tree.

```
fun lex-df-search(node);
  begin
  if not leaf(node) then
     for_all m ∈ sons_of(node) do (the sons are scanned in lexical order)
        lex-df-search(m)
     else store(node) (store the index of node in Pos_T
end;
```

In the literature there are algorithms to build the suffix array directly from the text. They are, however, very intricate (their complexity is **logaritmic or even linear**).

4.4 APPLICATIONS AND COMPLEXITY OF SUFFIX ARRAY

Suffix arrays, exactly like suffix trees, can be used to solve *exact* matching problems – finding all the occurrences of a pattern P in a text T. The key is to **exploit the lexicographic order**: if P occurs in T then the **suffix array will store each occurrence consecutively**; therefore, to find all occurrences it is enough to find the lower and upper bounds with **two binary searches**: the remaining occurrences are all inbetween.
For example, suppose $P = mam$: with the first binary tree we find position 5, with the second position 6: since the indexes are consecutive, we only have occurrences in $T = mammamia$.

As for the **complexity of the binary search**, it depends on how many prefixes of P occur in T:

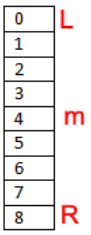
- **Worst case**, when P is very frequent in T: we have **almost** m **checks to be done** on the text and therefore the complexity is $O(mlogn)$.
- With random strings we have $O(m + logn)$.

To improve these complexities we can apply two tricks called **MLR** and **LCP** accelerators.

4.4.1 *MLR accelerator*

MLR accelerates the binary search by **avoiding useless comparisons**; in the worst case, complexity is still $O(mlogn)$; on average, however, we have $O(m + logn)$. MLR uses two indexes and two counters:

- Indexes L and R. They denote the left and right boundaries of the "current search interval" as the binary search proceeds. At each iteration of the binary search, L, R and $M = (L + R)/2$ are computed again (initially L equals to 1 adn R equals to n).



- Counters l and r. Given a pattern P, l works as a counter of **how many matches** we have in the lower positions of the suffix array (between L and M); r, instead, works as a counter of **how many matches** we have in the upper positions of the suffix array (between M and R).

For example, suppose $S = xabxac$ and $P = xa$. If, during one of the iterations, $L = 4$ and $R = 6$ then $l = 0$ and $r = 2$.

Pos_T	1	2	abxac
	2	5	ac
	3	3	bxac
	4	6	c
	5	1	xabxac
	6	4	xac

The value $mlr = min(l, r)$ can be used to accelerate the lexical comparison of P with the suffix array. In fact, if we know that P matches for l suffixes in the first part of the array and for r suffixes in the last part of the array, then $mlr = min(l, r)$ is the **minimum amount of matches in both portions of the suffix array**: therefore the binary search must focus not on the whole suffix array but only on the **portion from** $mlr + 1$ **on**. As another example, suppose $S = mammamia$ and $P = mam$, and that at a certain point we have $L = 2$ and $R = 5$; in this case $l = 0$ (no match on the lower end of the suffix array) and $r = 1$ (only one match of the pattern with the text) so there's not much advantage; on the other hand if we have $L = 5$ and $R = 7$ then $l = 1$ and $r = 2$ Their mlr value is 1 so the comparison must start from location 2.

MLR optimization has a **very good trade-off**: to maintain it we only need to keep track of l, r and $min(l, r)$, which causes a **negligible overhead** but optimizes greatly the binary search. As said before, the worst case still has complexity $O(mlogn)$, since the symbols in P from $mlr + 1$ to $max(m, l)$ are compared more than once; on average, however, by minimizing the number of useless comparison we can obtain a better cost, $O(m + logn)$ (any symbol in P is examined only once). Space-wise, MLR requires to keep track of two counters, which has a **constant cost**.

4.4.2 *LCP super accelerators*

MLR purports very good results but is not the *best* solution available. In fact, in the worst case complexity is still $(O(mlogn))$, meaning that symbols of the pattern P are examined more than once (**redundancy**). Redundancy is exactly what LCP super-accelerator aims to minimize: the goal is to reduce the number of redundant character examinations to at most one per iteration of the binary search.

> **Shortcomings of mlr**. Since $mlr = min(l, r)$, when $l \neq r$ all characters in P from $mlr + 1$ to $max(l, r)$ will have already been examined, so any comparison of those characters will be redundant.

s

What is needed is a way to begin comparisons at the maximum of l and r. Now, $lcp(i, j)$ is defined as **the longest common prefix between to** i, j **positions in the array**.

$lcp(i, j) =$ the longest common prefix between $Pos_T[i]$ and $Pos_T[j]$

For example, when $T = mississippi$, the suffix with position $i = 3$ is $issippi$, the suffix with position $j = 4$ is $ississippi$. To speed up the search, LCP uses two additional values ($lcp(L, M)$ and $lcp(M, R)$) for each triple (L, M, R) that arises during the execution of the binary search. Suppose we already have these values: how can we use them? We have four different cases:

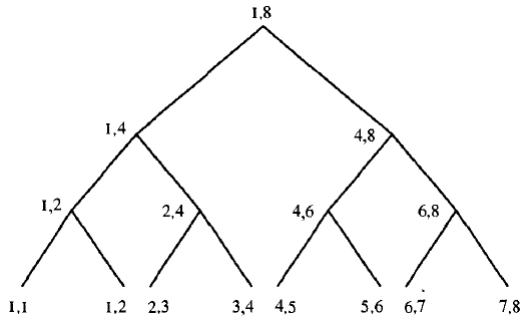
- **Trivial case**. If $l = r$ or, in other words, $min(l, r) = l = r$, then we must compare P with the suffix starting from position $mlr + 1$, exactly as before.
- **Interesting cases**. Suppose that $l \neq r$, with $l > r$. Depending on the value of $lcp(L, M)$ we have three situations:
  - $lcp(L, M) > l$: we search on the **right side of** M. In fact, this means that P agrees with suffix Pos(M) up to character l. L gets changed to M while l and r remain the same.
  - $lcp(L, M) < l$ we search on the **left side of** M. In fact, this means that P agrees with suffix Pos(M) up to $lcp(L, M)$. R gets changed to M, r is changed to $Lcp(L, M)$, l remains the same.

– $lcp(L, M) = l$ in this case, as before, P agrees with suffix $Pos(P)$ up to character l. The algorithm has to compare P to suffix $Pos(M)$ starting from character l+1, thus determining which of L or R change along with the corresponding change of l or r.

To sum up, by analyzing $lcp(L, M)$ we understand where to move our search in the suffix array minimizing the number of redundant comparisons. Complexity is therefore the sum of the cost of reading the pattern plus the cost of the search: $O(m + \log n)$;

4.4.3 *Computing the lcp(L,M) values*

To understand how the lcp values are computed we must introduce B, the complete binary tree with n leaves associated to the suffix array. Each node is labelled with the endpoints (L, R) of all the possible search intervals that could arise in the binary search of the suffix array.



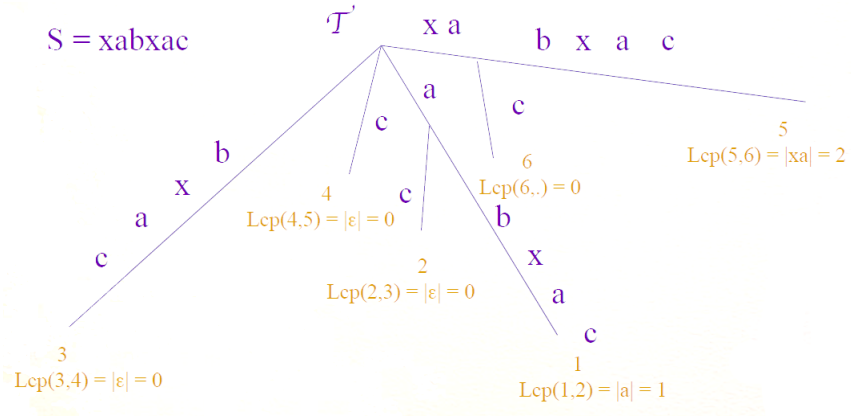
We associate:

- $lcp(i, i+1)$ to the leaves of B; these values are usually precomputed during the creation of the suffix tree, at no additional cost.
- $lcp(i, j)$ with $j > i+1$ to the internal nodes of B.

This way the number of lcp values we need is only $O(n)$. Now, to compute these values we have two approaches:

- $lcp(i, i+1)$, the lcp values of the **leaves of the binary tree** B. In this case i represents a branch with a certain suffix, $i+1$ the next branch in the lexicographic order. To get the value it's enough to **traverse the suffix tree depth first**. In fact, when we reach a branch and its next, if these branches have a common ancestor

the lcp value is the length of the label of the closest ancestor to the root; otherwise the lcp value is 0. For example, consider again the string xabxac and its suffix tree τ:



- $lcp(i,j)$, the lcp values of the **internal nodes of the binary tree** B. Luckily we can reuse what we've just computed:

$$lcp(i,j) = \min_{k=i}^{j-1} lcp(k,k+1)$$

In other words we get the minimum of the lcp values we've computed before, where i is k and $i+1$ is up to $j-1$. Note that this comes from the fact that $lcp(i,j)$ will always be **less or equal and greater or equal to any** $Lcp(k,k+1)$, for $i \leqslant k \leqslant j-1$: inner nodes are in fact common ancestors. Again, since the nodes traversed in the binary tree B are $2n-1$ the complexity of this exploration is $O(n)$.

To sum up, once we have the binary tree B representing the search intervals, by associating the $Lcp(i,i+1)$ to the leaves we can compute the $Lcp(i,j)$, for $j > i+1$, with a B-U traversal from the leaves and by computing the minimum value as seen before.

5

ALIGNMENT FOR DNA AND PROTEINS

So far we've considered *exact* string matching, a problem that is not so common in biology. **Approximate** string matching is instead far more realistic since it takes into considerations errors – i.e. mutations. It's employed to study sequences of nucleobases (DNA strand) or amino acids (protein strand).

5.1 WHAT IS ALIGNMENT?

The term *alignment* comes from biology: roughly speaking aligning two sequences means comparing their symbols to **find identities and similarities**. Consider for example the following sequence alignment between two human zinc finger proteins where the strands contain 60 and 40 amino acids, each with a specific property[1]:

```
AAB24882        TYHMCQFHCRYVNNHSGEKLYECNERSKAFSCPSHLQCHKRRQIGEKTHEHNQCGKAFPT 60
AAB24881        --------------------YECNQCGKAFAQHSSLKCHYRTHIGEKPYECNQCGKAFSK 40
                                    ****: .***:  * *:** * :****.:* *******..

AAB24882        PSHLQYHERTHTGEKPYECHQCGQAFKKCSLLQRHKRTHTGEKPYE-CNQCGKAFAQ- 116
AAB24881        HSHLQCHKRTHTGEKPYECNQCGKAFSQHGLLQRHKRTHTGEKPYMNVINMVKPLHNS 98
                 **** *:***********:***:**.: .***************    :  *.: :
```

The third line shows not both perfect matches (∗) but also **different degrees of similarities** (same group ":", same shape ".") between amino acids. The symbol "—" represents instead a **gap**: gaps allow sequences to be stretched or slided to find the best possible alignment. Alignment is a fundamental technique in bioinformatics and is often the **first step in many evolutionary and functional studies**: since it's very error-pronte, any error in alignment is **critical because it's amplified in later computational stages**. Alignment is all over biology: for example it's used in...

- **Function prediction**: finding the protein, the gene or the motif associated to a certain function;
- **Searching in a database**: databases are used to look for similar sequences and not exact sequence. BLAST and FASTA are the most important tools for these kinds of searches.

1 Red: small, hydrophobic, aromatic. Blue: acidic. Magenta: basic. Green: hydroxyl, amine, amide, basic. Gray: others.

- **Comparing a whole genome**, done for example to single out genes of a new organism or virus.
- **Genetic divergence and sequence assembly**, to find different species or individuals in the same population (due to migration or other phenomena).

Alignment is mostly impactful when dealing with evolutionary studies (similarity is mostly indicative of common ancestry) and function studies (sequences that are similar probably have similar functions). Note also that **life is monophyletic**, which means that even **very different biological entities share common ancestry**[2].

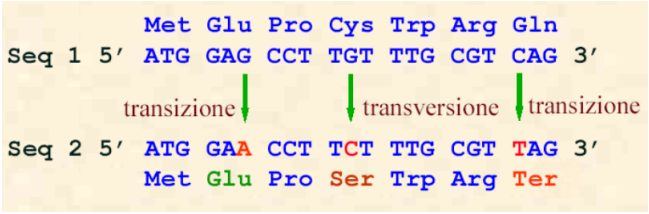
5.2 THE ERRORS ALLOWED BY ALIGNMENT

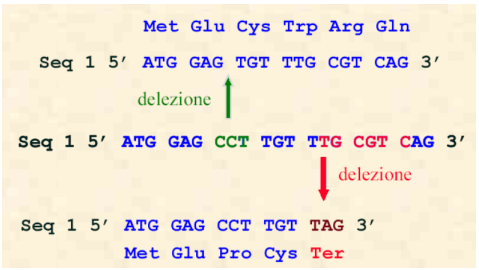
Alignment is interested in *approximate* matches between strings: this means that **errors** between strings are allowed. These errors are:

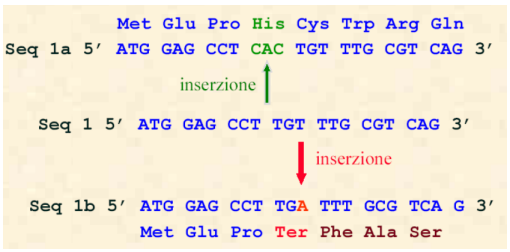
- **Mutations**. Caused by the duplication of DNA, mutations can be seen as **substitutions of symbols**. They can be **transitions** (a purine or pirimidine is transformed into another purine or pirimidine: A = G, C = T) or **transversions** (a purine is transformed into a pirimidine or viceversa: A = T, C = G). An example of mutation is:
- **Deletions**, where a symbol is wrongly deleted, and **insertions**, where a symbol is wrongly inserted.
- **Inversions**. Portions of symbols are switched, either between different strands or within the same strand:

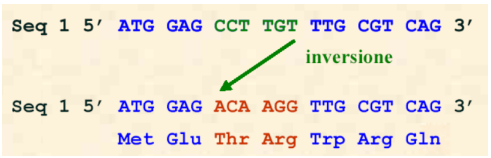
2 Homology is a field that studies similarities between different species resulting from common ancestry. More in details, we have four different relations between genetic information and function:

- A is **homologous** to B if they are **related by divergence from a common ancestor**;
- A is **paralogous** to B if they're governed by different genes but explain very similar functions in the same organism;
- A is **orthologuous** to B if they're governed by different genes but explain the same function in different organism. A wing in a butterfly is not related to a wing of a but, but their function is the same;
- A is **analogous** to B if they're governed by **different sequences with the same structure**: they might have a common ancestor.









Given two sequences we always assume that they're related by evolution, even if their functions are completely different (**homologous sequences**): a whole history of mutations and corrections has happened before our comparison. Note that we can't infer what happened to their common ancestor by simply looking at these strands: we assume that the **minimum number of transformations** has happened (**maximal parsimony**).

5.3 HOW TO STUDY AN ALIGNMENT?

In order to solve alignment problems, we must realize that there are **many different directions**. Which kind of alignment we want to perform? We can have:

- **Global or local alignment**: we can align two sequences as a whole or only some portions of them. Local alignment is used in database exploration while global alignment is used in comparative and evolutionary studies.

- **Multiple or pair alignment**: we can align two sequences or a whole group of them. Pairwise alignment problems have exact solutions while group alignment problems have approximate (heuristic) solutions.
- **Gaps**: are they allowed – to stretch our sequence and facilitate the matching – or not? Gaps can be internal (indicating a deletion or an insertion) or terminal, and may indicate missing data.

```
GCGG-CCATCAGGTAGTTGGTG--
GCGTTCCATC--CTGGTTGGTGTG
```

- **Score system and evaluation**: how are error scored? Which penalty can we associate to a mismatch between sequences? And how can we evaluate our results?
- **Algorithms**: depending on the problem at hand we must choose the best algorithm;

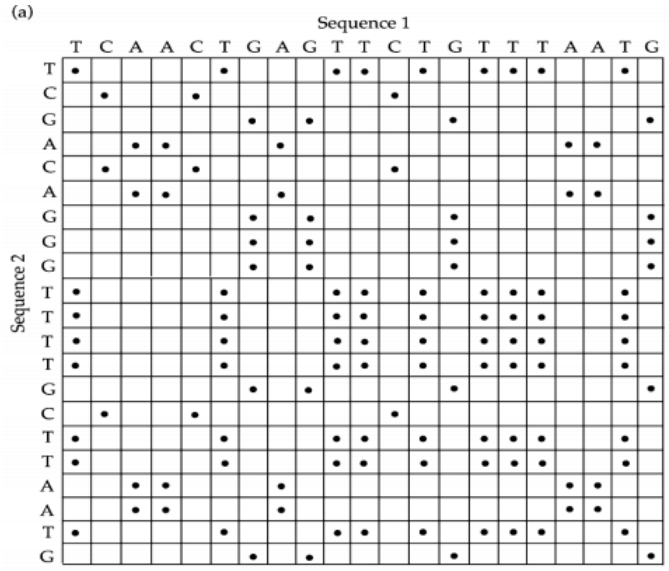
5.4 METHODS OF ALIGNMENT

Alignment can be solved using many different methods.

- **Manual alignment**. When gaps are few and the two sequences are not too different from each other a reasonable alignment can be obtained by **visual inspection**. This technique is still used in the final phases of many analyses.
- **Dot matrix**. The two sequences are written out as column and row headings of a 2−dimensional matrix called *dot matrix*. A dot is inserted if the symbols are identical.
- **Exact techniques** for pairwise alignment;
- **Heuristics**, with a certain degree of confidence, for group alignment.

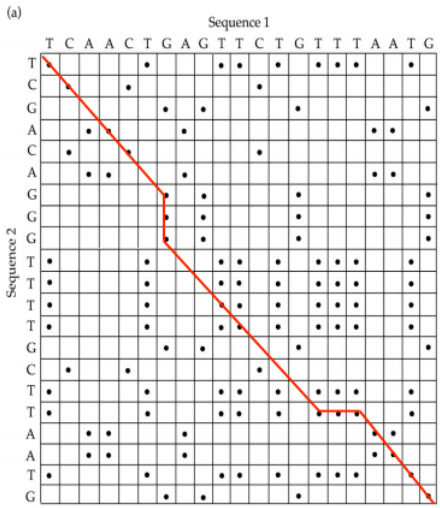
5.4.1 *Dot matrix*

Invented by Gibs and McIntyre (1970), the *dot matrix* approach uses a **binary matrix to visually represents matches/mismatches** between two strings. The sequences we want to match are used as **column and row headings of the matrix**: when there's an exact match between two symbols a dot is inserted in it.



Chains of dots and empty cells can be read on the dot matrix using the following rules:

- **Diagonal step**: if it's through a dot then we have a match, otherwise we have a mismatch;
- **Vertical step**: if it's through a dot then we have a match, otherwise we have a gap on the top sequence;
- **Horizontal step**: if it's through a dot then we have a match, otherwise we have a gap on the left sequence.

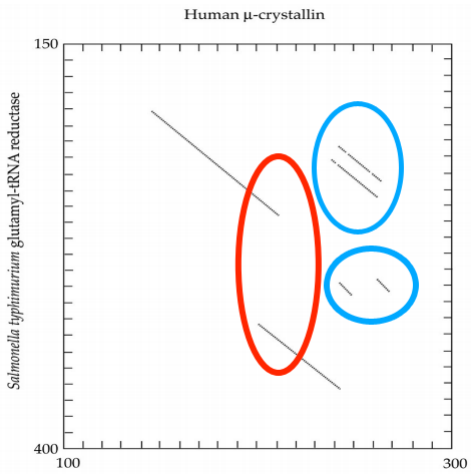


When very long sequences are analyzes the dot matrix **may become cluttered**: for example, when comparing DNA sequences, 25% of the elements will be occupied by dots by chance alone! We can improve

the readability of the matrix and reduce the noise by introducing a **sliding window with a threshold** j: we have a dot only if there are at least j matches within the selected sliding window.

```
for y = 0 to m-k do
   for x = 0 to n-k do
      compare x+1 .. x+k with y+1 .. y+k
      if at least h are identical then
         put a dot on [x+kdiv2, y+kdiv2]
```

An example of windows and thresholds is listed in fig. 44. As a reference, an example of dot matrix obtained using specific softwares is the following (human protein vs. salmonella):



As we can see we have two matching portions (diagonal lines); mismatches are abundant as well as gaps in the human protein (red circle: a coding region corresponding to 75 amino acids has either been deleted from the human gene or inserted into the bacterial gene). The parallel lines (blue circle) most probably indicate that two small internal **duplications** occurred in the bacterial gene.
A more realistic and rather unreadable dot matrix is instead the following (human zinc protein vs. itself): squares represent **palindromic structures** in the strand.

The dot matrix approach has **advantages and disadvantages**: on one hand it's **very simple** and has many applications (by changing the sliding window size and its threshold the data can be thorougly unraveled); on the other hand, however, building a dot matrix for very large sequences occupies a lot of memory; furthermore it values *only* matches and mismatches (no room for different degrees of similarity between strings).

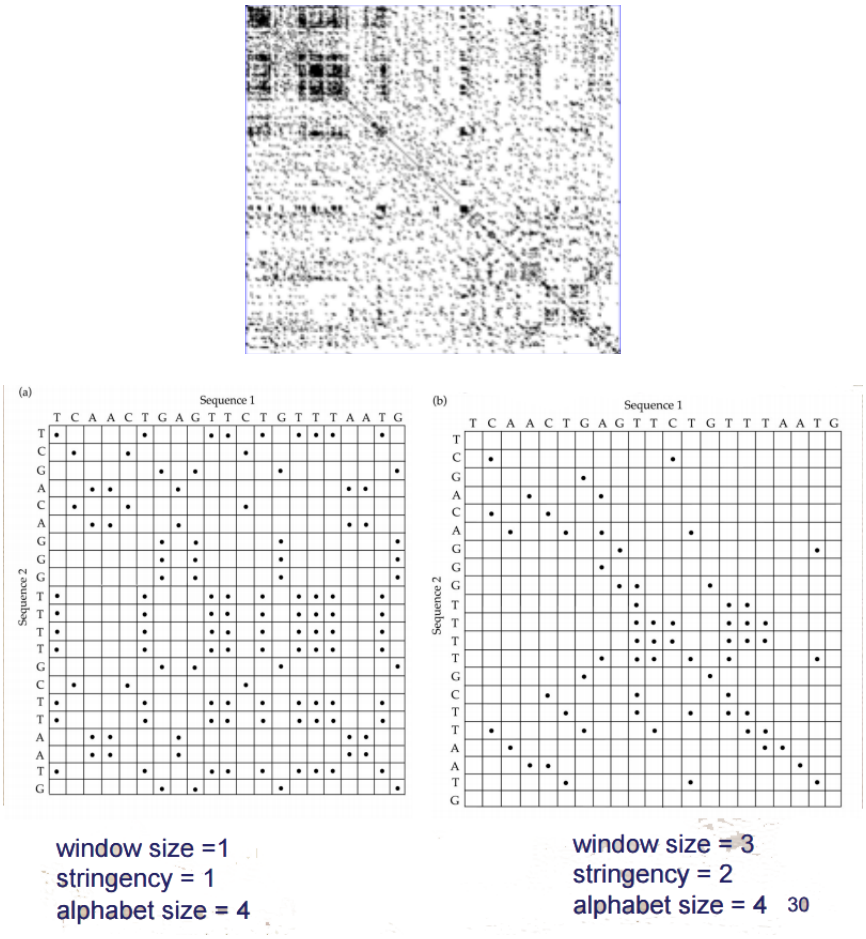


Figure 44: Different windows and thresholds.

5.5 OPTIMAL ALIGNMENT

As said before, we can't obtain their "true" alignment reflecting the evolutionary relationships between them. In practice we focus on the so-called *optimal* alignment, where the number of mismatches and gaps between them is **minimized**, depending on a certain scoring scheme.

5.5.1 *What is a scoring scheme?*

A scoring scheme is **a way of scoring matches, mismatches and gaps** between nucleotides or amino acids depending on the strand type. It usually comprises:

- A **gap penalty**, if gaps are allowed;

- A **scoring matrix** M, where $M(a, b)$ specifies the score associated to the match or mismatch between a and b.

Different scoring matrices depend on the strand type.

5.5.2 *DNA scoring matrices*

DNA scoring matrices are **very simple, with only** 4×4 **values**. The simplest approach gives a positive penalty to a match and a negative one for a mismatch, whatever it may be:

$$M(a, b) = \begin{cases} > 0 & \text{if } a = b \\ < 0 & \text{if } a \neq b \end{cases}$$

The simplest score matrix is binary (match 1, mismatch 0):

	A	T	C	G
A	1	0	0	0
T	0	1	0	0
C	0	0	1	0
G	0	0	0	1

A similar score matrix (match 5, mismatch -4) is used by BLAST:

	A	T	C	G
A	5	-4	-4	-4
T	-4	5	-4	-4
C	-4	-4	5	-4
G	-4	-4	-4	5

More complex matrices distinguish between transitions transitions (-1) and transversions (-5):

	A	T	C	G
A	1	-5	-5	-1
T	-5	1	-1	-5
C	-5	-1	1	-5
G	-1	-5	-5	1

As a general rule, **the more a certain type of mismatch is frequent the less it is penalized**. Depending on how we assign penalties the optimal alignment between two strings may vary.

5.5.3 *Amino acids scoring matrices*

Amino acids storing matrices are fairly more complex: the number of elements is **bigger** (20×20) and the various types of mismatches[3] are **penalized differently**. The value $M(i,j)$ of the score matrix represents the **probability of substitution** of one amino acid with the other; to understand what that means, suppose $x = x_1, ..., x_n$ and $y = y_1, ..., y_n$ are the pair of sequences we want to align. The score value between x_i and y_i is computed as the **log ratio** between two probabilities:

- **Random probability**. When **symbols are aligned randomly** (R) the probability of aligning a certain x_i of the first strand with a y_j of the second is:

$$P(x, y|R) = \prod_{i=1}^{n} q_{xi} \prod_{j=1}^{n} q_{yj}$$

  where q_{xi} and q_{yj} are the probabilities of symbols x_i and y_j.

- **Probability of an empirical pair occurrence**. When there's a model (M) correlating the two strands, the probability of alignment between x_i and y_j is:

$$P(x, y|M) = \prod_{i=1}^{n} q_{xiyi}$$

By computing the ratio of these probabilities we obtain $M(i,j)$, the penalty associated to (i,j):

$$Ratio = \frac{P(x, y|M)}{P(x, y|R)} = \frac{\prod_{i=1}^{n} q_{xiyi}}{\prod_{i=1}^{n} q_{xi} \prod_{j=1}^{n} q_{yj}} = \prod_{i=1}^{n} \left(\frac{q_{xiyi}}{q_{xi} q_{yj}} \right)$$

The overall penalty of the whole alignment of x and y is the sum of the **log of every penalty** between two symbols:

$$\sum_{i=1}^{n} \log \left(\frac{q_x iyi}{q_{xi} q_{yi}} \right)$$

3 Depending on chemical or physical properties. For example, a mismatched pair consisting of `Leu` & `Ile`, which are very similar biochemically to each other, may be given a lesser penalty than a mismatched pair consisting of `Arg` & `Glu`, which are very dissimilar from each other.

Very often these values are very small, hence they're **multiplied by 10 to get an integer**.

5.5.4 *Gap penalties*

To score gaps we must remember that insertions/deletions of whole sequences of symbols are more frequent than insertions/deletions of single symbols. This means that we should **distinguish between longer and shorter sequences of gaps**: the longer the gap sequence (usually in the beginning or end of a sequence) the lower the penalty. There are, of course, many ways to assign penalties to gaps:

- **Fixed cost**: every gap, no matter the amount of type, costs the same (0 or other numbers)
- **Constant cost**: every gap costs a constant penalty d, which is multiplied by the number of gaps.
- **Affine score**: opening/closing gaps cost o, extension gaps cost *e*.
- **Logarithmic cost**: the extension gap cost increases with the **logarithm of the number of gaps** (no direct proportionality between gaps and amounts).

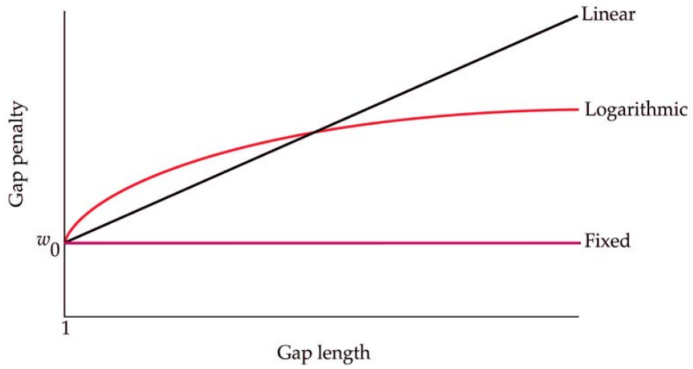
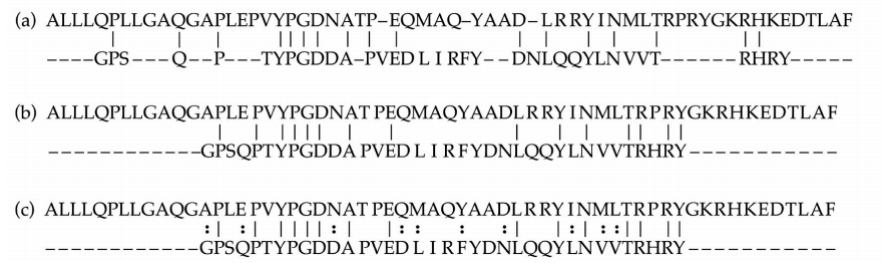


Figure 45: Different penalties for gaps.

The comparison of the same sequences purports different results depending on the selected penalty sistem: in case (a) we have no penalties on the gaps, which are introduced everywhere we want. In (b) gaps are penalized, however long sequences of gaps are less costly than single gaps; finally, in (c) an enhanced scoring matrix is used to show pairs of biochemically similar amino acids.



5.6 SCORE MATRICES FOR AMINO ACIDS: PAM

Acronym of *Percent/Point Accepted Mutation*[4], PAM is a **widely-used family of scoring matrices** used for for amino acids (PAM, PAM250, PAM60, ...).

5.6.1 *General implementation of PAM*

PAM, the first and oldest group of score matrices, was developed by Margaret Dayhoff (1978). PAM uses the evolutionary relations between very similar sequences of amino acids to produce a ***mutation probability matrix***: by comparing similar sequences (up to 85% of similarity) the score matrices are produced. The steps to infer these *mutation probability matrices* are the following:

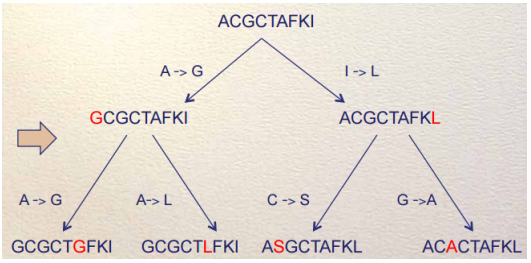
1. **Multiple alignment and phylogenetic tree**. Suppose we have the following group of sequences (same size and no gaps):

   ```
   ACGCTAFKI
   GCGCTAFKI
   ACGCTAFKL
   GCGCTGFKI
   GCGCTLFKI
   ASGCTAFKL
   ACACTAFKL
   ```

   We infer a **phylogenetic tree** – a binary tree showing **one chain of possible evolutionary relations** between these sequences (common ancestor is the root, remaining descendands are on the internal nodes and leaves).

2. **Frequency of substitution** F_{ij}. Next, we compute how many times an amino acid i is substituted with another j:

4 Point Accepted Mutation is the replacement of a single amino acid in the primary structure of a protein with another amino acid, done by evolution.



$$F_{ij} = \text{n. of times i is substituted with j and/or viceversa}$$

Note the **symmetry** (we don't know the evolutionary direction of this substitution). For example, the frequencies of substitution of the previous phylogenetic tree are:

$$F_{AG} = F_{GA} = 3$$
$$F_{AL} = 1$$
$$F_{CS} = 1$$
$$F_{IL} = 1$$

3. **Relative mutability** m_j. Next, for each amino acid j, we compute how much it is mutable:

$$m_j = \frac{\text{n. of mutations for j}}{\text{total number of mutations in the tree} * 2 * F_j * 100}$$

where F_j is the frequency of the symbol in the whole sequence:

$$F_j = \frac{\text{n. of j amino acids}}{\text{total number of amino acids}}$$

2 is used for simmetry reasons and 100 is a scale factor.

4. **Mutation probabilities** M_{ij} **and PAM matrix**. First for each pair of amino acids i, j we compute their **mutation probability** M_{ij}:

$$M_{ij} = \frac{m_j F_{ij}}{\sum_i F_{ij}}$$

Next, the i, j element of the matrix R_{ij} is:

$$\log(M_ij/F_i) = \frac{\text{observed ij mutation rate}}{\text{expected ij mutation rate}}$$

This value corresponds to the ratio we saw in the previous pages and can be **multiplied by a scaling factor**, 10, to avoid decimals; a positive score signifies a common replacement whereas a negative score signifies an unlikely replacement.
Note that the **diagonal elements** (where an amino acid is **substitued with itself**) R_{jj} are computed as follows:

$$R_{jj} = \log\left(\frac{m_j - 1}{F_i}\right)$$

The intuition is that if m_j measures the mutability of a symbol, then $(1 - m_j)$ measures its **non-mutability**.

5.6.2 PAM^n

PAM^n is extremely similar to PAM: it's enough to compute the power of n of the previous logarithm.

$$R_{ij} = \left(\log(\frac{M_{ij}}{F_i})\right)^n$$

The n means that the amount of evolutionary change we allow is n **mutations in 100 symbols**; PAM(1) allows only 1 mutation over 100 symbols.

5.6.3 PAM250

PAM250 works exacly as PAM, assuming that sequences of amino acids are separated by 250 PAMs – 250 mutations in 100 aminoacids (longer evolutionary time). PAM250 is used for comparing *distantly related sequences* while a lower PAM is suitable instead for comparing more closely related sequences. An example of PAM250 matrix is in fig. 46.

5.6.4 *How can we select a PAM matrix?*

It depends on the problem at hand: low PAMs are suitable for short sequences with strong local similarities while high PAMs are used for long sequences and weak similarities. As a rule of thumb we use:

- PAM60 for close relations (60% identity)

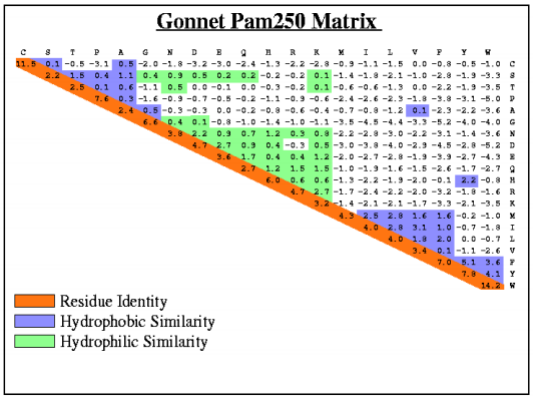


Figure 46: Example of PAM250.

- PAM120 recommended for general use (40% identity)
- PAM250 for distant relations (20% identity)

When in doubt, we try several different matrices (PAM40, 120 and 250 are generally recommended).

5.6.5 *Pros and cons*

The PAM approach makes several **unrealistic assumptions** which undermines its value. First of all, it assumes that evolution happens with a regular time and that different species have the same evolutive speed. It also works on phylogenetic trees which are inferred from the data and therefore only probable. Furthermore, it assumes that amino acid mutations are symmetric and that sequences have no gaps. Despite these cons, however, the method works and it's still widely used nowadays.

5.7 EMPIRICAL SUBSTITUTION MATRIX: BLOSUM

Acronym for *BLOcks/SUbstitution Matrix*, BLOSUM is again one of the most commonly used scoring matrices **for comparing amino acids**. It's more recent and modern than PAM.
BLOSUM matrices are **not based on evolution**: the values represented in the scoring matrix are extracted from a **database of fragments of proteins related by function**. BLOSUMn matrices are also numbered: for example BLOSUM62 means that he amino acids are **at most** 62% **identical**. An example of BLOSUM matrix is the following:

Blosum 45 Amino Acid Similarity Matrix

```
Gly  7
Pro -2  9
Asp -1 -1  7
Glu -2  0  2  6
Asn  0 -2  2  0  6
His -2 -2  0  0  1 10
Gln -2 -1  0  2  0  1  6
Lys -2 -1  0  1  0 -1  1  5
Arg -2 -2 -1  0  0  0  1  3  7
Ser  0 -1  0  0  1 -1  0 -1 -1  4
Thr -2 -1 -1 -1  0 -2 -1 -1 -1  2  5
Ala  0 -1 -2 -1 -1 -2 -1 -1 -2  1  0  5
Met -2 -2 -3 -2 -2  0  0 -1 -1 -2 -1 -1  6
Val -3 -3 -3 -3 -3 -3 -3 -2 -2 -1  0  0  1  5
Ile -4 -2 -4 -3 -2 -3 -2 -3 -3 -2 -1 -1  2  3  5
Leu -3 -3 -3 -2 -3 -2 -2 -3 -2 -3 -1 -1  2  1  2  5
Phe -3 -3 -4 -3 -2 -2 -4 -3 -2 -2 -1 -2  0  0  0  1  8
Tyr -3 -3 -2 -2 -2  2 -1 -1 -1 -2 -1 -2  0 -1  0  0  3  8
Trp -2 -3 -4 -3 -4 -3 -2 -2 -2 -4 -3 -2 -2 -3 -2 -2  1  3 15
Cys -3 -4 -3 -3 -2 -3 -3 -3 -3 -1 -1 -1 -2 -1 -3 -2 -2 -3 -5 12
   Gly Pro Asp Glu Asn His Gln Lys Arg Ser Thr Ala Met Val Ile Leu Phe Tyr Trp Cys
```

As we can see we have a list of amino acids: the higher the score between two amino acids the more closely related they are. But how can we select a BLOSUM matrix? Generally speaking, a higher n value is used for very similar sequences. For general use BLOSUM62 is recommended while BLOSUM80 is suitable for close relations, BLOSUM45 for distant relations.

5.8 COMPARISON OF PAM AND BLOSUM

Despite being different approaches, PAM and BLOSUM matrices are roughly equivalent:

```
PAM100 ==> Blosum90  (less divergent)
PAM120 ==> Blosum80
PAM160 ==> Blosum60
PAM200 ==> Blosum52
PAM250 ==> Blosum45  (More divergent)
```

BLOSUM matrices with higher numbers and PAM matrices with low numbers are both designed for comparisons of closely related sequences. BLOSUM matrices with low numbers and PAM matrices with high numbers are designed for comparisons of distantly related proteins.

6

OPTIMAL ALIGNMENT

We will now see the algorithms developed for *optimal alignment*. Optimality is a rather interesting concept: it conveys the idea that two strings can be perfectly aligned when, in fact, what we're looking for is a **probable alignment where errors are allowed up to a certain threshold**. Since errors are involved, optimality depends on a **score matrix** where penalties for insertions, deletions, substitutions and gaps are listed: without it it's impossible to construct any algorithm for optimal alignment.

There is a strong relationship between optimal alignment, a purely biological concept, and **approximate pattern matching**, which is instead studied by computer science.

6.1 COMPUTER SCIENCE: APPROXIMATE PATTERN MATCHING

The problem of approximate pattern matching is **very important in Computer Science**: one of the first problems of building a compiler is to find typos, strings similar but not quite identical to the exact term we want to use. One string approximates another if there's some sort of similarity between them; **similarity is computed as distance**, whose value depends on the errors we're considering and the cost associated to each of them. The errors we will focus in the next pages are **DNA mutations**:

- Substitution of a symbol with another one: $aaa \rightarrow aab$;
- Insertion of a symbol: $aaa \rightarrow aaba$;
- Deletion of a symbol: $aaa \rightarrow aa$;
- Transposition or switching of two adiacent symbols: $aaba \rightarrow aaab$.

Each of these errors has an **associated cost**, which can be constant (every error has the same cost) or depending on the operation or on the symbol involved. Scoring matrices can be very complex but in the following pages we will consider a simpler situation where the **penalty is always 1 for every operation and every symbol**.

6.2 COMPUTER SCIENCE: DISTANCE

The distance between two strings depends on the overall number of operations needed to transform one string into the other: if that value is below a certain threshold the strings are similar. Our goal is to **minimize the number of those operations**: the **alignment with the lowest cost is optimal**. For example, given these strings...

```
A = POLITE
B = PLATE
```

...we want to compute the minimal chain of tranformations from A to B. If there's no such chain the cost is ∞. Now, POLITE can be transformed in PLATE with two operations:

$$\text{POLITE} \rightarrow \text{del(2), subs(3,A)} \rightarrow \text{PLATE}$$

The minimal distance between A and B is thus $D(A, B) = 1 + 1 = 2$. In Computer Science we have **many different definitions of distance**:

6.2.1 *Hamming distance*

Interested only in **substitutions between strings of the same size**, it's symmetric, non-transitive and has the triangular property:

$$1.d(A, B) \geqslant 0$$
$$2.d(A, B) = 0 \mathtt{iff} |A| = |B|$$
$$3.d(A, B) = d(B, A)$$
$$4.d(A, B) + d(B, C) = d(A, C)$$

For example, if $X = aaaccd$ and $Y = abcccd$ then $d(X, Y) = 2$ since at least two substitutions are necessary.

6.2.2 *Levenshtein Distance or Edit distance*

A more flexible and used distance, it allows the "edit" operations (**insertion, deletion and substitution**) between strings of various sizes. For example, if $X = aaaccd$ and $Y = abccd$ then the distance is $d(X, Y) = 2$ since at least a substitution and a deletion are necessary.

6.2.3 *Episode distance*

It allows only insertions between strings with the **same or larger size**. It is not symmetric and there may be strings that are not comparable at all (in that case $d(A, B) = \infty$). For example if $X = aaccd$ and $Y = abbaccd$ then $d(X, Y) = 2$.

6.3 SIMILARITY BETWEEN TWO STRINGS

The notion of distance is related to the concept of similarity. Let d be a distance and e be an error threshold: then, two strings are similar if their distance is below the threshold.

If $d(A, B) \leqslant e$ then A and B are similar.

6.4 BIOLOGY: OPTIMAL ALIGNMENT

Biology is interested with alignment, which is closely related to approximate pattern matching. Let's consider again `POLITE` and `PLATE`; if we assume that we cannot slide one string outside the bounds of the other, there are 6 possible *global alignments* between the two strings:

```
POLITE  POLITE  POLITE
.PLATE  P.LATE  PLA.TE

POLITE  POLITE  POLITE
PL.ATE  PLAT.E  PLATE.
```

As we can see, gaps `-` `.` `-` are used to stretch the strings; each alignment also represents a certain sequence of transformations (insertions, deletions or substitutions). Now, what we want to obtain is the *optimal* alignment, the one where the **similarity score is the highest**: such score is computed additively, position by position, as we can see in the example below.

```
POLITE  POLITE  POLITE
.PLATE  P.LATE  PLA.TE
  2       3       3

POLITE  POLITE  POLITE
PL.ATE  PLAT.E  PLATE.
  3       4       5
```

Since we want to maximize the similarity, we need to minimize the distance between strings: therefore the optimal alignment is the first. Suppose now that we have the previous example but strings can be moved outside their bounds: the **number of possible associations becomes combinatorial** and, as such, very large. Therefore, trying every possible alignment by hand does not work: to solve the problem of optimal alignment we use **techniques based on dynamic programming** – where the concept of distance or similarity score are defined inductively; a matrix wiith partial computations is produced and the optimal result is inferred from it. The algorithms used are:

- Needleman-Wunsch (N-W) Algorithm for global alignment;
- Variants of Needleman-Wunsch (N-W) Algorithm for semiglobal alignments;
- Smith-Waterman (S-W) Algorithm for local alignment

6.5 NEEDLEMAN-WUNSCH (N-W) ALGORITHM

Based on the inductive definition of the *edit distance*, N-W is a **dynamic programming algorithm** that can be used to solve the *optimal and global alignment* problem. Given two strings, A of size n and B of size m, the algorithm inductively produces a matrix D where the element $D[i,j]$ is the **edit distance** between the prefix $A[1..i]$ and the prefix $B[1..j]$. Note that each edit distance is computed inductively.

6.5.1 *The N-W induction*

Suppose we want to define the edit distance $D[i,j]$ inductively. Assuming that the cost of every transformation is always 1, we have:

- Initialization(s): steps used to populate the first column and the first row of the matrix D.
  - if $i = 0$ then $D[0,j] = j$; in fact, if A is the empty string then we need at least j transformations to obtain B from A;
  - if $j = 0$ then $D[i,0] = 1$; as before, if B is the empty string we need at least i transformations to obtain A from B.
- Inductive assumption: $A[1,i-1]$ and $B[1,j-1]$ are already known.

- Inductive step: step used to populate the remaining rows and columns of the matrix D. When $i, j \neq 0$ we compute the minimal number of tranformations to obtain $A[1..i] = B[1..j]$:

$$D[i,j] = \min\left(D[i-1,j-1] + f(i,j), D[i-1,j] + 1, D[i,j-1] + 1\right)$$

Note that the three possibilities above have precise meanings:

  - $D[i-1,j-1] + f(i,j)$ means that that A can be transformed into B with $D[i-1,j-1]$ transformation plus $f(i,j) = 0$ if the symbols $A(i)$ and $B(j)$ are equal, $f(i,j) = 1$ if they're different. It represents a **substitution** (or the absence of operations). In the matrix D, it's the value on the top-left diagonal.
  - $D[i-1,j] + 1$ means that we need an insertion in the first sequence, A. it represents an **insertion** in the first string. In the matrix D, it's the value on the left.
  - $D[i,j-1] + 1$ means that we need a **deletion** in the first sequence, A (or an insertion in the second, B). In the matrix D it's the value on the top.

6.5.2 *Computing the* D *matrix*

Let's get back to the previous example where A = PLATE and B = POLITE. The N-W table we obtain is:

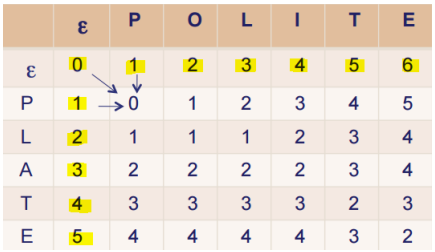
	ε	P	O	L	I	T	E
ε	0	1	2	3	4	5	6
P	1	0	1	2	3	4	5
L	2	1	1	1	2	3	4
A	3	2	2	2	2	3	4
T	4	3	3	3	3	2	3
E	5	4	4	4	4	3	2

It is populated as follows:

- **Initialization**. In this phase we focus on the first row and column. Let's see few examples:
  - (0,0): no transformation is needed to the distance is 0.
  - (0,1): to obtain P from the empty character we need an insertion so the distance is 1.

- (0,2): to obtain PO from the empty character we need two insertions so the distance is 2.
- ...

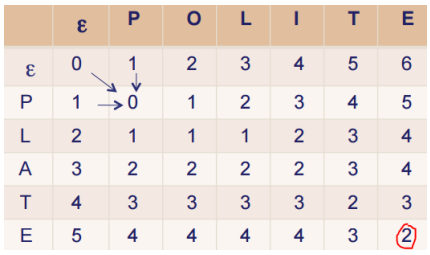
Eventually, we obtain the following result:



	ε	P	O	L	I	T	E
ε	0	1	2	3	4	5	6
P	1	0	1	2	3	4	5
L	2	1	1	1	2	3	4
A	3	2	2	2	2	3	4
T	4	3	3	3	3	2	3
E	5	4	4	4	4	3	2

- **Inductive step**. We can inductively compute the remaining values:
  - (1,1): $A(1) = B(1) = P$ so $f(i,j) = 0$. Therefore the minimum value between 1, 0 and 1 is 0. Incidentally, since A(1) and B(1) have the same symbol, no transformation is needed. To sum up, if the symbols are identical there's no additional cost to sum.
  - (1,2): $A(1) \neq B(2)$ so $f(i,j) = 1$. The minimum value between $0+1$, $1+1$ and $2+1$ is 1. To sum up, if the symbols are different there's an additional cost to consider, 1.
  - ...

  By following these rules we can fill the whole table. The **last value computed**, $D[n,m]$ stores the **minimal distance between A and B**.

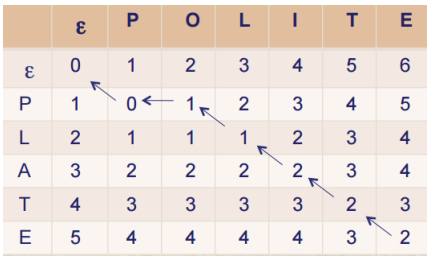


	ε	P	O	L	I	T	E
ε	0	1	2	3	4	5	6
P	1	0	1	2	3	4	5
L	2	1	1	1	2	3	4
A	3	2	2	2	2	3	4
T	4	3	3	3	3	2	3
E	5	4	4	4	4	3	2

6.5.3 *Reconstructing the transformations*

Suppose we're populating the matrix D: every time we choose one of the three values (diagonal, top, left) we could **create a link** to keep

track of the various optimal solutions. However it's possible to **reconstruct the edit path** – the chain of operations yielding the minimal cost $D[n, m]$ – even after the whole matrix has been populated by simply following the minimum value as follows:



	ε	P	O	L	I	T	E
ε	0	1	2	3	4	5	6
P	1	0	1	2	3	4	5
L	2	1	1	1	2	3	4
A	3	2	2	2	2	3	4
T	4	3	3	3	3	2	3
E	5	4	4	4	4	3	2

The reconstructed path is $2-2-2-1-1-0$; it corresponds to the following transformations:

```
POLITE
P LITE <- deletion
P LATE <- transformation
```

Note that we can move **on the diagonal only between identical values** ($1 \to 1 \to 1$) while we can move **on the left or on the top only from a bigger to a smaller value** ($2 \to 1 \to 0$). Of course there's no unique edit path; however, following these rules, we get the *minimal* one.

	ε	P	O	L	I	T	E
ε	0	1	2	3	4	5	6
P	1	0	1	2	3	4	5
L	2	1	1	1	2	3	4
A	3	2	2	2	2	3	4
T	4	3	3	3	3	2	3
E	5	4	4	4	4	3	2

6.5.4 *Complexity of N-W*

As for complexity, the memory space occupied by D is $S(nm)$. The time for building D is $O(nm)$, exactly as the time for optimal alignment.

6.6 SIMILARITY SCORE

Suppose now that the **operation costs are different** (and not constantly 1 as before) and that **gaps are also considered**. We can use the *similarity score* and define it inductively exactly as before. Suppose that $D[i,j]$ is the score of the optimal alignment between $A[1,i]$ and $B[1,j]$. Then:

- Initialization(s):
  - if $i = 0$ then $D[0,j] = j\gamma$, where γ is the gap penalty; in fact, if A is the empty string, it can be seen as a sequence of j gaps.

    ```
    POLITE
    ------
    ```

  - if $j = 0$ then $D[i,0] = 1\gamma$, for the same reasons as above.
- Inductive assumption: $A[1,i-1]$ and $B[1,j-1]$ are already known.
- Inductive step: when $i,j \neq 0$ we compute the maximal similarity between $A[1..i]$ and $B[1..j]$:

$$D[i,j] = max\left(D[i-1,j-1] + \sigma(i,j), D[i-1,j] + \gamma, D[i,j-1] + \gamma\right)$$

  ...where $\sigma(i,j)$ is the similarity score $M(i,j)$. As before, the three possibilities above have precise meanings:
  - $D[i-1,j-1] + \sigma(i,j)$ means that that A can be transformed into B with $D[i-1,j-1]$ transformations plus the similarity score of the positions we're considering, i and j.
  - $D[i-1,j] + 1$ means that we add a gap to the first sequence in order to align the first with the second sequence. It corresponds to an insertion in the first sequence.
  - $D[i,j-1] + 1$ means that we add a gap to the second sequence in order to align the second with the first. It corresponds to a deletion in the first sequence.

  Finally, note that our goal is to *maximize* the similarity, therefore we're looking to the maximum value between the three.

An algorithm exactly similar to the previous one can be inferred from this inductive definition. Note that so far we've treated sparse

and contiguous gaps with no differences; however, we could distinguish between a lower *opening penalty* and a bigger *extension penalty* if we assume that gaps are more frequent as opening segments.

6.7 SEMIGLOBAL ALIGNMENT

So far we've seen global alignments where whole strings are compared. We could be interested, however, in *partial alignment*, focusing only on portions of strings. In partial alignment **gaps are not penalized** but used to move the smaller sequence to find the best alignment. Suppose, for example, that we have the following strings:

```
ACTCG        or    AATCGTATA
ACAGTAGTCC         TATG
```

In both cases gaps are used to align the shorter strings with the longest ones so they must not be penalized.

```
-----ACTCG    or    AATCGTATA
ACAGTAGTCC          -----TATG
```

We could also have gaps in the **tail**: again, they're not penalized.

```
ACTGAAAT      or    ATAC----
CATGA---            ATCCGTAC
```

The inductive definition of the **edit distance** $D[i, j]$ **with no penalty for the introduction of gaps** in the initial position is not much different from the one we've seen before. In fact, the only change is in the **initialization**: the first row is completely 0. Everything else is exactly the same.

	ϵ	A	C	T	C	G
ϵ	0	0	0	0	0	0
A						
C						
T						
G						
A						
A						
A						
T						

Semiglobal alignment is very similar to approximate pattern matching. The algorithm to solve a semiglobal alignment problem receives in input a pattern P of size m, a text T of size n and an error threshold, ρ; it produces a matrix $D_{m \times n}$ containing the edit distance. The

procedure to obtain D is exactly as seen before (the only small difference is the first row, completely at 0). For example, suppose we have a pattern `P = TRENO`, a text `T = TRENTATRETREN` and a threshold $\rho = 2$ (it means that only when the cost of the chain of transformations is less than 2 we have an occurrence of the pattern in the text). Suppose also that we already have the following matrix D:

	ε	T	R	E	N	T	A	T	R	E	T	R	E	N
ε	0	0	0	0	0	0	0	0	0	0	0	0	0	0
T	1	0	1	1	1	0	1	0	1	1	0	1	1	1
R	2	1	0	1	2	1	1	1	0	1	1	0	1	2
E	3	2	1	0	1	2	2	2	1	0	1	1	0	1
N	4	3	2	1	0	1	2	3	2	1	1	2	1	0
O	5	4	3	**2**	**1**	**1**	**2**	3	3	**2**	**2**	**2**	**2**	**1**

To find the **occurrences of the pattern P in the text T** we need to check the last row and see whether there are positions whose values are less than the threshold. In this case we only have three occurrences:

	ε	T	R	E	N	T	A	T	R	E	T	R	E	N
ε	0	0	0	0	0	0	0	0	0	0	0	0	0	0
T	1	0	1	1	1	0	1	0	1	1	0	1	1	1
R	2	1	0	1	2	1	1	1	0	1	1	0	1	2
E	3	2	1	0	1	2	2	2	1	0	1	1	0	1
N	4	3	2	1	0	1	2	3	2	1	1	2	1	0
O	5	4	3	**2**	**1**	**1**	**2**	3	3	**2**	**2**	**2**	**2**	**1**

To find these occurrences we consider the **edit path** from each of these values in the last row **up to the first row**. The first is...

	ε	T	R	E	N	T	A	T	R	E	T	R	E	N
ε	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0
T	1	**0**	1	1	1	0	1	0	1	1	0	1	1	1
R	2	1	**0**	1	2	1	1	1	0	1	1	0	1	2
E	3	2	1	**0**	1	2	2	2	1	0	1	1	0	1
N	4	3	2	1	**0**	1	2	3	2	1	1	2	1	0
O	5	4	3	**2**	**1**	**1**	**2**	3	3	**2**	**2**	**2**	**2**	**1**

...and corresponds to:

```
TREN
TRENO <- INSERTION
```

Another occurrence is:
...which corresponds to:

	ε	T	R	E	N	T	A	T	R	E	T	R	E	N
ε	0	0	0	0	0	0	0	0	0	0	0	0	0	0
T	1	0	1	1	1	0	1	0	1	1	0	1	1	1
R	2	1	0	1	2	1	1	1	0	1	1	0	1	2
E	3	2	1	0	1	2	2	2	1	0	1	1	0	1
N	4	3	2	1	0	1	2	3	2	1	1	2	1	0
O	5	4	3	2	1	1	2	3	3	2	2	2	2	1

```
TREN
TRENO <- INSERTION
```

The last occurrence is:

	ε	T	R	E	N	T	A	T	R	E	T	R	E	N
ε	0	0	0	0	0	0	0	0	0	0	0	0	0	0
T	1	0	1	1	1	0	1	0	1	1	0	1	1	1
R	2	1	0	1	2	1	1	1	0	1	1	0	1	2
E	3	2	1	0	1	2	2	2	1	0	1	1	0	1
N	4	3	2	1	0	1	2	3	2	1	1	2	1	0
O	5	4	3	2	1	1	2	3	3	2	2	2	2	1

Again, it corresponds to:

```
TREN
TRENO <- INSERTION
```

Notice that the rule to traverse the matrix are exactly the same as before.

6.8 LOCAL ALIGNMENT

So far we've seen *global* and *semiglobal* alignments – alignment where whole sequences are concerned. Local alignment, instead, aims to identify the **similar portions of two given sequences**. This approach is usually used together with global alignmnent when not much is known about the sequences we're studying. An example of local alignment is:

```
      tccCAGTTATGTCAGgggac
         ||||||||||||
gcgttttcagCAGTTATGTCAGatc
```

From the computational point of view local alignment is not so different from global alignment: we're looking to *sub*sequences with strong similarities between two different strings.

6.9 SMITH-WATERMAN ALGORITHM FOR LOCAL ALIGNMENT

The S-W is another dynamic programming algorithm, used to solve local alignment problems. As such, it's based on an **inductive definition of local alignment**, a matrix of partial results (both positive and negative) and a reconstruction of the chain of operations performed on the strings. Note that we don't want to compare whole sequences but only portions. Substrings are found by cutting when the similarity reaches 0. Shifting in either direction is not penalized (idea of semiglobal alignment). This is reflected on the inductive definition below.

6.9.1 *Inductive definition of S-W*

- Initialization(s): steps used to populate the first column and the first row of the matrix D. If $i = 0$ then $D[0, j] = 0$; the same applies to $j = 0$: $D[i, 0] = 0$; in fact, **gaps are not penalized** in either case.
- Inductive assumption: $A[1, i-1]$ and $B[1, j-1]$ are already known (we know the number of transformation "so far").
- Inductive step: step used to populate the remaining rows and columns of the matrix D. When $i, j \neq 0$ we compute the maximal similarity between $A[1..i] = B[1..j]$:

$$D[i,j] = max\left(D[i-1,j-1] + \sigma(i,j), D[i-1,j] + \gamma, D[i,j-1] + \gamma, 0\right)$$

...where γ is the gap penalty while $\sigma(i,j)$ is the score matrix value $M(i,j)$. Note the **fourth option**, 0: when the similarity becomes negative it is cut out (similarity is 0).

  - $D[i-1,j-1] + f(i,j)$ means that that A can be transformed into B with $D[i-1,j-1]$ transformation plus $\sigma(i,j)$, wshich is the score $M(i,j)$. It represents a **substitution** (or the absence of operations).
  - $D[i-1,j] + \gamma$ means that we need an insertion in the first sequence, A, where we previously had a gap before.
  - $D[i,j-1] + \gamma$ means that we need a **deletion** in the first sequence, A (or an insertion in the second, B; note that we're still considering a gap). In the matrix D it's the value on the top.

The algorithm receives in input two sequences, A and B, to be aligned, plus a threshold ρ for the minimal score. The output is a matrix $D_{m \times n}$ containing the values of the similarity score. In practice, suppose the score matrix M has the following values:

- Match: +1;
- Mismatch: -1;
- Gap penalty: -1.

Initially we set the first row and column at 0. Next, we compute the remaining values using the previous inductive definition. Notice that all the values are non-negative: any similarity below 0 is in fact cut out, as explained before.

	ε	A	A	C	C	T	A	T	A	G	C
ε	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	1	0
C	0	0	0	1	1	0	0	0	0	0	1
G	0	0	0	0	0	0	0	0	0	1	0
A	0	1	1	0	0	0	1	0	1	0	0
T	0	0	0	0	0	1	0	2	1	0	0
A	0	1	1	0	0	0	2	1	3	2	1
T	0	0	0	0	0	1	1	3	2	2	1

We can find local alignments focusing our attention on the highest similarities on the whole matrix; the procedure is:

- Find all the highest values higher than the threshold);
- Traverse the table **up and left up to a 0**;
- Choose independent alignments.

With threshold 2 we obtain two interesting values (the 3s; the 2s are second choices):

	ε	A	A	C	C	T	A	T	A	G	C
ε	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	1	0
C	0	0	0	1	1	0	0	0	0	0	1
G	0	0	0	0	0	0	0	0	0	1	0
A	0	1	1	0	0	0	1	0	1	0	0
T	0	0	0	0	0	1	0	2	1	0	0
A	0	1	1	0	0	0	2	1	3	2	1
T	0	0	0	0	0	1	1	3	2	2	1

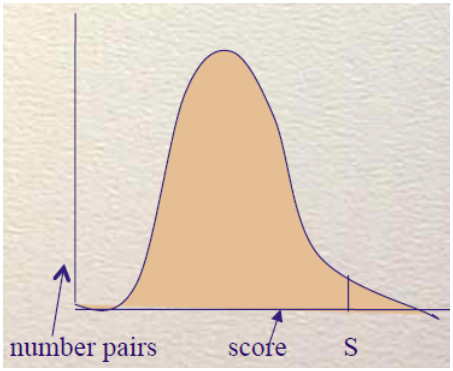
As for the **complexity** of this approach, it's quadratic in the size of the sequences we're comparing, exactly like in the case of *global* alignment.

6.10 RESULTS EVALUATION

Suppose we've just aligned two sequences: does this optimal alignment really reflect the similarities between A and B or is it optimal by chance? Could I find a better alignment between, say, A and another sequence? For example, suppose we consult a database and ask for a sequence similar to "`A= pincopallino`". The result is "`B =pappagallino`":

```
pincopallino
pappagallino
```

Many symbols of these sequences match but there's no complete coherence; is this an optimal similarity? Furthermore, this alignment has an associated score S: can we infer something from it to evaluate this alignment?
Theoretically, to evaluate the result, we should use a **control population** of sequences on which to reason; in practice, however, this **population is created artificially** by permuting the target symbols of A. Many new alignments are thus redone, each with an associated score. The distribution of these scores is usually the following:



To determine whether our previous alignment was optimal or not we study the **E-value**, the number of alignment with an identical or better score than S. In practive we evaluate the **region from** S **on**:

- If the **area is large** then our previous alignment was probably obtained by chance;
- if the **area is small** then the result is meaninful and optimal.

This technique is mainly used for ***local alignment***. For ***global alignment***, instead, we compute the **Z-score**: the greater, the more the global alignment is valid.

6.11 HEURISTIC TECHNIQUES

Heuristic methods are, as the name implies, *probable* methods: despite being rather good, the optimal alignment is not guaranteed to be found. However they're **very efficient and very fast** – faster than any exact method like N-W and S-W – for both global and local alignments: we can efficiently compare whole genomes (MUMmer) or search on databases with millions of sequences (FASTA, BLAST). The basic idea behind their use is that they comibine *fast filtering* of the uninteresting portions of the sequences and exact methods alignments.

6.11.1 *MUMmer*

MUMmer identifies Maximal Unique Matches (MUMs) in two given sequences using a generalized suffix tree. It's also capable of processing the gaps in the alignment and choosing the best, and to find the longest increasing sequences of MUMs.

6.11.2 *FASTA*

FASTA (1985) is a family of very accurate heuristic tools still used today mainly for **DNA comparison**. The idea behind FASTA is simple: even when an alignment between two sequences is not perfect **there must be small chunks of symbols that still match**. FASTA exploits this idea:

- It decomposes the target sequence into words of 4-6 (DNA) or 1-2 (proteins) symbols;
- It **indexes both the query and the target**: the query matrix contains the position in which the words are found; the target matrix contains instead the shifts with respect to the query.
- The best similarities are determined more or less like in the *dot matrix* approach, moving on the diagonals and extending the diagonals with gaps.

- Finally, it applies Smith-Waterman (exact pattern matching) to the diagonals, the most promising areas.

This approach is heuristic because it identifies **alignments using words, not single symbols**; usually, however, it returns good results. All in all, FASTA mixes up short exact matches, dot plot and exact matching techniques.

6.11.3 *An example of FASTA execution*

Suppose the query sequence is `falkgfijyapgcl` and the target sequence is `tgfikylpgact`. Words are sized 1, exactly like in the dot plot approach. Let's index the query string: we simply need to annotate where each symbol occurs on the whole list of 20 possible amino acids.

```
A  C  D  E  F  G  H  I  J  K  L  M  N  P  Q  R  S  T  V  W  Y
2  13          1  5     7  8  4  3        11                 9
10             6  12             14
```

Next, to **index the target sequence** by compute the relative distance between `tgjylpgcat` and `falkgfijyapgcl`:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14
f a l k g f i j y a  p  g  c  l

1   2 3 4 5  6  7  8  9  10  11  12
T   G F I J  Y  L  P  G   C   A   T
3  -1 3 3 3  3  -3 3  3  -4  -8   2
10  3                    3   0
```

We can see a diagonal (3-3-3-3-3-3) and another one (3-3) which corresponds to two exact matches (`GFIJY` and `PGC`).

6.11.4 *BLAST (Basic Local Alignment Search Tool)*

Developed in 1990 after FASTA, BLAST is another tool used to study **protein alignments with or without gaps**, with lots of parameters set by the users[1]. It's generally regarded as an improvement of FASTA. The idea behind it still uses words, which are obtained by **sliding a**

1 Blast is strongly statistically-controlled and through lots of parameters we can set in order to find our alignment: lots of responsibilities for the users.

window of predefined size ($W = 3$ or 4) on the query sequence. The steps to execute BLAST are the following:

- Split the query into words of length W by a sliding window of size W on the query sequence;
- Filter out common words (very common amino acids, for example): only very specific words are kept).
- For each of the resulting words, generate a **list of similar words** with a certain similarity score $\geqslant T$ (global alignment with N-W is applied); note that a perfect match is not requested!
- Next, we proceed to find good local alignments between each word and its list of similar words, usually using a database. The matching string is called seed. In the first implementation of BLAST it was enough to find seeds with one match (one hit method); the latest versions implement instead a two hits method (seeds with matches in two points).
- Once we have the query words and the corresponding seeds, seeds are **extended on both sides without gaps**, up to a score threshold S. If the score is negative, X specifies how much to insist on extending the alignment.

6.11.5 *An example of BLAST execution*

Suppose we have the following query:

```
AILVPTV
```

The sliding window has size 4 so the words are:

```
AILV
ILVP
LVPT
VPTV
```

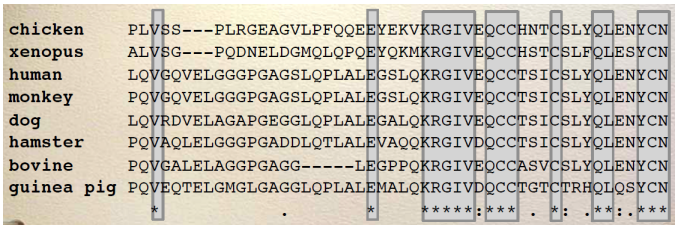
Suppose the **last two proteins contain common amino acids**: they are discarded. now we must generate the list of similar words (where the similarity score is at least T). Suppose C and A are *very* similar amino acids: then, a similar word for AILV is CILV. Once two lists of similar words are generated, we search for matches between the original words and the generated words; depending on the version

of BLAST we need at least 1 or 2 "hits" (number of matches within two sequences). Once these seeds are found, they're extended incrementally without gaps, up to a certain threshold score S.

7

MULTIPLE ALIGNMENT (MSA)

We will now focus on the problem of **aligning many sequences at the same time**. But why should we compare more than two sequences? Multiple alignment allows to compare the **genome of groups of different organism** and to **reveal the subtle similarities that pairwise alignment cannot show**. The example below shows that chickens, guinea pigs, hamsters, dogs and even humans share identical portions of amino acid sequences despite their different appearances:



How can we **score multiple alignment**? Exactly as with pairwise alignment, we assume:

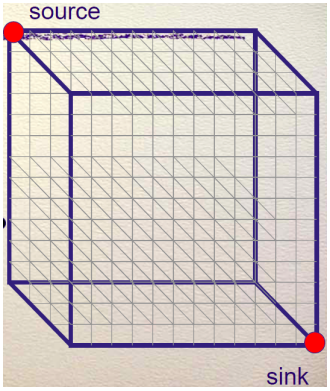
- **Additivity**: in each sequence the context of each symbol is not relevant so we simply perform the sum of each penalty or score;
- **Independence**: each position is independent from the other, a convenient assumption that simplifies the analysis. The only real change is that we consider **whole columns at a time**, as in this example:

```
A T _ G C G _
A _ C G T _ A
A T C A C _ A
```

7.1 EXACT MATCHING TECHNIQUES

To solve exact global matching problems, with 2 sequences we could use the *inductive definition of the edit distance*: the optimal alignment was yielded by a traversal of the edit table from the bottom right value up to the top. With three dimensions the strategy is the same but, instead of a table, we have a **cube** where each axis represents a

sequence to align. The traversal is compute from source to sink (top to bottom) as in this example:



As for the scoring matrix, we need a 3-dimensional scoring matric with entries $\delta(x, y, z)$ – clearly anot trivial to compute. The inductive case, in fact, is rather complex (we must consider not only the diagonal, up and left values of the edit table but four more values):

$$s_{i,j,k} = \max \begin{cases} s_{i-1,j-1,k-1} + \delta(v_i, w_j, u_k) & \text{cube diagonal: no indels} \\ s_{i-1,j-1,k} + \delta(v_i, w_j, \_) & \\ s_{i-1,j,k-1} + \delta(v_i, \_, u_k) & \text{face diagonal: one indel} \\ s_{i,j-1,k-1} + \delta(\_, w_j, u_k) & \\ s_{i-1,j,k} + \delta(v_i, \_, \_) & \\ s_{i,j-1,k} + \delta(\_, w_j, \_) & \text{edge diagonal: two indels} \\ s_{i,j,k-1} + \delta(\_, \_, u_k) & \end{cases}$$

Dynamic programming algorithms such as the ones ween before can be extended to groups of sequences easily, but they're impractical due to thei exponential complexity: $(O(2^k n^k)$ for k sequences! Exact approaches are useful only when few sequences are to be aligned accurately, otherwise we **prefer heuristic methods**.

7.1.1 *From multiple to pairwise and viceversa*

Remember that every **multiple alignment induces several pairwise alignments**...

```
x: AC-GCGG-C          x: ACGCGG-C; x: AC-GCGG-C; y: AC-GCGAG
y: AC-GC-GAG   ->     y: ACGC-GAC; z: GCCGC-GAG; z: GCCGCGAG
z: GCCGC-GAG
```

...but the **inverse is not true**: we can't always construct a multiple alignment starting from several pairwise alignments (for example if

the single sequences have gaps, multiple alignment is not allowed). This means that **pairwise alignments may be inconsistent**.
Furthermore, when we have an optimal multiple alignment, we can infer pairwise alignments between all pairs of sequences, but are **not necessarily optimal**.

7.2 MULTIPLE ALIGNMENT SCORES

For multiple alignment, we can use **three different score systems**:

7.2.1 *Multiple LCP (Longest Common Subsequence) score*

This is a very simple approach – so simple that it's used only for sets of very similar sequences. We have a **match if every symbol of the same column is the same**; for example, int this case the score is 1.

```
AAA
AAA
AAT
ATC
```

7.2.2 *Entropy score*

. For each column, we compute the frequency of each symbol. For example:

- First column: $pA = 1$, $pT = pG = pC = 0$;
- Second column: $pA = 0.75, pT = 0.25, pG = pC = 0$
- Third column: $pA = 0.50, pT = 0.25, pC = 0.25 pG = 0$

Next, we compute the **entropy of each column** : in the best case it's 0, otherwise it's 2.

$$\sum_{x=A,T,C,G} p_x \log p_x$$

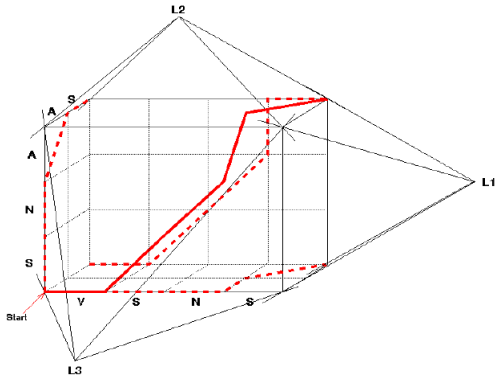
Once we have an entropy measure for each column, we sum them to obtain the overall score of the multiple alignment:

$$\sum_{i \in \text{columns}} (- \sum_{x=A,C,T,G} p_x \log p_x)$$

To have the best alignment the overall entropy must be **minimized**.

7.2.3 *SP-Score (Sum of Pairs)*

The SP-score is based on the pairwise alignments inferred from a multiple alignment. Visually, it is like projecting a 3-D multiple alignment path on to a 2-D face of the cube.



The SP-score is computed as follows: given a multiple alignment with k sequences, we sum the score of every possible pairwise alignment of these sequences.

$$s(a_1, \ldots, a_k) = \sum_{i,j} s * (a_i, a_j)$$

Since each position in each sequence is counted more than once, it corresponds to considering more possible ancestors in evolution (mutations are counted more than once).

The **scoring of gaps** in multiple alignment usually privileges the alignment of gaps (not distributed all over but conserved in the position they were found). We will see that when they're set they're also maintained where they are.

7.3 PROFILE REPRESENTATION OF MULTIPLE ALIGNMENT

Profile representation is the simplest technique to work with multiple alignment. It's based on a **concise numerical representation** showing only relevant information on the alignment. There are databases and tools to deal with this kind of representation (particolar versions of BLAST like PSI-BLAST). Let's see how it works. Suppose we have the following five sequences in a multiple alignment:

```
- A G G C T A T C A C C T G
T A G - C T A C C A - - - G
```

```
C A G - C T A C C A - - - G
C A G - C T A T C A C - G G
C A G - C T A T C G C - G G
```

To obtain its profile representation we consider the frequency of each symbol with respect to the column where it is placed:

```
A    1           1        .8
C .6        1       .4  1     .6 .2
G     1 .2                 .2        .4  1
T .2           1    .6              .2
- .2    .8                    .4 .8 .4
```

As we can see, in the first column we have 3 Cs (60%), one T (20%) and one gap (20%). We **lose the position of each symbol in each column** but we "capture" the global information. Note taht this is a **sparse representation**, requiring less memory. To further reduce memory space a threshold can be introduced (losing further information). Using this representation we can:

- **Align a sequence** A **to a profile** W, to see whether that sequence belongs to the profile group or not.
- **Align a profile** A **against another profile** B.

7.4 THE GREEDY APPROACH (HEURISTIC TECHNIQUE)

The first method for multiple alignment is called *greedy approach*. Given k sequences, we use as the starting point of the algorithm the **best pairwise alignment** – with the highest similarity score – which becomes a profile. At each step, the algorithm looks for the best alignment between the previous profile and the remaining sequences, finally converging to a multiple alignment. For example, consider the following four sequences:

```
s1 GATTCA
s2 GTCTGA
s3 GATATT
s4 GTCAGC
```

The score matrix is 1 for a match, 0 for a mismatch, -1 for a gap. The pairwise alignments are $\binom{4}{2} = 6$:

```
s2 GTCTGA                 s1 GAT-TCA
s4 GTCAGC (score = 2)     s2 G-TCTGA (score = 1)

s1 GAT-TCA                s1 GATTCA--
s3 GATAT-T (score = 1)    s4 G--T-CAGC(score = 0)

s2 G-TCTGA                s3 GAT-ATT
s3 GATAT-T (score = -1)   s4 G-TCAGC (score = -1)
```

The first alignment is the best so it becomes the first profile to use. In the next step we find the best alignment between this profile and the remaining sequences, eventually converging to a multiple alignment. The advantage is that the method is **very simple** and its cost depends on the number of pairs we have to consider. It is however an **heuristic technique**, with a certain degree of uncertainty.

7.5 PROGRESSIVE ALIGNMENT

Progressive alignment is a widely-used and flexible suite of tools (Clustal, T-coffee). It can be seen as a variation of the previous greedy algorithm with a somewhat **more intelligent strategy for choosing the order of pairwise alignments**.

7.5.1 *ClustalW*

ClustalW is a popular multiple alignment tool. "W" stands for "weighted": at each step the **user can control the advancement of the multple alignment**. ClustalW works through **three steps**:

1. As before, we compute the similarity of every possible pairwise alignment;
2. A **guide three** is built;
3. Progressive alignment (a new string is aligned to the previous profile) is performed. It is guided by the guide tree.

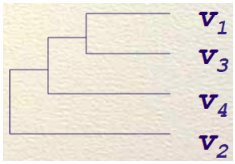
Different scoring matrices are allowed, one for each step of the algorithm: for example, we can assume with more related sequences and we end with less related sequences. **Gaps can also be used freely**, determining how symbols are aligned at each step.

7.5.2 *Example of ClustalW*

Suppose we have four different sequences we need to align, $v_1, \ldots, v_4$. In the first step we perform the six pairwise alignment, giving the following similarity matrix (computed as exact matches / sequence length):

	v_1	v_2	v_3	v_4
v_1	-			
v_2	.17	-		
v_3	.87	.28	-	
v_4	.59	.33	.62	-

The most similar sequences are $v_1 - v_3$ (similarity is 87%) so they're used as the starting profile. We create a guide tree using the similarity matrix, roughly reflecting evolutionary relations:



Next, we perform the progressive alignment: we start by aligning the two most similar sequences and, following the guide tree, we progressively align the remaining sequences to the previous profile. **Gaps can be inserted** if necessary.

```
v_1,3 = alignment (v_1, v_3)
v_1,3,4 = alignment((v_1,3),v_4)
v_1,2,3,4 = alignment((v_1,3,4),v_2)
```

Note that at each step of the way the user can set several parameters (in the latest versions of Clustal we can also alter the structure of the guide tree). All in all, progressive techniques are the rather simple, fast, not expensive and widely used; however they're still heuristics (**optimal alignment is not guaranteed**); they're also very **sensitive to the initial choice of pairwise alignment**: if the initial pair is not optimal, large errors may be produced in the final stages of the algorithm.

7.6 (DETERMINISTIC) ITERATIVE ALIGNMENTS

Iterative alignment solves the sensitivity problem of progressive alignment: first an initial multiple alignment is produced using a progres-

sive method; then, one or more sequences are extracted and realigned iteratively to **refine the previous result**. The score is thus recomputed until it **maximizes a certain objective function**, or until a bound on the number of iterations is reached. Note that this procedure is guaranteed to converge to a *local* maximum; in fact, if the initial alignment is poor, it may not converge to a global maximum.

7.6.1 *(Stochastic) Iterative alignments*

Other iterative alignments can also be stochastic: sequences are extracted randomly and a **control function** (based on mutations or other factors) decides the survivability of each new alignment.

7.7 CARILLO-LIPMAN METHOD

A heuristic method, the Carillo-Lipman approach is very similar to what we've seen in the *exact* multiple alignment technique. The idea was to to **compute a smaller area of the cube**, there there is the highest probability to find the optimal values; the area is defined using pairwise alignments. In the same way, the C-L approach provides **a bound to the hypercube**, limiting the search space where the multiple alignment is found.

The steps are:

1. Consider the pairwise alignments of each pair of sequences and create a phylogenetic tree from these scores.
2. Produce a draft multiple alignment, built from the phylogenetic tree.
3. The pairwise alignments and draft MSA provide a reduced solution space on which dynamic programming is used to find the solution.

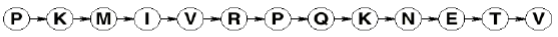
This method does not guarantee an optimal alignment for all the sequences in the group because **so much of the solution space is excluded from the search**. Still, it's very useful where at most 8 sequences of average size and similarity need to be aligned.

7.8 PARTIAL ORDER MULTIPLE ALIGNMENT (POA)

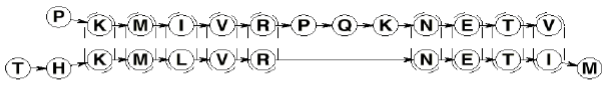
POA is used mainly for **aligning fragments of proteins**; in proteins, in fact, **mutations may interest whole portions of amino acids**, which are moved somewhere else on the strand. The method is based on the **graph representation of pairwise alignments**: given a conventional pairwise alignment between two sequences...

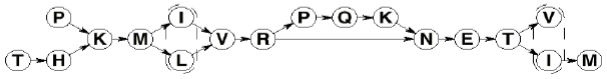
A
```
. . P K M I V R P Q K N E T V .
T H . K M L V R . . . N E T I M
```

...each sequence is represented as a graph...

B 

...and by combining the graph representation of each sequence we obtain a graph representation of the pairwise alignment, where vertices with identical labels are fused and dashed edges connect equivalent positions

C 

D 

The goal, now, is to find the g**raph rapresentation of the domain structure** – a graph G such that every sequence in the multiple alignment is a path in G (every sequence is mapped to the graph). To align sequences onto a directed acyclic graph we have the following steps:

1. Build a guide tree by considering the similarity score of all possible pairwise alignments;
2. Perform progressive alignment following the guide tree

8

MODELLING SEQUENCES USING GRAMMARS

Until now we've treated biological sequences as strings of independent, uncorrelated symbols – an assumption that is computationally convenient but not realistic. In this chapter we will consider **how groups of strings can be modelled using grammars**, applying *formal language theory* to molecular biology. In the following pages we will focus on grammars with the following features:

- **Context-free (type 2) and regular (type 3) grammars only**: they're the only grammars whose algorithms have **acceptable complexities**; furthermore, sequences produced by these grammars can be recognized linearly, scanning the text from left to right.
- **Potentially ambiguous**: the same sequence can have different parsings.

8.1 COCKE-YOUNGER-KASAMI (C-Y-K) ALGORITHM

CYK is a parsing algorithm that given a grammar G and a sequence w, **finds the optimal parsing tree for** w **in** G if $w \in L(G)$. Notice that:

- If the grammar is ambiguous CYK finds one of the many possible optimal trees of the sequence; potentially, the algorithm can find them all.
- The grammar used by CYM must be in **Chomsky normal form with no empty productions**.

The structure of the algorithm is based on the usual **dynamic programming approach**: first we inductively build a **parse table** T containing all the parse trees for all possible substrings of w; then we infer the optimal parse tree from the table.

8.1.1 *Building the parse table*

The parse table T is a **triangular table** explored using two indexes, i (the starting position of the substring of w) and j (the length of the

substring). Each cell $t_{i,j}$ contains a set of one or more[1] nonterminals whose production rules generate the substring from $a_i ... a_j$. Induction is performed on the length j:

- **Initialization**: if $j = 1$ then $t_{i,1} = \{A | A \to a_i \in P\}$, meaning that $t_{i,1}$ contains the nonterminal whose production rule generates the single symbol a_i.
- **Inductive case**: if $j > 1$ then $t_{i,j} = \{A | \exists k \in [1, j]$ such that $A \to BC \in P$ and $B \in t_{i,k}, C \in t_{i+k,j-k}\}$. This means that CYK considers **every possible partition** of the substring $a_i ... a_j$ and looks for the nonterminal$P$ whose production rule $P \to BC$ produces the first half with B and the second half with C.

For example suppose we have a CNF grammar with productions:

$S \to AA|AS|b$
$A \to SA|AS|a$

Is the sequence `w = abaab` in $L(G)$? To answer we must apply the CYK algorithm. First we **initialize the parse table** populating its first row:

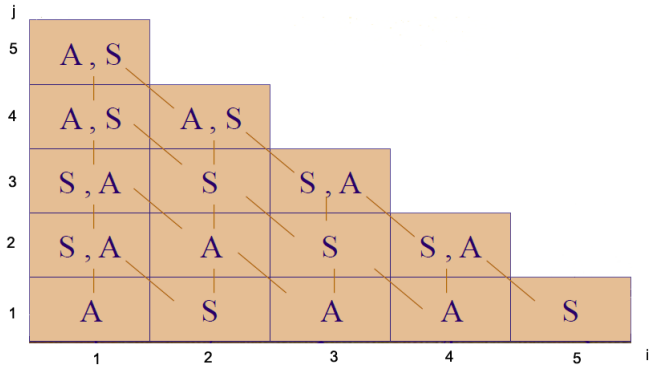
- $t_{1,1}$ for "a". The corresponding production is A since $A \to a$. The same applies for $t_{3,1}$ and $t_{4,1}$.
- $t_{2,1}$ for "b". The corresponding production is S since $S \to b$. The same applies for $t_{5,1}$.

Next we populate inductively the rest of the parse table. The idea is simple: get the bottom and bottom-right symbols of the parse table and check which rules produce them. As an example.

- $t_{1,2}$ for "ab". The nonterminals we need to use are $\{A\}, \{S\} = \{AS\}$. $\{AS\}$ is produced by both S and A so this cell contains $\{S, A\}$;
- ...

The final result is (notice the lines to show how induction is performed):

1 Notice the ambiguity: the same substring can be derived using different rules.



8.1.2 *Understanding if $w \in L(G)$*

Once we have the parse table, to understand if the sequence is recognized by the grammar and belongs to the corresponding language it is enough to **check the bottom-right cell of the parse table**: if it contains the starting nonterminal then the sequence is recognized and we can proceed with the construction of the optimal parse tree, otherwise it's not. For example, the previous table contains S in the bottom-right cell so the sequence `abaab` belongs to $L(G)$.

8.1.3 *Producing the optimal parse tree*

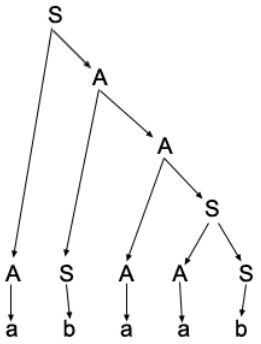
Once we're sure that $w \in L(G)$ and we have its parse table, we can reconstruct its optimal derivation tree (or one of the many optimal derivation trees if the grammar is ambiguous). We have two solutions:

- Store links while populating the parse table. This way, once the table is completed, we can traverse it to obtain the corresponding tree.
- Recompute the leftmost derivation tree for w using a method very similar to the *edit path* seen for the Needleman-Wunsch algorithm.

The **second method** works as follows:

- Start from the leaves: each symbol of the sequence corresponds to a **leaf of the parsing tree**;
- For each leaf go up one level, associating the nonterminal of the corresponding production rule;

- Group together the rightmost couple of nonterminals and insert, as a parental node, the nonterminal corresponding to their production rule: for example, if the rightmost couple is $\{AS\}$, S is a valid parental node. Notice the ambiguity: A would've been another exact solution. Iterate this step until a root is found.



8.1.4 *Complexity of CYK*

Space-wise, CYK needs to store the parse table, which is n^2 and contains up to $|V_N|$ elements: $S(|V_N|n^2)$. Time-wise, all the parse trees are obtained in $O(n^3)$. Notice that these complexities are **too high for practical applications**. If the grammar is not ambiguous, however, **complexity can be reduced** (only one parse tree is produced).

8.2 WEIGHTED GRAMMARS

Formal languages are *exact* definition techniques while the biological world has many exceptions. How can we model this variability with a grammar? A naive solution would be to relax our formal definitions, adding more options: for example, if we have this regular expression describing a set of sequences...

[RK]-G-{EDRKHPCG}-[AGSCI]-[FY]-[LIVA]-x-[FYM]

...new options could be added as follows:

[RK] $\rightarrow$ [RKNADOF...]

However, this approach makes the regular expression **less informative**. The actual solution is to use grammars whose production rules have an associated weight. This way, it is possible to assign a higher

weight to production rules modelling standard situations and a lower weight to production rules modelling instead the "exceptions".

A weighted grammar is a context-free grammar where **each production has an associated weight**. Formally it's defined as:

$$G = (V_N, V_T, P, S, J, M, \mu)$$

...where V_N is the set of nonterminals, V_T is the set of terminals, P is the set of production rules with S as the starting symbol, J is the set of labels associated to each production rule, M is the **weight space** and $\mu : J \to M$ is the **weight function** associating a weight to each labelled production rule. An example of type-3 weighted grammar is:

```
VN = {S,A}
VT = {a,b}
M = Z
P:
   (1) S -> aS   1
   (2) S -> aA   1
   (3) A -> bA   -1
   (4) A -> b    -1
```

8.2.1 *Derivations with weighted grammars*

With weighted grammars derivations are **still chains of production rules**: the only difference is that each time we apply a production rule we must keep track of its weight.

$$S \xrightarrow{\mu(r_1)} \alpha_1 \xrightarrow{\mu(r_2)} \alpha 2 \ldots \xrightarrow{\mu(r_m)} x$$

Examples of possible derivations are the following:

```
  +1    +1     +1      -1       -1
S -> aS -> aaS -> aaaA -> aaabA -> aaabbA
  r1    r1     r2      r2       r4

  +1    +1     +1      -1       -1        -1
S -> aS -> aaS -> aaaA -> aaabA -> aaabbA -> aaabbb
  r1    r1     r2      r3       r3        r4
```

8.2.2 *Weight of a derivation*

Each derivation has a corresponding weight that can be computed as follows:

$$\mu(r_1) * \mu(r_2) * \ldots * \mu(r_m)$$

...where $*$ is an **associative and commutative operation** since we should get the same weight no matter the order in which production rules are used.

8.2.3 *Weight of a sequence*

If the weighted grammar is ambiguous we know that the same sequence can have many different derivations, each with a corresponding weight. Given a grammar G, the overall weight associated to a string x is therefore:

$$\mu_G(x) = V[\mu(r_1) * \mu(r_2) * \ldots * \mu(r_m)]$$

...where V is applied to all the leftmost derivations of x in G. A common choice for $*$ and V are the sum and the product respectively. As an example consider the string `aaabb` in the previous grammar: since there's only one derivation for it, the weight associated to this string is $1+1+1-1-1-1=-1$.

8.2.4 *Languages defined by weighted grammars*

Languages can be generated using a weighted grammar and a threshold value $\lambda \in M$: for example, a language whose sequences have weight at least λ are:

$$L(G, \lambda) = \{x | x \in L(G), \mu_G(x) > \lambda\}$$

Languages whose sequence have λ as the exact weight are instead:

$$L(G, =, \lambda) = \{x | x \in L(G), \mu_G(x) = \lambda\}$$

Examples are $L(G, 0) = \{a^m b^n | m > n > 0\}$ and $L(G, =, 0) = \{a^n b^n | n > 0\}$.

8.2.5 *Other properties of weighted grammars*

Weighted grammars are particularly interesting because they allow us to use techniques developed for simple languages (type 1 or 2) even for more complex languages: in fact, a type-3 grammars becomes a type-2 or type-1 grammar if weights are added; furthermore, a type-2 grammar becomes a type-1 grammar if weights are added. Of course it is necessary to properly adapt such techniques to the weights.

8.3 STOCHASTIC GRAMMARS

Stochastic grammars are **weighted grammars where weights are probabilities**: each sentence produced by a stochastic grammar has an associated probability (a **stochastic language** is created, in other words). Formally, a stochastic grammar is defined as the couple $(G, \mathbb{P})$ where:

- G is a context-free grammar with a set of nonterminals $V_N = \{A_1 ... A_i ... A_k\}$, a set of terminals and productions P_{ij}, $1 \leqslant j \leqslant n_i$;
- $\mathbb{P}$ is a set of probabilities p_{ij}, each of which is associated to exactly one production P_{ij}. A production rule with probability 0 can be deleted.

Remember that, since we're working with probabilities, each p_{ij} is a non-negative value between 0 and 1, and that the sum of all probabilities is 1. An example of stochastic grammar is the following:

$G = (V_N, V_T, P, S)$, $P = \{p_{ij}\}$
$V_N = \{S, A\}$, $V_T = \{a\}$
P:

- $S \to SS, p_{11} = 3/4$;
- $S \to A, p_{12} = 1/4$;
- $A \to aA, p_{21} = 2/3$;
- $A \to a, p_{22} = 1/3$

8.3.1 *Probability of a derivation*

Given a stochastic grammar G what is the probability associated to a certain derivation? To answer we must remember that we're working with **context-free** stochastic grammars: as a result, the probability of

certain derivation is simply the product of the probabilities associated to the production used in the derivation[2]. With a formula, if this is our derivation...

```
   ph1k1      ph2k2           phnkn
S ------>  α2 ------>  α2 ... ------> αn = x
   Ph1k1      Ph2k2           Phnkn
```

...then $p(\delta) = \prod_{i=1}^{n} P_{hi,ki}$. Notice that **the longer the derivation, the more its probability tends to 0**. As an example, if the derivation of aaa is:

```
  3/4    3/4     1/4     1/3     1/4     1/3     1/4     1/3
S --> SS --> SSS --> SSA --> SSa --> SAa --> Saa --> Aaa --> aaa
  P11    P11     P12     P22     P12     P22     P12     P22
```

...then the probability of this derivation is:

$$p(\delta) = 3/4 * 3/4 * 1/4 * 1/3 * 1/4 * 1/3 * 1/4 * 1/3 = 9/27648$$

8.3.2 *Probability of a sequence*

Suppose we have a string x produced by a stochastic grammar G: what is the probability associated to the sequence? If the grammar is ambiguous then x might have many different derivations, each with a certain probability: then, it is enough to **sum the probabilities of all derivations**[3]. With a formula, if x has derivations $\delta_1 ... \delta_m$ then the probability of x is:

$$p(x) = \sum_{i=1}^{m} p(\delta_i)$$

If we consider the previous grammar, the sequence aaa has four different derivations each with a certain probability:

```
S --> SS --> SSS --> SSA --> SSa --> SAa --> Saa --> Aaa --> aaa
```

2 This is because **each production is considered an "independent event"**: the probability of an event (the sequence) composed by independent events (the productions) is the product of the probabilities of the independent events.

3 This is because each derivation is considered as a "different way" to obtain the same event.

```
S -> SS -> SA -> SaA -> Saa -> Aaa -> aaa
```

```
S -> SS -> SA -> Sa -> Aa -> aAa -> aaa
```

```
S -> A -> aA -> aaA -> aaa
```

Therefore the probability of aaa is:

- $p(\delta_1) = 3/4 * 3/4 * 1/4 * 1/3 * 1/4 * 1/3 * 1/4 * 1/3 = 9/27648$
- $p(\delta_2) = 3/4 * 1/4 * 2/3 * 1/3 * 1/4 * 1/3 = 6/1728$
- $p(\delta_3) = 3/4 * 1/4 * 1/3 * 1/4 * 2/3 * 1/3 = 6/1728$
- $p(\delta_4) = 1/4 * 2/3 * 2/3 * 1/3 = 4/106$

$$p(aaa) = \sum_{i=1}^{4} p(\delta_i) = 9/27648 + 6/1728 + 6/1728 + 4/106 = 0,044307$$

8.3.3 *Stochastic languages*

Stochastic grammars $(G, \mathbb{P})$ induce stochastic languages – languages where each string has an associated probability. Given an alphabet, a stochastic language is defined as:

$$(L, \phi)$$

...where L is the language defined on the alphabet and $\phi : V_T^* \to \mathbb{R}$ is the **probability function** associating to each sequence a certain real probability. Notice that the probability function verifies the **usual conditions**: every string has a non-negative probability between 0 and 1, the sum of all probabilities is 1, if a string has probability 0 then it does not belong to the language. As an example, a stochastic language is:

$V_t = \{a, b\}$
$L = \{a^n b^n | n \geqslant 0\}$ $\phi(a^n b^n) = 1/(e * n!)$ and $\phi(x) =$ 0for$x \notin L$.

Stochastic languages can also be generated using a **threshold value** $\lambda \in [0, 1]$: for example $L(G, \lambda) = \{x | x \in L(G), \mu_G(x) > \lambda\}$ is the language whose sequences have at least probability λ. Notice that the **threshold** 0 **is always interesting** for stochastic languages.

8.3.4 *Relation between stochastic grammars and stochastic languages*

So far we've seen how to define both stochastic grammars and stochastic languages. However, **not every stochastic grammar can define a stochastic language (and viceversa)**:

- **From grammars to languages**. A stochastic grammar defines a stochastic language only if the probability space $\mathbb{P}$ satisfies the previous conditions seen before and, in particular, the third condition ($\sum_{x \in L} p(x) = 1$). This is **not always true**!
- **From languages to grammars**. Given a stochastic language we're **not guaranteed that there exist at least one stochastic grammar to generate it**. A counter-example is the following: if the **language** L **is stochastic** and defined as $L = \{a^n b^n | n \geqslant 0\}$, then there's no type-2 grammar that can generate L with probability function $\phi(a^n b^n) = 1/(e * n!)$ – which means that the corresponding grammar is *not* stochastic.

8.4 CONSISTENT STOCHASTIC GRAMMARS

Given a stochastic grammar $(G, \mathbb{P})$, it is consistent if and only if the **sum of the probabilities of all sequences** $x \in L(G)$ **is 1**:

$$\text{if } \sum_{x \in L} p(x) = 1 \text{ then the grammar is consistent}$$

Inconsistency may be generated by **circling derivations**, where the same sequence can be generated by longer and longer chains of repeated derivations. Since longer derivations have lower probabilities, in such situations the probability tends to 0.

$$S \to A \to B \to S$$

To verify if we have a cycle we must simplify the grammar using methods for stochastic grammars[4]. Another possible cause of inconsistency is **unbalanced probabilities**, when there's a **divergency in the way probabilities are distributed to productions**:

```
G:  A -> AA (99/100)
    A -> a (1/100)
```

4 Note that if a type-3 grammar does not have redundancy then it's **guaranteed to be consistent**.

8.4.1 *Deciding consistent stochastic grammars*

How can we **decide whether a stochastic grammar is consistent**? We already know that a type-3 stochastic grammar with no cycles is consistent, but what can we way about type-2 stochastic grammars? We need to work with **two matrices**:

- **Probability matrix** $Q_{|V_n| \times |P|}$. Simply put, Q maps each left-side nonterminal with its corresponding probability: Q_{ij} contains the probability p_j if variable A_i is on the left of production P_j, otherwise 0. It represent the probabilities of the various rewritings for each variable. As an example, suppose we have these productions: the corresponding probability matrix Q is listed below.

```
P: E -> aEET     p1 = 0.1
   E -> T        p2 = 0.9
   T -> bE       p3 = 0.5
   T -> c        p4 = 0.5
```

Q	P_1	P_2	P_3	P_4
E	0.1	0.9	0	0
T	0	0	0.5	0.5

- **Occurrence matrix** $C_{|P| \times |V_n|}$. Simply put, C is the table mapping probabilities with the corresponding nonterminals on the right of production rules: C_{ij} is the number of times of variable A_i in the right side of production P_j. C represents variable occurrences generated by the various productions. Again, given the previous production rules, the corresponding occurrence matrix is:

C	E	T
p_1	2	1
p_2	0	1
p_3	1	0
p_4	0	0

By multiplying $Q \times C$ we obtain the **stochastic expectation matrix $\mathbb{A}$ of a stochastic grammar**, a square $|V_N| \times |V_N|$ matrix where element $\mathbb{A}_{ij}$ can be seen as how many occurrences of A_j we expect when a single production is applied to A_i.

$$\mathbb{A}_{|V_N|\times|V_N|} = Q \times C$$

Getting back to our previous example, the resulting stochastic expectation matrix is showed below. Notice that $\mathbb{A}_{1,1} = 0.2$, which means that we expect 0.2 E when rewriting E.

$$\mathbb{A} = Q \cdot C = \begin{pmatrix} 0.1 & 0.9 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 \\ 0.5 & 0.5 \end{pmatrix} \begin{pmatrix} 2 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{pmatrix} =$$

$$= \begin{pmatrix} 0.2 & 1 \\ 0.5 & 0 \end{pmatrix}$$

$\mathbb{A}$ can be used to decide whether a stochastic grammar is consistent or not: if the absolute value of the largest eigenvalue of $\mathbb{A}$, $\rho(\mathbb{A})$, is < 1 then the grammar is consistent. Since this method is rather costly for large stochastic expectation matrices we can use an **equivalent sufficient condition**: the absolute value of the largest eigenvector of $\mathbb{A}$ is < 1 if and only if there exists a $n \geqslant 1$ such that the sum of the absolute values of the elements in any row of M^n is < 1[5].

$$\forall i \in [1, m] \sum_{j=1}^{m} |(M^n)_{ij}| < 1$$

To decide the consistency of a stochastic grammar is enough to multiply $\mathbb{A}$ for itself (in such a way that n is a power of 2) and check the sum of each row: if they're all below 1 then the sufficient condition is met and the stochastic grammar is consistent, otherwise we continue with the multiplications. Note that the **procedure might not terminate** as **consistency is a semi-decidable problem** for stochastic grammars. As an example, consider the following stochastic expectation matrix:

$$\mathbb{A} = \begin{pmatrix} 0.2 & 1 \\ 0.5 & 0 \end{pmatrix}$$

The sum of the first row is 1.2, the sum of the second is 0.5 so the sufficient condition is not verified. Let's multiply the matrix by itself:

$$\mathbb{A}^2 = \begin{pmatrix} 0.54 & 0.20 \\ 0.10 & 0.50 \end{pmatrix}$$

The sum of the first row is 0.74 and the sum of the second is 0.60 hence the grammar is consistent.

5 As a **corollary**, if we find n then any $n' > n$ has the same property.

9

STOCHASTIC AUTOMATA AND WEIGHTED RECOGNIZERS

So far we've seen finite automata, simple and efficient parsers of seqnences generated by regular grammars. Can we we still use finite automata to parse sequences produced by weighted or stochastic languages?

9.1 WEIGHTED FINITE AUTOMATA

Weighted finite automata are simple automata used in many contexts, from natural language processing to molecular biology.

9.1.1 *Sequential translators (from string to weight)*

A first example of weighted finite automata are **sequential translators**, finite automata that, given a sequence, produce the corresponding weight (a combination of the weights in output at each transition)[1]. Formally they're defined as

$$T = (\epsilon, Q, I, F, E, \lambda, \rho)$$

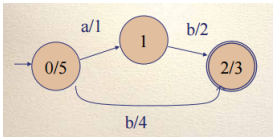
...where ϵ is the alphabet, Q is the set of states (with $I \subset Q$ initial states and $F \subset Q$ final states) and E is the set of transitions. λ and ρ are **two functions**:

- $\lambda : I \to \mathbb{R}^+$ is used to assign weights to the initial states;
- $\rho : F \to \mathbb{R}^+$ is used to assign weights to the final states;

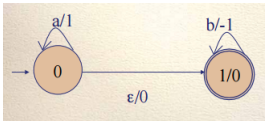
The example below shows a deterministic sequence translator with only 3 states. Notice that only transitions, final and initial states are weighted: 0/5 is the initial state 0 with weight 5, 1 is the state 1 with no weight, 2/3 is the final state 2 with weight 3. As for the transitions, a/1 is the transition triggered by a and with weight 1, etc. Furthermore, since the automaton is deterministic, the weight

1 Notice that if the finite automata is **non-deterministic** we need to define how to combine the weights obtained on the different paths recognizing the input.

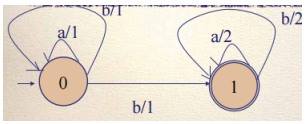
of string ab can be obtained as the **sum of the weights**, both in the states and in the transitions: $ab \to 5+1+2+3=11$.



Often, transition weights are more relevant than initial/final weights: in the following automaton, capable of recognizing strings $a^n b^n$ plus the empty string, the initial and final weights are 0.



A more complex translator is shown below: it's capable of associating to binary strings their corresponding decimal value (a encodes 0 and b encodes 1).



9.2 STOCHASTIC AUTOMATA

Suppose now that the operations to obtain the weight associated to a certain sequence are:

- $*$: **sum of the output weights** (transitions, initial and final states);
- V: **minimization**, to find the path with the lowest weight.

In this case the sequence translator **corresponds to a stochastic grammar**: the resulting weight is interpreted as the **negative logarithm of the probability** of a sequence and minimizing over all possible paths means finding the most probable (and therefore best) derivation for that sequence.

Now that we've bridged the gap between stochastic grammars and finite automata we can try to solve **three problems**:

- **Optimal alignment problem**: given a sequence x produced by a stochastic grammar G, what is the best derivation (with the highest probability)?

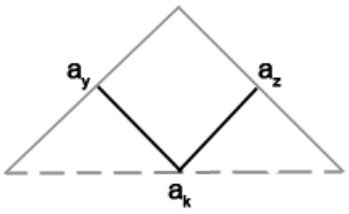
- **Scoring problem**: given a sequence x produced by a stochastic grammar G, what is the overall probability of x in G?
- **Training problem**: given a stochastic grammar G and set of independent sequences $x_1 ... x_m$, how can we use them to readjust the original probabilities of the stochastic grammar?

9.3 BEST DERIVATION PROBLEM: OPTIMAL ALIGNMENT ALGORITHM

Given a sequence w and a stochastic grammar $(G, \mathbb{P})$, the optimal alignment algorithm finds the optimal derivation tree of w in G – the derivation tree with the highest probability. The **input** is composed by two elements:

- The sequence $w = a_1 ... a_n$;
- The stochastic grammar $(G, \mathbb{P})$ where:
  - G is a stochastic grammar with V_N nonterminals, V_T terminals and P production rules (starting nonterminal A_1). Since this algorithm is a variant of CYK G is in **Chomsky normal form and has no empty productions**.
  - $\mathbb{P} = (R_+ \cup \infty, +, min, 0, \infty)$ is a set of probabilities associated to each production rule: $t_v(y, z)$ for binary production rules $A_v \rightarrow A_y A_z$, $e_v(a)$ for unary production rules $A_v \rightarrow a$. Notice that these probabilities are **non-negative real values** combined together using the sum and the minimization operator, exactly as requested.

As for the **output**, the algorithm returns the **probability table** T, a variant of the CYK parse table where the element $t(i, j, v)$ represents the negative logarithm of the most probable parse of $A_v \rightarrow a_i ... a_j$. A reconstruction link to rebuild the best derivation tree, $r(i, j, v)) = (y, z, k)$ is also stored.



9.3.1 *The algorithm*

As expected, the algorithm is based on the dynamic programming approach:

- **Initialization**: if $j = 1$ then the algorithm computes for every nonterminal v the negative algorithm of the probability of the most probable parse of $A_v \rightarrow^* a_i$. Notice that we always consider substrings of length 1. We also store "starting" reconstruction links, $(0, 0, 0)$.

```
for i=1 to n
    for v=1 to m
         t(i,i,v) := -log e_v(a_i);
         r(i,i,v) = (0, 0, 0)
```

- **Inductive step**: if $j > 1$ then the algorithm computes the negative algorithm of the probability of the most probable parse of $A_v \rightarrow^* a_i ... a_j$. Every substring of every length (> 1) and every nonterminal v is considered. In other words, this step check every possible parse tree of the subsequence $a_i ... a_j$ and stores the negative logarithm of the one with the highest probability.

```
for i:= n-1 downto 1 do
  for j:= i+1 to n do
    for v:= 1 to m do
      t(i,j,v) = min_(y,z) min_(k=i...j-1)
                  {t(i,k,y) + t(k+1,j,z) - log t_v(y,z)}
      r(i,j,v)= argmin_(y,z,k)
                 {t(i,k,y)+ t(k+1,j,z) - log t_v(y,z)}
```

Once T is populated the negative logarithm of the probability of the most probable parse tree is stored in $t(1, n, 1)$ – the usual bottom-right cell of the table. To reconstruct the optimal parse tree it is enough to follow inductively the reconstruction links starting from $t(1, n, 1)$.
To conclude, the **complexity** of this algorithm is the same as CYK: $S(|V_N|n^2)$ space-wise, $O(n^3)$ time-wise.

9.4 INTRODUCTION TO THE INSIDE-OUTSIDE ALGORITHM

In the next pages we will study two algorithms that, taken separately, perform the same task: given a sequence w and a stochastic grammar

$(G, \mathbb{P})$, compute the probability of w in G taking into account that w can be generated by many different derivation trees. Since the naive approach – produce all derivation trees of w and multiply their probabilities – would be an exponential problem, algorithm are instead used. Despite this, the *Outside* algorithm is more complex and rarely used to solve the scoring problem; instead, it is combined with the *Inside* algorithm to solve the training problem.

9.5 INSIDE ALGORITHM

The Inside algorithm takes in **input** the usual sequence $w = a_1 ... a_n$ and a stochastic grammar $(G, \mathbb{P})$ in Chomsky normal form with no empty productions. G is basically the same as the previous algorithm; $\mathbb{P}$, instead, is defined as follows:

$$\mathbb{P} = ([0, 1], *, +, 1, 0)$$

Exactly as before every production rule has an associated probability: $t_v(y, z)$ for binary productions $A_v \to A_y A_z$ and $e_v(a)$ for unary productions $A_v \to a$; still, these probabilities are all belonging to $[0, 1]$ and the operations to combine them are $\times$ (production of probabilities of different production rules) and $+$ (sum of probabilities of different derivations). As for the **output**, the Inside algorithm produces a probability table In where $In(i, j, v)$ is the probability of $A_v \to^* a_i ... a_j$.

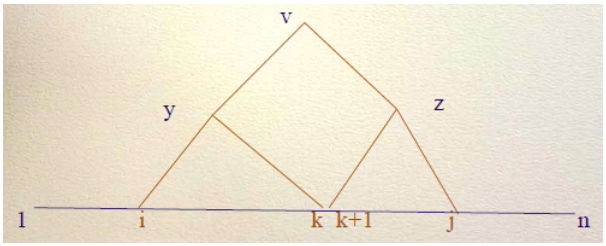
9.5.1 *The algorithm*

Since the Inside algorithm is another variation of CYK the algorithm is based on the **dynamic programming approach**:

- **Initialization**: if $j = 1$ then we simply populate the bottom row of In with the probability of the production producing each symbol a_i: $In(i, i, v) = e_v(a_i)$.

```
for i:= 1 to n do
   for v:= 1 to m do In(i, i, v) := ev (ai);
```

- **Inductive step**: if $j > 1$ then we inductively compute the probability of every parse tree for every substring of w. It's easier to understand if we look at this picture:



For every possible subsequence of w, the algorithm computes inductively the probability of all left and right subtrees rooted in y and z and then multiplies it by the probability of "adding" a new root v to the structure: $t_v(y, z)$.

```
for i:= 1 to n-1*
    for j:= i+1 to n
       for v:= 1 to m do
           In(i,j,v) = sum_{y=1}^m sum_{z=1}^m sum_{k=i}^{j-1}
           In(i, k, y) In(k+1, j, z) t_v(y, z);
```

Once In is filled, the overall probability of w is stored in position $p(w) = in(1, n, 1)$, the usual bottom-right of the sequence. Complexity is a little different than CYK: $S(n^2)$ space-wise and $O(n^3)$ time-wise.

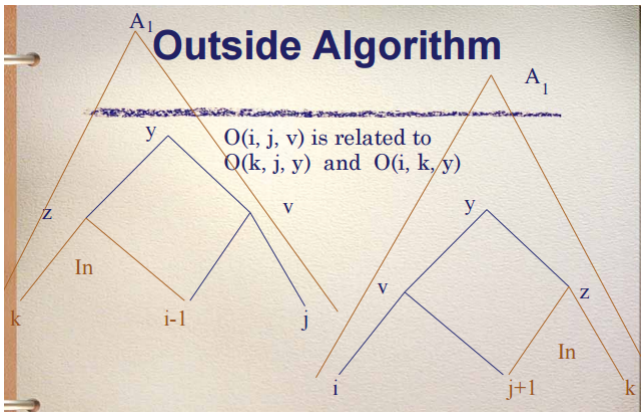
9.6 OUTSIDE ALGORITHM

The *Outside* algorithm can be used to solve the scoring problem but, since it would be redundant here, it's more interesting to see how it can be combined with the Inside algorithm to solve the **training problem**. Let's start from the **input**:

- A sequence $w = a_1...a_n$, exactly as before;
- A stochastic grammar $(G, \mathbb{P})$ with the same features as the one in the Inside algorithm; $\mathbb{P}$ still contains the same probabilities as before, divided into binary and unary productions;
- The In table, produced applying the Inside algorithm to w with the same stochastic grammar.

As for the **output**, the algorithm produces a probability table O where $O(i, j, v)$ is the probability of $A_1 \rightarrow^* a_1...a_n$ without considering all the subtrees for $A_v \rightarrow^* a_i...a_j$. Basically the algorithm uses

a **top-down approach**: first it computes the probability of an almost-empty tree with only the root and then it recursively fills this structure, excluding smaller and smaller subtrees until the complete tree is considered.



9.6.1 *The algorithm*

Again, this is a dynamic programming algorithm based on CYK so:

- **Initialization**: we consider the largest almost-empty subtree (with only a root) and initialize the corresponding cell $t(1, n, 1)$ to 1, the neutral element for the product, while everything else is at 0.

```
O(1, n, 1) := 1;
for i:= 2 to m
O(1, n, i) = 0;
```

- **Inductive step**. The previous tree is filled inductively using the information taken from the In table: we consider all the possible subsequences of w, $a_i ... a_j$, from the longest to the shortest and, for each possible root v of the corresponding subtree, compute $O(i, j, v)$.

```
for i:= 1 to n , j:= n downto i , v:= 1 to m do
O(i, j, v) =
   ϵ_y,z ϵ_k=1^i-1 In(k, i-1, z) O(k, j, y) ty(z, v) +
ϵ_y,z ϵ_k=j+1^n In(j+1, k, z) O(i, k, y) t_y(v, z);
```

Notice that the algorithm fills the empty tree on the left (k to $i-1$) and on the right ($j+1$ to k), taking the data from the In

table; since the same substring can be derived in many different ways, we consider all possible cases ($\sum$ over y and z).

Once O is filled we can solve the scoring problem for w by computing the sum below[2]

$$p(w) = \sum_{v=1}^{m} O(i, i, v) e_v(a_i) \text{ for any } i \tag{2}$$

In other words the information is stored in the diagonal positions for every possible choice of the root (for *any* i). Finally, complexity is the same as before: $S(n^2)$ space-wise, $O(n^3)$ time-wise.

9.7 THE TRAINING PROBLEM

The combination of the *Inside* and *Outside* algorithm is used to solve the training problem, which can be formulated as follows. Given in **input**...

- A stochastic grammar $(G, \mathbb{P})$ and a corresponding stochastic language (L, ϕ);
- A training set $W = \{w_1, w_2, \ldots, w_n\}$ where $w_i \in L(G)$. Notice these sequences are assumed **independent**.

...how can we update the probabilities of the productions of G so that the **probability of generating the training set is maximized?** The answer comes from the *Inside-Outside algorithm*, an expectation maximization algorithm developed by Larry-Young (1990). Notice that the algorithm does not modify the structure of G, only its production probabilities.

9.7.1 *The algorithm*

Let's consider a single sequence $a_1 ... a_n$ from the training set: by computing its probability tables In (with the Inside algorithm) and O (with the Outside algorithm) we obtain two **complementary data sets**:

2 Interestingly this is not the goal of the Outside algorithm, at least here; still, the information can be extracted from the table.

- In contains in $In(i,j,v)$ the probability of $A_v \rightarrow a_i...a_j$;
- O contains in $O(i,j,v)$ the probability of $A_1 \rightarrow a_1...a_n$ excluding all the subtrees $A_v \rightarrow a_i...a_j$;

With this data we can compute:

- Expected number of times a nonterminal A_v is used in the derivation of w:

$$c(v) = \frac{1}{p(w)} \sum_{i=1}^{n} \sum_{j=i}^{n} In(i,j,v)O(i,j,v)$$

  Since v can be the root of any subtree of the derivation tree of w, we need to consider all possible starting positions (i) and lengths of w (j). The result is also normalized (division by the overall probability of w, which is extracted by In).

- Expected number of times $A_v \rightarrow A_y A_z$ is used in a derivation of w:

$$c(v \rightarrow yz) = \frac{1}{p(w)} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=i}^{j-1} O(i,j,v)In(i,k,y)In(k+1,j,z)$$

  $$t_v(y,z)$$

  For any occurrence of the production rule we compute its probability times all the possible Inside and Outside probabilities.

With this information it's possible to recompute the probability of a binary production $A_v \rightarrow A_y A_z$:

$$t'_v(y,z) = \frac{c(v \rightarrow yz)}{c(v)}$$

...and the probability of a unary production $A_v \rightarrow a$:

$$e'_v(a) = \frac{c(v \rightarrow a)}{c(v)}$$

Since the sequences in the training set are independent we can sum together the new probabilities. Notice that this process is repeated until the new probability is almost similar to the previous one (using a threshold). To conclude, the *Inside-Outside* algorithm can be

used both to assign new probabilities to a stochastic grammar and to refine probabilities already given. It is, however, very **costly** since each application of the Inside-Outside algorithm requires $O(n^3)$, and there's **no guarantee to obtain the optimum probabilities**. There are, however, many variants to improve it.

9.7.2 *An example on the Inside-Outside algorithm*

Suppose we have a very simple stochastic grammar:

```
S -> aS    3/4
S -> b     1/4
```

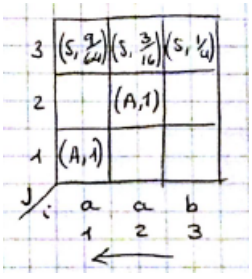
We want to determine the probabilities of the Inside-Outside algorithm for a training set of just one sequence, $W = \{aab\}$. First of all we must **transform the grammar** into a Chomsky normal form with no empty productions. This is done by adding a new rule:

```
S -> AS
A -> a
S -> b
```

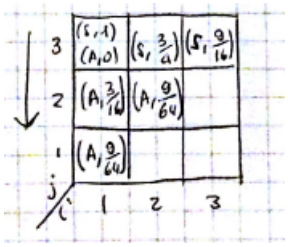
"Untouched" production rules will maintain their previous probability; the added production rule, instead, will have a **trivial probability** of 1, the neutral element of production.

```
S -> AS    3/4
A -> a     1
S -> b     1/4
```

To update the production probabilities we apply the *Inside-Outside* algorithm. Suppose these are the In and O tables respectively:



Let's compute $c(S)$: we need to traverse In and O varying the values of i (from 1 to n) and j (from i to n). Since we're interested in S, which is fixed in the formula, we have three positions in both



tables we're interested in – everywhere we have S: $\text{In}(i = 2, j = 3, S)\text{O}(i = 2, j = 3, S)$, $\text{In}(i = 2, j = 3, S)\text{O}(i = 2, j = 3, S)$ and $\text{In}(i = 3, j = 3, S)\text{O}(i = 3, j = 3, S)$. The result is:

$$c(S) = \left(\frac{9}{64} * 1 + \frac{3}{4} * \frac{3}{16} + \frac{1}{4} * \frac{9}{16}\right) \times \frac{64}{9} = 3$$

Note that everything's multiplied by $\frac{1}{p(w)}$; since $p(w) = \frac{9}{64}$ we have the previous result. We can also compute in the same way $c(A)$: since $v = A$ is fixed, we're interested in $\text{In}(i = 1, j = 1, A)\text{O}(i = 1, j = 1, A)$ and $\text{In}(i = 2, j = 2, A)\text{O}(i = 2, j = 2, A)$):

$$c(A) = \left(\frac{9}{64} + \frac{9}{64}\right) \times \frac{64}{9} = 2$$

Finally, we compute $c(S \to AS)$. $v = S, y = A, z = S$ are fixed and therefore, by applying the previous formula, we must multiply:

$$c(S \to AS) = \left(\frac{9}{64} + \frac{9}{64}\right) * \frac{64}{9} = 2$$

This means that the production rule $S \to AS$ will be used twice. We can recompute its overall probability:

$$\frac{S \to AS}{c(S)} = 2/3$$

...which is different than the previous 3/4.

10

MARKOV AND HIDDEN MARKOV MODELS

Markov and *Hidden Markov* models are stochastic finite automata applied whenever **a signal has to be found among noise**. Aside from applications in speech and handwriting recognition and computer vision, Markov and Hidden Markov models have been widely employed in bioinformatics, for example to find gene, to recognize signals (promoters, splicing sites, ...), to perform classify families of proteins, etc.

10.1 MARKOV MODELS

Markov models are **stochastic finite automata** used to model random processes that undergo transitions from one state to another. They're composed by:

- A finite set of **states** $\{S_1, \ldots, S_n\}$, where each state S_i has an associated **initial probability** $\pi_i = P(S_i)$. Notice that **any state can be the initial state**.
- A finite set of **transitions** $S_i|S_j$, each with an associated probability $a_{ij} = P(S_i|S_j)$.

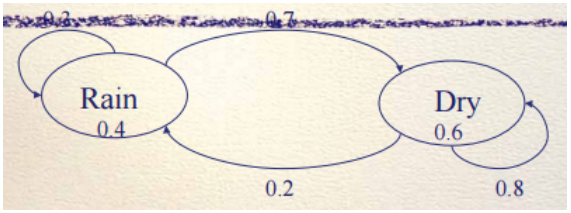
A process moves from one state to another, generating a sequence of states $S_{i1}, S_{i2}, \ldots, S_{i,k}, \ldots$. Interestingly, the **Markov chain property** assumes that the probability of the k-th state depend only on the probability of the $(k-1)$th state, and not on the probability of the sequence of $1...k-1$ states that preceded it.

$$P(S_{ik}|S_{i1}, S_{i2}, \ldots, S_{i,k}, \ldots) = P(S_{ik}|S_{ik-1})$$

An example of Markov model is showed below:

- States are "Rain" and "Dry";
- Probabilities associated to states are $P(\mathtt{Rain}) = 0.4$, $P(\mathtt{Dry}) = 0.6$;

- Transitions (and the relative probabilities) are $P('Rain'|'Rain') = 0.3$, $P('Dry'|'Rain') = 0.7$,$P('Rain'|'Dry') = 0.2$, $P('Dry'|'Dry') = 0.8$.



Notice that the sum of state probabilities is 1; furthermore the sum of the probabilities of the transitions exiting from the same state is 1.

10.1.1 *Evaluation problem with Markov models*

Suppose we want to compute the probability of a sequence of states, $P(S_{i1}, S_{i2}, ..., S_{ik})$. Thanks to the Markov chain property we know that the probability of a state depends *only* on the probability of the previous state; the probability of a chain of states is therefore:

$$P(S_{i1}, S_{i2}, ..., S_{ik}) = P(S_{ik}|S_{ik-1}P(S_{ik-1}|S_{ik-2})...P(S_{i2}|S_{i1})P(S_{i1})$$

As an example, the probability of sequence "Dry,Dry,Rain,Rain" is:

$$P('Dry','Dry','Rain', Rain') = P('Rain'|'Rain')$$

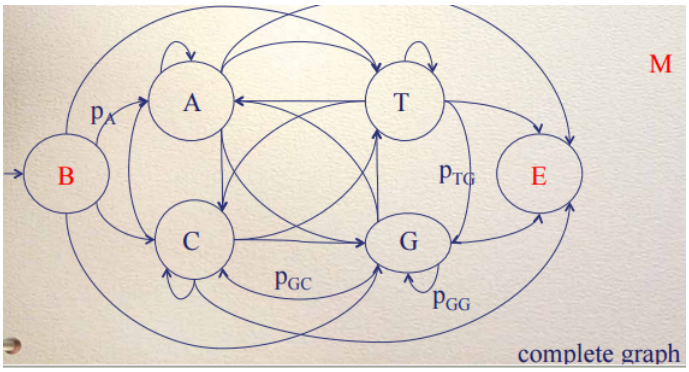
$$P('Rain'|'Dry')P('Dry'|'Dry')P('Dry') = 0.3 * 0.2 * 0.8 * 0.6$$

10.1.2 *Example of evaluation problem with Markov model: CpG Island*

In the human genoma the sequence `CG` is often transformed into `TG`: as a result, `CG` appears rarely in the genome. There are regions, called **CpG islands**, where `CG` sequences are instead frequent. To determine if a short sequence is part of a CpG island we produce two sets:

- A set of m sequences with CpG islands;
- A set of m sequences without CpG islands.

Next we create a Markov model for each set, M^+ and M^-: each has a starting state B, an ending state E and four additional states



A, C, T, G; Transitions represent to the different symbols found in the short sequence.

The transition probabilities of both models are computed as follows:

$$p_{st} = \frac{\text{how many couples st}}{\text{how many couples starting with s}}$$

Next, to understand if our short sequence $x = x_1 ... x_n$ contains a CpG island, it is enough to compute its score:

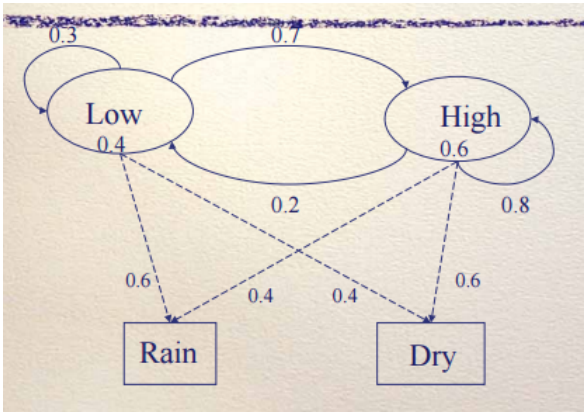
$$S(x) = \log \frac{P(x|M^+)}{P(x|M^-)}$$

The score is usually normalized by n. A threshold is used to discriminate sequences in M^+ and sequences in M^-.

10.2 HIDDEN MARKOV MODELS (HMM)

Hidden Markov Models are Markov Models where **noise in injected**: as a result the actual set of states modelling the process are not directly visible; what is visible is a set of **observations** $\{v_1, ... v_M\}$, visible states randomly generated by the invisible states. In general, any Hidden Markov model is defined as $M = (A, B, \pi)$ where:

- A is the matrix with transition probabilities, from one hidden state to another: $a_{ij} = P(S_i|S_j)$;
- V is the matrix with observation probabilities, from a hidden state to an observation: $b_i(v_m) = P(v_m|S_i)$.
- π is a vector with the initial probabilities of the hidden states, $\pi_i = P(S_i)$.



For example consider the following HMM:
We have:

- Two hidden states, "low" and "high" pressure, with probabilities 0.6 and 0.4 respectively;
- Two visible observations, "Rain" and "Dry";
- Probabilities of the transitions between the invisible states: $P(\text{'Low'}|\text{'Low'}) = 0.3$, $P(\text{'High'}|\text{'Low'}) = 0.7$, $P(\text{'Low'}|\text{'High'}) = 0.2$, $P(\text{'High'}|\text{'High'}) = 0.8$;
- Probabilities of observations: $P(\text{'Rain'}|\text{'Low'}) = 0.6$, $P(\text{'Dry'}|\text{'Low'}) = 0.4$, $P(\text{'Rain'}|\text{'High'}) = 0.4$, $P(\text{'Dry'}|\text{'High'}) = 0.3$.

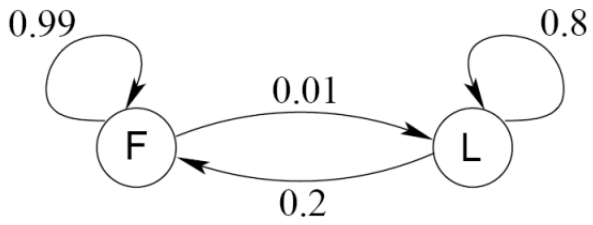
Another typical example of Hidden Markov Model is the so-called *dishonest casino*, where **two dices** are used: one is fair $P(1) = P(2) = P(3) = P(4) = P(5) = 1/10$ and another is loaded ($P(6) = 1/2$). The dealer switches between them with certain probabilities:

- $P(\text{Fair} \to \text{Loaded}) = 0.01$.
- $P(\text{Loaded} \to \text{Fair}) = 0.2$.

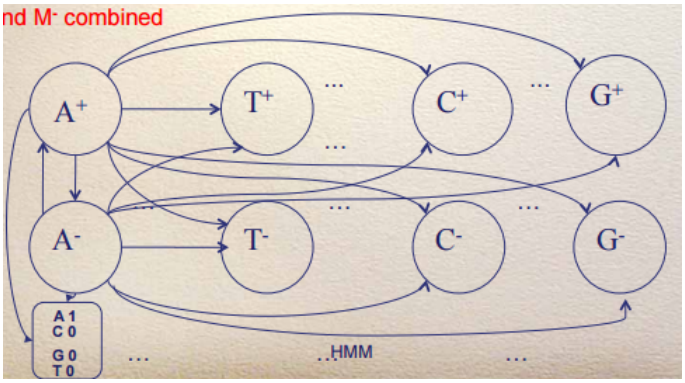
This situation is modelled as a Hidden Markov model where the various faces of the dice are the observations and the states are "Loaded" and "Fair":

10.2.1 *Optimal alignment problem: CpG islands and Hidden Markov Models*

Let's get back to the CpG islands: given a *long* DNA sequence, how can we determine if it contains a CpG island? This time we use a *Hid-*



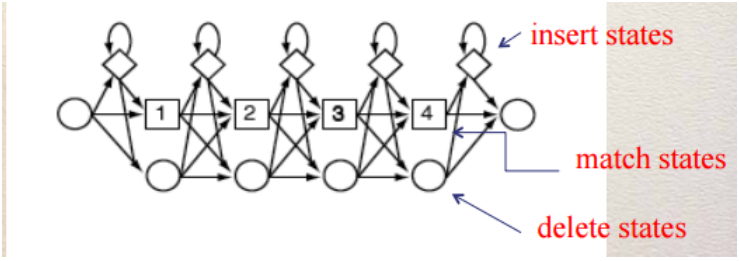
den Markov Model that combines the previous M^+ and M^- models in one:



We assign a low probability to the connections between the two models, a greater probability $M^+ \to M^-$ and a lower probability $M^- \to M^+$. Now, any nucleotide – for example C – of our sequence can be recognized either by C^+ or C^-. The problem is to determine the subsequences of x which are CpG islands and those who are not, namely the sequence of hidden states which best corresponds to the given sequence (has the highest probability).

10.3 PROFILE HMM

Profile HMM are used to determine if a protein belongs to a family: since profile HMM can be aligned, databases of profile HMMs are searched. Typically, a profile HMM is **derived from a multiple alignment** and has the following structure:



We have **three types of states**:

- A sequence of central **square states** representing each (relevant) column of the alignment; the actual number of square states may be different than the number of columns of the alignment: **columns that have less than half of the symbols are usually discarded**.
- **Diamond states** positioned inbetween square states representing insertions. Note: they're positioned even inbetween "start state, first square" and "last square, end state";
- **Round states** representing deletions, positioned under the square states;
- Two **initial and final** states are also inserted.

Transitions are all over the profile:

- Chain of transitions from the start state to the end state **passing through each square state**;
- Chain of transitions from the start state to the end state **passing through each circle state**;
- Diamond states have self-loop (more than one insertion in the same position of the sequence); they can also be seen as an "intermediate step" inbetween the first chain of transitions and can always lead to a deletion.

Probabilities are computed exclusively for the square states: we're interested in the probability of each central state plus in the probabilities of each transition exiting the central state. Generally speaking these probabilities are computed as **frequencies**: to **avoid overfitting pseudocounts are also used**.

- **Probability of square states**. Central states have 20 probabilities, one for each amino acid, to avoid overfitting. They're computed as frequencies and normalized using the Laplace smoother that adds one observation to each symbol. For example, suppose we align 7 sequences: if the first column has 5 Vs and 2 Fs the resulting probabilities – 5/7 and 2/7 – would cause overfitting: with the Laplace smoother, instead, we obtain $\frac{5+1}{7+20}$ for V, $\frac{2+1}{7+20}$ for F and $\frac{1}{27}$ for every other symbol.

- **Probability of transitions from square states**. They depend on the number of transitions exiting from the square state we're considering and on the destination of the transition:
  - Transition from square 1 to square 2: it's the ratio between the number of symbols in column 2 and the overall number of aligned sequences. The value is normalized by adding 1 to the numerator and the number of transitions exiting from state 1 in the denominator.
  - Transition from square i to a circle (deletion) state: it's the ratio between the number of gaps (now a gap, in the previous column a symbol) and the overall number of aligned sequences. The result is normalized as above.
  - Transitions from square i to diamond (insertion) state: it's the ratio between the number of insertions (now a symbol, in the previous column a gap) and the overall number of aligned sequences. The result is normalized as above.

To sum up, profile HMM are very powerful since they can be aligned with other sequences. As a downside, the inferred probabilities are usually overfitting so the problem at hand is not represented from a general point of view.

10.3.1 *Example of construction of HMM profile*

Suppose we have the following multiple alignment:

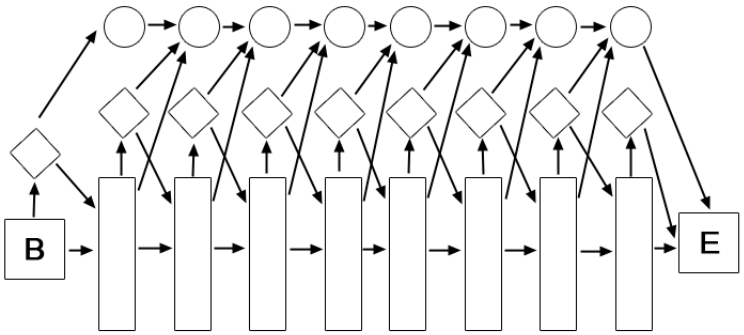
```
___VGA__HAGEY_____
___V____NUDEV_____
___VEA__DVAGH_____
___VKG______D_____
___VYS__TYETS_____
___FNA__NIPKH_____
___IAGADNGAGV_____
```

We start with the structure of the profile:

- Square states: we have 5 "start state" B, an "end" state E and 8 square states (two columns are not considered since they're mostly composed by gaps);
- Insertion (diamond) states are positioned inbetween all the previous states;

- Deletion (circle) states are positioned in correspondence with every square state (start and end excluded).

Transitions are all over the place: we basically follow the rules described above obtaining this result.



Let's compute some of the probabilities:

- Probability of state 1: we have 5 Vs, 1 F and 1 I so, using the Laplace smoother, we obtain: $P_V = 6/20$, $P_F = 2/20$, $P_I = 2/20$, $P_{else} = 1/20$.
- Probability of transition from state 1 to state 2: since we have 5 symbols, we obtain $P_{1,2} = (5+1)/(7+3) = 6/10$;
- Probability of transition from state 1 to deletion: since we have 1 gap we obtain $P_{1,D} = (1+1)/(7+3) = 2/10$;
- Probability of transition from state 1 to insertion: since we have 0 insertions we have $P_{1,I} = (0+1)/(7+3) = 1/10$.

10.4 MAIN PROBLEMS USING HMMS

Now that we've seen how Hidden Markov models are structured we can introduce the **three main problems** with Hidden Markov models:

1. **Evaluation problem** (or "scoring problem"). Given a HMM $M = (A, B, \pi)$ and a sequence of observations $O = o_1 ... o_K$, compute the probability that M generates O;
2. **Decoding problem** (or "optimal alignment"). Given a HMM $M = (A, B, \pi)$ and a sequence of observations $O = o_1 ... o_K$ compute the most probable sequence of hidden states that produces the observation sequence O;

3. **Learning problem** (or "training problem"). Given some training observation sequences $O = o_1...o_K$ and the general structure of HMM (number of hidden and visible states), determine A, B and π that best fit the data.

10.5 EVALUATION (OR SCORING) PROBLEM

Suppose we're given a Hidden Markov model $M = (A, B, \pi)$ and a sequence of observations $O = o_1...o_K$: what is the probability of M generating O? For example, let's get back to our previous case of study and assume we want to compute the probability of the sequence {"Dry","Rain"}. The Naive approach would compute the probability of every possible sequence of hidden states to produce that sequence:

$$
\begin{aligned}
&P(\text{'Dry'},\text{'Rain'}) = P(\text{'Dry'},\text{'Rain'},\text{'Low'},\text{'Low'}) + \\
&P(\text{'Dry'},\text{'Rain'},\text{'Low'},\text{'High'}) + \\
&P(\text{'Dry'},\text{'Rain'},\text{'High'},\text{'Low'}) + \\
&P(\text{'Dry'},\text{'Rain'},\text{'High'},\text{'High'})
\end{aligned}
$$

Of course, the more hidden states we have, the more this computation becomes unfeasible to solve: complexity is eventually **exponential**. For example, just the first term of the previous computation is the following:

$$
\begin{aligned}
&P(\text{'Dry'},\text{'Rain'},\text{'Low'},\text{'Low'}) = P(\text{'Dry'},\text{'Rain'}|\text{'Low'},\text{'Low'}) \\
&P(\text{'Low'},\text{'Low'}) = P(\text{'Dry'}|\text{'Low'})P(\text{'Rain'}|\text{'Low'}) \\
&P(\text{'Low'})P(\text{'Low'}|\text{'Low}) = 0.4 * 0.4 * 0.6 * 0.4 * 0.3
\end{aligned}
$$

The actual solution is to use the efficient **Forward HMM algorithm**, a dynamic programming algorithm that works as a counterpart of the *Inside* algorithm seen before for stochastic grammars. The idea behind the algorithm is to compute inductively a set of "forward probabilities" $\alpha_k(i)$: each represents the probability of being in state $q_k = S_i$ after having observed the prefix subsequence $o_1...o_k$ of the sequence.

$$\alpha_k(i) = P(o_1...o_k, q_k = S_i)$$

The name "forward" comes exactly from the idea of "moving forward", from the first to the last state. As for the cost, it is $O(N^2K)$ – linear with respect to the size of the sequence K and quadratic in the number of hidden states.

10.5.1 *Inductive definition*

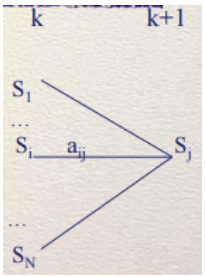
We perform induction on k, as per the dynamic programming algorithm:

- **Initialization**: if $k = 1$ then we need to compute the joint probability of sequence o_1 and of S_i being the "first" state of the sequence. This is the product of the observation probability of o_1 times the initial probability of S_i:

$$\alpha_1(i) = P(o_1, q_1 = s_i) = \pi_i b_i(o_1)$$

- **Inductive step**. Inductively we compute the probability of being in state $q_{k+1} = S_j$ having observed the subsequence $o_1 ... o_{k-1}$. It's computed as the the sum, for all possible states, $\alpha_k(i)a_{ij}$ times the observation probability of o_{k+1} needed to reach the last state S_j.

$$\alpha_{k+1}(j) = P(o_1 ... o_{k+1}, q_{k+1} = s_j) =$$
$$\sum_i P(o_1 ... o_{k+1}, q_k = s_i, q_{k+1} = s_j) =$$
$$\sum_i P(o_1 ... o_k, q_k = s_i) a_{ij} b_j(O_{k+1} =$$
$$[\sum_i \alpha_k(i)a_{ij}]b_j(O_{k+1})$$



- **Termination**. In the termination step we compute the overall probability of the sequence of observations; We simply sum the joint probability over all paths.

$$P(o_1 ... o_K) = \sum_i P(o_1 ... o_K, q_K = s_i) = \sum_i \alpha_K(i)$$

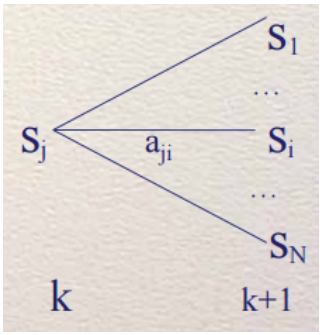
10.6 BACKWARD RECURSION FOR HMM

Counterpart of the *Outside* algorithm for stochastic grammars, the *Backward recursion* algorithm for HMM performs the **same task of**

the Forward algorithm but with an opposite approach: instead of performing the joint probability of "being in state i" and "having observed $o_1...o_k$", it computes the probability of the subsequence $o_{k+1}o_{k+2}...o_K$ *given* that at time k we're in state S_i. In other words we're dealing with a **conditioned probability**:

$$\beta_k(i) = P(o_{k+1}o_{k+1}...o_K|q_k = S_i)$$

The algorithm extends each probability computation going backward, working on **longer and longer suffixes** of the sequence.



10.6.1 *The algorithm*

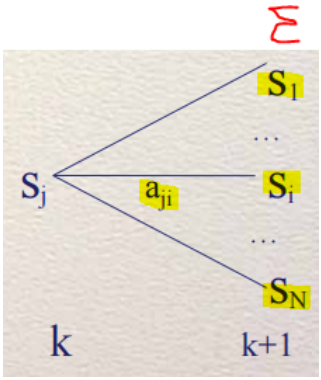
As before, this is a dynamic programming algorithm so it proceeds inductively:

- **Initialization**: in the initialization step we start from the smallest suffix, without K observations: since we haven't seen anything the probability $\beta_K(i)$ must be 1 (neutral element of product) whatever the state might be.

$$\beta_K(i) = 1$$

- **Inductive step**: The conditional probability of subsequence $o_{k+1}o_{k+1}...o_K$ given that at time k the state is S_i is given by three elements: the sum of all backward probabilities up to $k+1$ times a_{ij} times the observation probability $b_i(o_{k+1}$.

$$\begin{aligned}
&\beta_k(j) = P(o_{k+1}o_{k+2}...o_K|q_k = s_j) \\
&= \textstyle\sum_i P(o_{k+1}o_{k+2}...o_K, q_{k+1} = s_i|q_k = s_j) = \\
&\textstyle\sum_i P(o_{k+2}...o_K|q_{k+1} = s_i)a_{ji}b_i(o_{k+1}) = \\
&\textstyle\sum_i \beta_{k+1}(i)a_{ji}b_i(o_{k+1}) \text{ with } 1 \leqslant j \leqslant N, 1 \leqslant k \leqslant K-1
\end{aligned}$$



It becomes clear if we consider the picture above.

- **Termination**: in the end the overall probability of this sequence is the "accumulated" probability defined as:

$$P(o_1 o_2 ... o_K) = \sum_i P(o_1 o_2 ... o_K, q_1 = s_i) =$$
$$\sum_i P(o_1 o_2 ... o_K | q_1 = s_i) P(q_1 = s_i) =$$
$$\sum_i \beta_1(i) b_i(o_1) \pi_i$$

10.7 DECODING PROBLEM AND VITERBI ALGORITHM

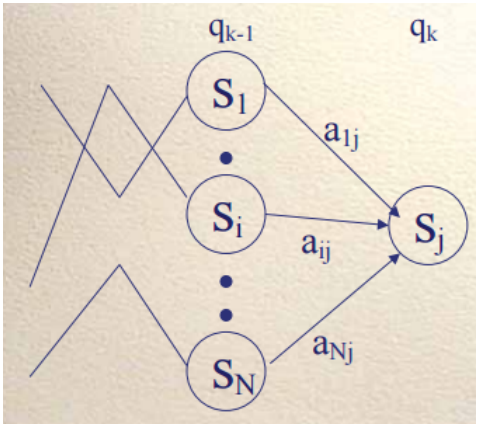
Given a HMM $M = (A, B, \pi)$ and a sequence of observations $o_1 ... o_K$ the decoding problem finds the most likely sequence of hidden states producing that sequence of observations. With formulas, we want to find the sequence of states $Q = q_1 ... q_k$ such that $P(Q|o_1 ... o_k)$ is maximized. To solve this problem we can't use a brute-force algorithm since considering all possible sequences of states would take an exponential time. The solution is to use the brute-force approach is **Viterbi algorithm**.

10.7.1 *The algorithm*

The Viterbi algorithm is based on a simple idea: if the best path ending in $q_k = S_j$ goes through $q_{k-1} = S_i$, then it should "coincide" with the best path ending in $q_{k-1} = S_i$: in other words, every sub-path of the best path is a best path itself (with a different ending state). The algorithm is based on the definition of a variable, $\delta_k(i)$, the **maximum probability of producing** $o_1 ... o_k$ when moving along any state sequence $q_1 ... q_{k-1}$ and ending up in $q_k = S_i$.

$$\delta_k(i) = \max P(q_1 ... q_{k-1}, q_k = s_i, o_1 ... o_k)$$

...where max is taken over all possible paths $q_1 ... q_{k-1}$.



The inductive definition is almost identical to the *Forward recursion* algorithm; the only difference is that instead of the sum in the inductive step we have the **max operator**:

- **Initialization**. First, the maximal probability of having just one observation and being in state S_i is $\pi_i b_i(o_1)$; This is done by any i.
- **Inductive step**. The maximal probability of the best path of states to obtain $o_1 ... o_k$ is inductively computed considering the prefix of size k-1, the probability transition from i to j and the probability of being in state S_j when o_k is observed:

  $$\delta_k(j) = maxP(q_1 ... q_{k-1}, q_k = s_j, o_1 o_2 ... o_k) =$$
  $$max_i[a_{ij} b_j(o_k) maxP(q_1 ... qk-1 = s_i, o_1 ... o_{k-1})] =$$
  $$max_i[a_{ij} b_j(o_k \delta_{k-1}(i)]$$

  In the **termination phase** the best path is chosen: $max_i[\delta_K(i)$.

Additional backtracking is also used to recover the best path.

10.8 LEARNING PROBLEM

The learning problem is defined as follows: given a "training" observation sequences $O = o_1 ... o_k$ and the general structure of HMM (number of hidden and visible states), how can we adjust the HMM parameters $M = (A, B, \pi)$ to best fit the training data? We assume the training sequences are **independent**: this way we apply apply the algorithm separately to each sequence and then we "accumulate" the resulting probabilities.

10.8.1 *The algorithm*

To solve the problem we use an iterative **expectation-maximization algorithm**, the **Baum-Welch algorithm**, which combines both the *Forward* and the *Backward* probabilities seen before.

- **Expectation step**. In this step we must compute two values:
  - First $\xi_k i, j$, the probability of being in state S_i at time k and being in state S_j at time $k+1$ given the observation sequence $o_1 ... o_K$:

$$\xi_k i, j = P(q_k = S_i, q_{k+1} = S_j | o_1 ... o_K)$$

  - Next, we compute $\gamma_k(i)$, the probability of being in state S_i at time k given the observation sequence $o_1 ... o_K$:

$$\gamma_k(i) = P(q_k = S_i | o_1 ... o_K)$$

- **Maximization step**. Once we have the previous values we can conpute the three parameters we need:
  - $a_{ij} = \frac{\sum_k \xi_k(i,j)}{\sum_k \gamma_k(i)}$, or with words the ratio of the Expected number of transitions from state S_i to state S_j and the expected number of transitions out of state S_i;
  - $b_i(v_m) = \frac{\sum_{k, o_k = v_m} \gamma_j(i)}{\sum_k \gamma_k(i)}$ or, with words, the ratio of the expected number of times observation v_m occurs in state S_i and the expected number of times in state S_i;
  - $\pi_i = \gamma_1(i)$ or the expected frequency in state S_i at time $k = 1$.

Iteratively these parameters are recomputed until they stabilize (they're not that different from the values of the previous iteration).

11

GENOME SEQUENCING, PART 1

Genome sequencing is a laboratory process that **determines the sequence of the bases composing the genome**[1]. Genome sequencing is not interested in learning the meaning behind the genome: the only goal is to discover the sequence of symbols.

11.1 INTRODUCTION: COMMON MISCONCEPTIONS

The genome is the genetic material of an organism and is usually organized in different chromosomes within each cell. We usually assume that **genome complexity, the number of genes and the number of chromosomes** are directly proportional to the complexity of the corresponding organism while, on the contrary, many counterexamples show that these are only **misconceptions**:

- **C-value paradox**: there's no correlation between the complexity of an organism and its genome. For example the human genome is about 200 times smaller than the *amoeba dubia* genome.
- **K-value paradox**: there's no correlation between the number of chromosomes of an organism and its biological complexity. For example humans have 23 couples while some plants have 1260 chromosomes.
- **N-value paradox**: there's no correlation between the number of genes[2] of an organism and its biological complexity: a human being has around 30k genes while rice has 50k.

11.2 INTRODUCTION: HISTORY AND MAIN TECHNIQUES USED BY GENOME SEQUENCING

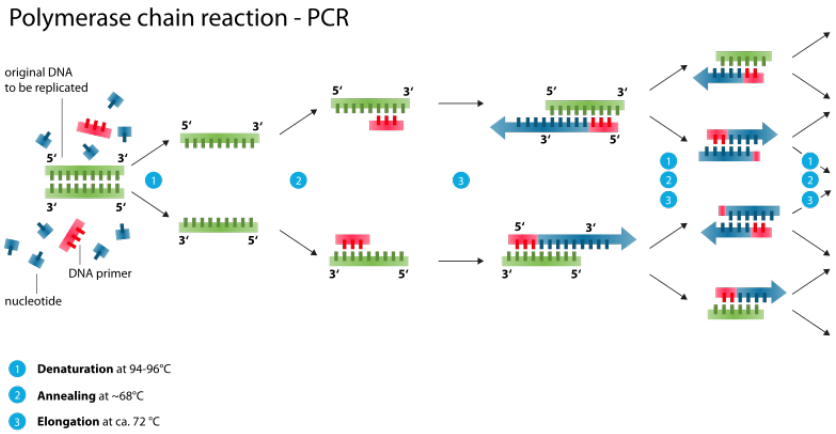
The first sequencing project was completed in 1965 by Holley, after 9 years of work: despite focusing on a very small fragment of RNA (only 77 bases) it was done completely by hand. In the following

1 The genome is organized in different chromosomes within each cell of an organism.
2 Remember that genes are portions of DNA encoding proteins.

years new technologies were developed to **sequence reads**, smaller fragments of DNA. Let's see the most important ones:

11.2.1 *Polymerase Chain Reaction (PCR)*

Developed in 1986, PCR is a technique used to **amplify a single copy of a fragment of DNA**, generating millions of copies of a particular DNA sequence. The process alternates **cooling and heating cycles**:



11.2.2 *Sanger sequencing technique (small fragments)*

Sanger sequencing technique, also known as the ***chain-termination method***, was developed in 1977[3] with the goal of **sequencing small fragments of DNA or RNA**. The method is completely based on chemistry: a computer is used only at the very end of the following process:

1. A single-stranded DNA fragment called *template DNA*is immersed into a solution containing a mixture of **replication-stopping bases**; after the reaction, a population of copies of the template is produced, **each truncated at a a different length**;

2. The population in inserted into a capillary tube gel: using electrophoresis the largest fragments move the slowest and the shortest fragments move the fastest along the tube; a laser positioned

3 Interestingly, in the same year Maxam and Gilbert developed another sequencing technique by means of chemical degradation.

at the bottom of the tube analyzes each falling sequence and determines the bases of the original DNA strand.

11.2.3 *Shotgun sequencing technique (whole genome)*

Still developed by Sanger, the *shotgun technique* is used to sequence a whole genome and **makes no a priori assumption** on it before the sequencing phase. It consists of **4 phases**:

1. The genome is **amplified and broken up into smaller fragments**, which are generated as **randomly** as possible;
2. Each fragment is sequenced separately, using Sanger's previous technique;
3. A fragment-assembly software reassembles the fragments.

Using this technique, in 1995 researchers were able to sequence the whole genome of a bacterium (*Haemophylyus influenzae* with 1.8 millions of bases); in 1996 the whole genome of yeast, with 13 million bases was also sequenced. In 1998 the sequence of the first multicellular organism was released: since then the list of complete genomes has been rapidly growing (the *Genome OnLine Database* reports more than 50k projects on different organisms, with half of them completely sequenced).

11.3 INTRODUCTION: THE HUMAN GENOME SEQUENCING PROJECT

A fundamental step for the development of new, modern tools for sequencing was the **Human Genome sequencing project**. Started in 1990 as a public research funded by the American Department of Energy and the National Institute of Health, the Human Genome sequencing project soon it became an **international effort** involving 16 institutions and various research laboratories in the world. The initial goals of the project were simple:

- sequence the whole human genome (3.3 billion bases) using hierarchical sequencing bac-by-bac and identify all its genes (more than 30k);
- produce databases and tools to support the research;
- deal with all the connected ethical, legal and social problems.

Scheduled to end in 2005, after eight years the scenario changed completely when Celera Genomics, a private competitor with a new technology (*whole genome shotgun*), came into play. The competition accelerated the research: in 2001 both institutes published a first draft of the whole genome and, in 2003, both published the final sequencing. Officially , the Human Genome sequencing project ended in 2003; still, since the results were different, the **discussion continues** on the validity of the techniques employed and on the quality of the obtained sequences.

- **Problems in genome sequencing: complexity, length and repeated regions**

Genome sequencing is not an easy process: a genome is so **intrinsically complex** that errors are often common – a fact that might make us question the reliability of the resulting sequences. Furthermore, not only the genome can be **extremely long** but it may also contain **repeated regions** – identical subsequences repeated in many places within the genome. This is typical of eukaryotes: for example, 50% of the the *Homo sapiens* genome is composed by repeated regions. Repetitions are especially problematic if we think about the Sanger shotgun technique seen before: with so many repeated areas, how can we correctly reassemble the various fragments of the genome? And what happens if repeated regions are longer or shorter, or if they're repeated in different chromosomes? As we can see genome sequencing is extremely complex, especially from a computational point of view.

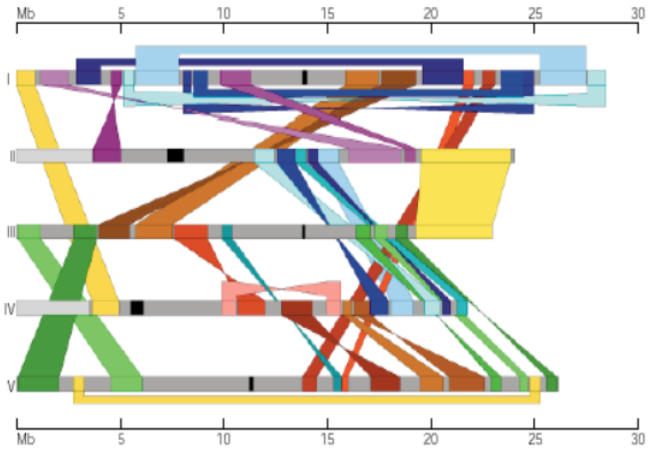


Figure 47: Example of repeated regions between different chromosomes.

11.4 WHOLE GENOME SEQUENCING: SHOTGUN TECHNIQUE

Many approaches have been developed to sequence the whole genome of an organisms. In the following pages we will focus on two classics, the **Shotgun sequencing** technique and its main improvement, **Double-barreled Shotgun sequencing**, both widely used until the development of Pyrosequencing. We will also mention Hierarchical sequencing, also known as *clone-by-clone* or *bac-by-bac*, used by the Human Genome project, the Sequence-tagged connector approach, the Whole-genome Shotgun sequencing used by Celera Genomics and Sequencing by hybridization (SBH). Notice that, no matter the chosen technique, every sequencing approach always has the **same 3-phase structure**:

1. DNA sequencing: the strand is fragmented (biochemical process);
2. Fragments are reassembled (software-based process);
3. The previous results are cleaned up and refined (still mostly done by hand).

Let's consider each phase with the Shotgun technique in mind.

11.5 SHOTGUN, PHASE 1: DNA SEQUENCING

The DNA sequencing phase can be imagined as a black box: given a single DNA fragment it produces a **population of copies of *reads*** randomly covering the original fragment.

The Shotgun technique implements the DNA sequencing phase applying the following steps are applied:

1. Partition the genome into fragments and, for each of them, produce a large number of copies;
2. Randomly fracture each copy to obtain billions of smaller fragments of different sizes;

3. Filter the fragments removing those that are too long or too short: we obtain a **library of fragments**.

4. Each remaining fragment is inserted into the DNA of genetically-engineered viruses called **vectors**; each vector infects a bacterium that multiplies itself rapidly, producing millions of clones each with an exact copy of the insert.

5. A biochemical process reads the **prefix of the insert** starting from a primer. Only a few hundred bases are read, depending on the vector: after that precision degrades.

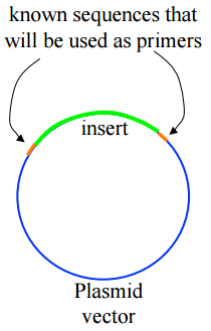
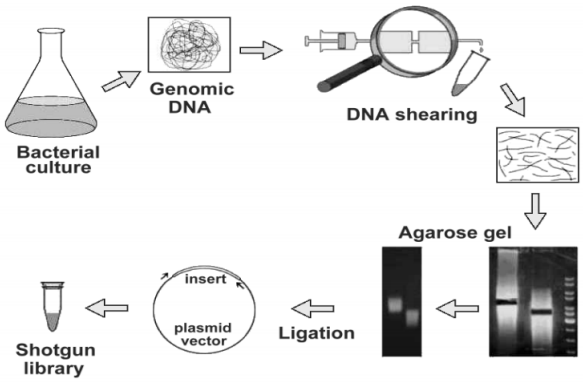


Figure 48: A *plasmid* with an insert.

As a result, a collection of **reads** covering each fragment of the original genome is obtained.



Unfortunately, the DNA sequencing phase can produce **biased reads** – reads that are not randomly sampled but biased to come from the initial DNA fragment. This can happen for a variety of reasons:

- Biased fracturing: the initial fracturing of each fragment is not random enough;

- Failed insertions of fragments into vectors: if a fragment is not inserted or only partially inserted into a vector then it will not be amplified; as a result, the information carried by that fragment will not be as represented.
- Failed bacterium cloning: if the combination of the vector and the insert produces a toxic reaction the bacterium might not be capable of cloning itself.

Errors may also happen when **reads are read from the bacteria DNA**: for example, the reading phase might start before the primer (so a bit of the DNA vector must be removed) or might end after the insert (if an insert is particularly short; technicians might need to trim the vector sequence from the end of the read). Finally, sequencing the individual fragments may also produce **reads with mutations**: we can have one or more bases wrongly inserted in a fragment (insertion errors), one or more bases wrongly deleted from a fragment (deletion errors), bases substituted with another bases (substitution errors), etc.

11.6 SHOTGUN, PHASE 2: FRAGMENT ASSEMBLY

In the second phase reads must be assembled into a contig, the reconstructed sequence corresponding to the original DNA strand.

A necessary condition to obtain valid contigs is to have a **good coverage of the original DNA fragment**. Coverage is measured as the ratio between the number of sequenced bases and the size of the genome: the more sequenced bases we have, the more the genome is covered.

$$c = \frac{\text{Number of sequenced bases}}{\text{genome size}}$$

The number of bases can be easily computed if we remember that we know both the amount of reads produced by the DNA sequencing phase and the average length of each read:

$$\text{Num Bases} = \text{Average length of read} * \text{Num of reads}$$

It is also possible to compute the probability P_0 that a base has not been sequenced with respect to the coverage (assuming a Poisson distribution):

$$P_0 = e^{-\text{coverage}}$$

The **percentage of effective coverage** is given instead by:

$$\%Cov = 100 * (1 - e^{-c})$$

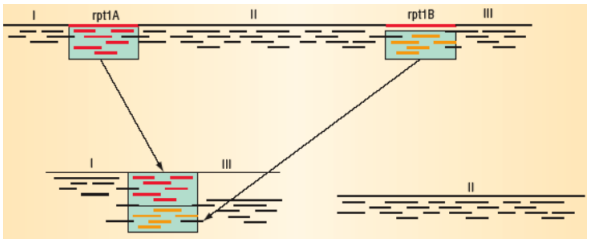
With an example, if we have a genome with 4 million bases sequenced with 4mb bases then the corresponding coverage is 1x; the percentage of effective coverage is 63.2%. Notice that **full coverage is never guaranteed**, even with coverage 10x: errors and biases in the sequencing phase make it impossible.

Coverage	Percentage effective coverage
1	63,2%
2	86,5%
3	95,0%
4	98,2%
5	99,3%
6	99,8%

Fragment-assembly is not an easy process for many reasons:

- **Coverage is always incomplete**: some parts of the original DNA sequence covered exactly c times while others are covered less than c times.
- **There might be *gaps***, whole regions of the original DNA strand that are not covered. This leads to fragmentation or incomplete solutions.
- **Errors** can be made when reading the insert within each bacterium, especially if we're reading far from the primer, so a fragment-assembly algorithm should model reads with mutations.
- **Errors can be caused by orientation**: the DNA is a double-strand structure but only one end of the insert is sequenced. The right sequence to use might be either the read or its reversed complement.

- **Repeated sequences** can cause assembly errors: the reads of two repeated regions may be associated to the same area of the original DNA fragment when the instead represent two different areas.



There are many fragment-assembly algorithms, some more practical and some more heuristic:

11.6.1 *Shortest superstring*

The first technique used to assemble reads is based on the *shortest superstring problem*: given a set of reads P, the target DNA is the shortest superstring of P – the shortest string containing all reads as substrings. To obtain it we simply **concatenate all reads in any order we like**. Repeated reads and symbols are collapsed together in the shortest possible way: `aab` and `bbc` is collapsed in `aabbc`.
As an example, if we have the following reads...

```
1. AAT
2. CTGG
3. AGC
4. TCT
```

...then the shortest superstring is `AATCTGGAGC`. Despite being very simple this technique has **two serious disadvantages**:

- Finding the shortest superstring is a **NP-hard problem**; heuristics are used to approximate it but they're still polynomials.
- It **does not work with mutated reads** (it is an*exact* technique in this sense), does not model fragment orientation and **fails to notice repeated regions** since they're collapsed.

For these reasons the SSP algorithm is considered of **theoretical interest only**.

11.6.2 *Overlap-Layout-Consensus (OLC)*

OLC is a heuristic, greedy approach widely used to assemble reads. It works in 3 steps: find the overlaps among reads, use this information to produce several possible layouts of the fragment, decide the consensus to converge on a single layout.

- **Step 1:** ***Overlap detection***

In the first step OLC finds the overlaps between every ordered pair of reads, also considering possible different orientations. Of course, overlaps **depend on the "error assumption"**:

[Exact reads]. If reads don't have mutations overlaps are detected using an exact approach, the **all-pairs suffix-prefix method**: for every ordered pair of strings (S_i, S_j) find the longest suffix of S_i which is a prefix of S_j. An optimal algorithm to find all the longest suffix-prefix matches is based on the construction of the **generalized suffix tree** for all k reads:

- Build the generalized suffix tree for all k reads, each with a different terminal symbol;
- Maintain a list for each internal node v: if the label of v is a suffix of read i then add i to the list.
- Traverse the tree depth-first maintaining a stack for each read: once the node v is reached push i into its stack if $i \in L(v)$. Once a leaf j is reached check the top element of each stack: if the stack is empty then there's no overlap with j. When the depth-first traversal backs up past v, pop the top of each stack whose index is in $L(v)$.

The cost of this algorithm is $O(m + k^2)$, where k^2 is number of ordered pairs of reads; as a result, the more reads we have the higher the computational cost; furthermore, **overlaps are rarely between *exact* reads**.

[Approximated reads]. If reads can have mutations then the previous all-pairs suffix-prefix problem is relaxed: for every ordered pair of strings (S_i, S_j) find a suffix of S_i and a prefix of S_j whose **similarity is maximum**. Now, suppose the similarity is defined using an objective function F: matches produce positive contributions while

mismatches and gaps produce negative contributions. Then, finding an overlap means performing a semi-global alignment on the ordered pairs of reads. We can use a **variation of the Needleman-Wunsch algorithm**:

- **Initialization**. Given read x of length n and read y of length m, $F[i,0] = 0$ and $F[0,j] = 0$ since initial and final gaps are not penalized;
- **Inductive step**. Given an appropriate score matrix and gap penalty, similarity is maximized as follows:

$$F[i,j] = max[F(i-1,j-1) + s(x_i, y_j), F[i-1,j] + d, F[i,j-1] + d]$$

...where d is the gap penalty while $s(x_i, y_j)$ is the score of the two bases. $F[i,j]$ is maximized for all ordered pair of reads.

The **maximal score is on the last row or column**: by **going backward** until one element of the first row or column is reached we obtain the alignment. As an example consider this similarity score matrix plus the gap penalty $d = -2$. By using the given recursive equation the algorithm is applied to the strings `GCCATCT` and `TCTAAGC`.

	A	C	G	T
A	1	-1	-1	-1
C	-1	1	-1	-1
G	-1	-1	1	-1
T	-1	-1	-1	1

The resulting table is:

	ε	G	C	C	A	T	C	T
ε	0	0	0	0	0	0	0	0
T	0	-1	-1	-1	-1	1	-1	1
C	0	-1	0	0	-2	-1	2	0
T	0	-1	-2	-1	-1	-1	0	3
A	0	-1	-2	-3	0	-2	-2	1
A	0	-1	-2	-3	-2	-1	-3	-1
G	0	1	-1	-3	-4	-3	-2	-3
C	0	-1	2	0	-2	-4	-2	-3

The highest values are in the last rows and columns: by going backwards we obtain the alignments. As for the computational cost, it still depends on the the size of each read (n) and the number of reads (R): $O(R^2n^2)$.

[Heuristics for overlaps with errors: LCS]. With a very high number of reads **heuristic techniques are preferred**. As a result the suffix-prefix match is further relaxed: or every ordered pair of strings (S_i, S_j) **find the longest common substring** between the two. In practice it is enough to build the **generalized suffix tree of each ordered pair of reads** and find the deepest node whose children have different terminator symbols: its path-label is the longest common string. The complexity is linear with respect to the sum of the lengths of the two reads. Notice that this approach has a downside: since it's only an heuristic, it does not distinguish between real overlaps and fake overlaps between repeated regions. For example, suppose we have the following reads (the longest common substring is between $*$):

```
S1 = AAT*AGTCCA*GCA$1
S2 = AC*AGTCCA*TGT$2 <- overlap may be fake!
```

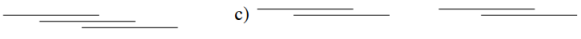
[Further heuristics]. Other heuristics use tools such as **BLAST** to find a good *local* alignment among all possible pairs of reads: the higher the score, the greater the probability that the two corresponding reads are overlapping. Notice that the **semi-global alignment criterion is further relaxed** in this way. All-in-all, these techniques greatly reduce the overall computation time needed for the sequence-assembly problem.

- **Step 2:** ***fragment layout***.

In the second step OLC tries to reconstruct the layout of the target DNA sequence starting from the overlapping reads. The most common approach is to use a **greedy algorithm**:

1. Get the pair of reads with the highest overlapping score and merge them together to **obtain the first contig** (a set of contiguous reads).

2. Select the pair of reads with the second highest overlapping score and merge them together: either we obtain a new distinct contig with two reads or we elongate the previous one, now containing three reads.



3. Iterate the previous steps.

Notice that since we usually don't have exact overlaps, gaps are allowed in the suffix-prefix matches: as successive reads are added to the contig, additional gaps may have to be inserted to be consistent with the already existing gaps. As a result **hundreds or thousands of different contigs are usually produced**. As for the problems and limitations of the greedy approach, we have:

- It's difficult to tell real overlaps between reads from fake overlaps between repeated regions: in the first case the difference between reads is due to sequencing errors, in the second is due to mutations;
- The greedy algorithm can get stuck to local maxima, never finding the "highest" overlaps;

Furthermore, the greedy assemblers have some problems: they can get stuck at local maxima and the overlaps induced by repeated regions may score higher than real overlaps: coping mechanisms are needed to avoid incorporating false-positive overlaps into contigs, producing contigs with unrelated sequences (***chimeras***).

- **Step 3:** ***consensus phase***.

Once we've produced one or more contigs we must determine the *consensus* on each of them: in other words, we need to select the actual base in each position taking into account the data coming from the various reads. Various approaches can be used:

- **Profile building method**: build a profile for each position (the frequency of each character for that position) and let the user decide how to use that information;
- **Plurality approach**: maintain the plurality of bases on each position, provided that each is above some preset threshold.

11.6.3 *Example*

Suppose we want to construct the contig(s) starting from the following reads:

```
AGCATCT
TACTGGA
```

```
GCCATCT
ACTTAG
TCTTTA
TCTAAGC
```

Suppose reads are already oriented, have no errors and that the coverage of the target DNA is 1. To find the contig(s) we apply the greedy approach starting from the couple of reads with the largest overlap: over to find the exact overlap between them. To do so, we notice that the largest overlap, 3:

```
   AGCATCT             [1]
TCTAAGC                [6]
```

There's another size-3 overlap between 3 and 1:

```
      AGCATCT          [1]
   TCTAAGC             [6]
GCCATCT                [3]
```

Another size-3 overlap between 6th and 5th:

```
      AGCATCT          [1]
   TCTAAGC             [6]
GCCATCT                [3]
           TCTTTA      [5]
%
```

Size 3 overlaps are over, so we look for size-2 overlaps. The first is between the 5th and the 2nd reads:

```
      AGCATCT          [1]
   TCTAAGC             [6]
GCCATCT                [3]
           TCTTTA      [5]
               TACTGCA [2]
```

Next, we have an overlap between the 2nd and the 4th:

```
      AGCATCT          [1]
   TCTAAGC             [6]
GCCATCT                [3]
           TCTTTA      [5]
               TACTGCA [2]
                     ACTTAG [4]
```

We've used every read: we obtain one of the possible results, a single contig with no errors, gaps or mismatches.

```
GCCATCTAAGCATCTTTACTGCACTTAG
```

11.7 SHOTGUN, PHASE 3: FINISHING

Once we've obtained one or (more probably) more contigs we need to **close the gaps between them** as much as possible to obtain the target DNA. The technique used is called **scaffolding**: different contigs are ordered and oriented into larger structures (scaffolds). Notice that gaps may still be present: if they're *contig gaps* they can be easily reconstructed using PCR; instead if they're scaffold gaps (between scaffolds) a large amount of manual labour and complex laboratory techniques are usually employed to refine the target DNA. Note also that in the finishing phase tools for visualization of the sequences are extremely useful.

11.8 OVERLAP GRAPHS

OLC is generally structured visualized by using **overlap graphs** where nodes represent reads and edges represent overlaps between reads. A **path between nodes represents a contig**. For example, consider the previous set of reads:

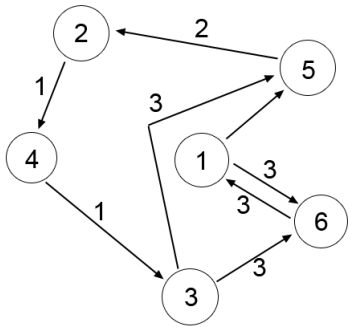
```
AGCATCT
TACTGGA
GCCATCT
ACTTAG
TCTTTA
TCTAAGC
```

With a graph we could represent the overlaps previously seen (1-6, 3-1, 6-5, 5-2, 2-4), annotating on arcs how much each couple overlaps (3,3,3,2,1 respectively). Notice that arcs are directed to show which is the suffix and which is the prefix involved. The result would be:

On it, using the greedy approach, we obtain these possible paths/-contigs:

```
3 -> 6 -> 1 -> 5 -> 2 -> 4 (1 contig)
```

```
4 -> 3 -> 6 -> 1 -> 5 -> 2 (1 contig)
```




```
1 -> 6, 4 -> 3 -> 5 -> 2 (2 contigs)
```

Once multiple contigs are found, a multiple sequence alignment (MSA) produced using heuristics determines the layout and then the consensus sequence. Note that the previous example was very simple since we don't have substring overlaps (only suffix-prefix). There are no annotations to guide the greedy technique: the general idea to find a contig is to **find the largest path without cycles**. As for **annotations**, the graph can contain further information: orientation of reads, lengths of the reads, lengths of the overlaps like done here, type of overlap (prefix-suffix, substring), number of reads for each position of the original DNA strand, etc. Annotations can also be used to clean the graph, eliminating the portions with little coverage (not many reads existing on that portion).

12

GENOME SEQUENCING, PART 2

In this second part on genome sequencing we will focus on the Double-barreled Shotgun technique; we will see the strategies used for sequencing the human genome and introduce new alternatives for sequencing: hybridization (SBH), the De Bruijin approach, with techniques and problems.

12.1 DOUBLE-BARRELED SHOTGUN SEQUENCING

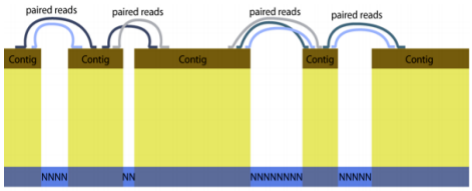
Introduced in the context of the Human Genome sequencing project, the Double-Barreled Shotgun sequencing technique **improved they way reads were produced** by the previous Shotgun technique:

- Reads are produced by **sequencing both ends of the fragment** inserted into the bacterium: this way the amount of data we obtain from a single bacterium is increased.
- We don't have situations where reads are a mixture of the insert and the genome of the bacterium: the average length of the insert is twice more than twice the length of each read.

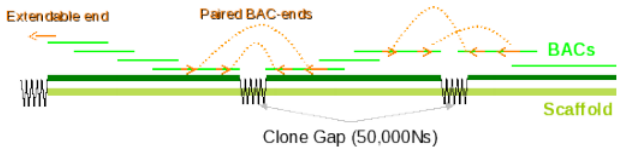
As we can see this technique produces **mates**, pairs of reads of which we know both **the orientation and the distance** at which they're positioned. This information is **exploited** to **reconstruct the order of contigs inside scaffolds**: if a read belongs to a contig and its mate belongs to another then we assume that the two contigs are adjacent; various mate pairs in two possibly adjacent contigs further confirm the order between contig.

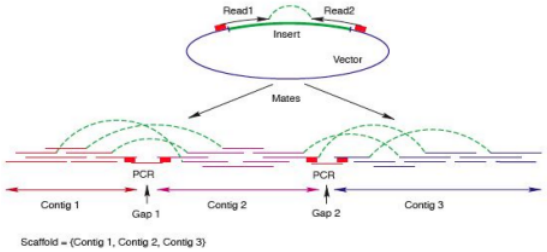
Contig gaps can also be filled using using the mate pair information.

Finally, we can also **identify repeats** since the probability that both reads of an insert fall into the same repeat is low.



Five contigs are joined into a single scaffold thanks to paired read information





Once contigs are obtained, the scaffold can be obtained using a **graph where nodes are contigs** and edges are inserted when mate pairs bridge the gap. Then we simply apply the previous **greedy algorithm**.

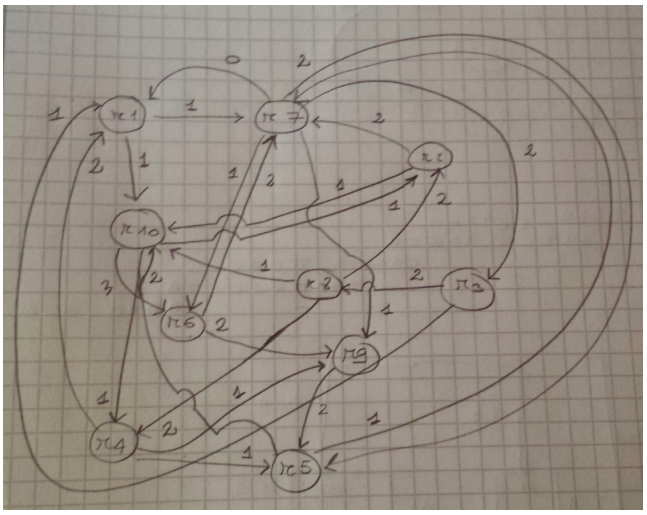
12.1.1 *Example*

Consider the following set of mate pairs. Assume that the orientation is already perfect, coverage is 1x, there are no overlaps between the reads of a mate pair and the average length of an insert is $10,8$bp.

1. (r1:TAATC, r2:GCAC)
2. (r3:TAATT, r4:GCCTA)
3. (r5:AATCC, r6:AAGCA)
4. (r7:CAGGTA, r8:TTCGC)
5. (r9:ATAA, r10:CCTAAG)

To reconstruct the target DNA using the Double-barreled Sshotgun approach we first need to represent the overlaps between reads of different mates as an **oriented graph**. The resulting graph is showed below; it can be read as "read r2 overlaps read r10 of size 1 and viceversa".

To reconstruct the contig(s) we apply the greedy approach, looking for the pair of reads with the strongest overlap. Since r10 (mate with



r9) and r6 (mate with r5) have a size-3 overlap we start from there: assuming between each read there's a gap of unknown length (either 1 or 2), this corresponds to the following overlaps:

```
r9-r10:   ATAA_CCTAAG
r5-r6:      AATCC_AAGCA
```

Size-3 overlaps are over. r6 has a size-2 overlap with r7, which is the mate of r8:

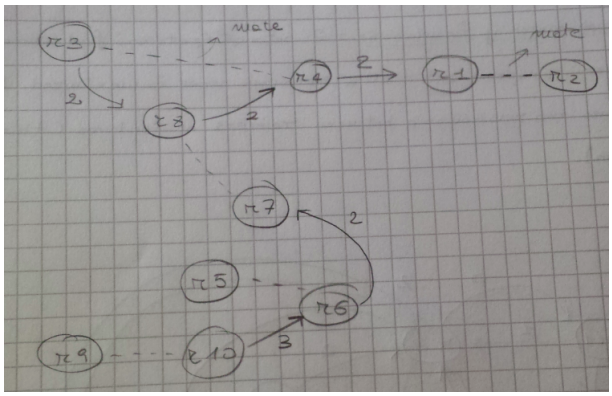
```
r9-r10:   ATAA_CCTAAG
r5-r6:      AATCC_AAGCA
r7-r8:                CAGGTA_TTCGC
```

But r8 overlaps with r4, which is the mate of r3. Notice that r3 also overlaps with r8 (size 2):

```
r9-r10:   ATAA_CCTAAG
r5-r6:      AATCC_AAGCA
r7-r8:                CAGGTA_TTCGC
r3-r4:                    TAATT_GCCTA
```

Finally, r4 overlaps with r1, which is the mate of r2; r2 does not overlap with anything else.

```
r9-r10:   ATAA_CCTAAG
r5-r6:      AATCC_AAGCA
r7-r8:                CAGGTA_TTCGC
r3-r4:                    TAATT_GCCTA
r1-r2:                            TAATC_GCAC
```



The overall result is:

The contig we obtain is $\mathtt{ATAATCCTAAGCAGGTAATTCGCTAATC_GCAC}$. Notice the gap of unknown size, caused by the **absence of support** for that piece of the target DNA. Everything else is reconstructed without mismatches.

12.2 INTRODUCTION: OTHER TECHNIQUES

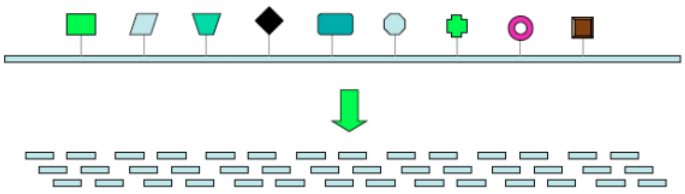
We will now focus on three different techniques used to sequence the human genome:

- **Hierarchical sequencing**, used by the public consortium and also called *clone-by-clone* or *BAC-by-BAC* sequencing);
- **Sequence-tagged connector approach**, a variant of the hierarchical sequencing;
- **Whole-genome Shotgun sequencing**, used by Celera Genomics.

12.3 CONSORTIUM: HIERARCHICAL SEQUENCING

Hierarchical sequencing is based on three steps:

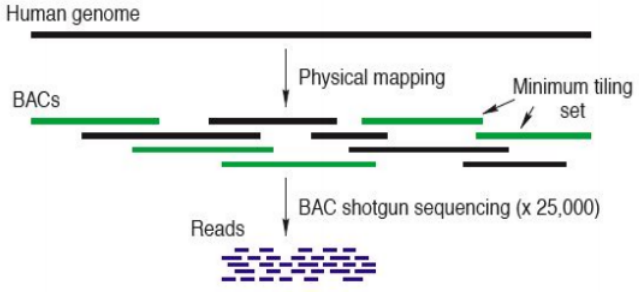
1. Randomly fragment the genome into sequences of $50-300$ kilobases and insert these fragments into BACs (Bacterial Artificial Chromosomes), producing a **BAC library**.
2. Build a **physical map of the genome**, containing a rough description of what we already know of the genome (markers, genes, patterns, etc). Since this map is used to order the different fragments it must be **accurate** (otherwise chimeras can be produced).



3. Produce a minimal tiling path between BACs, that is, a sequence of BACs with minimal overlapping and covering the whole genome. Each BAC in this chain is called **tiling clone**.

4. Sequence each tiling clone using the standard Shotgun technique;

5. Finishing phase to polish the overall result.

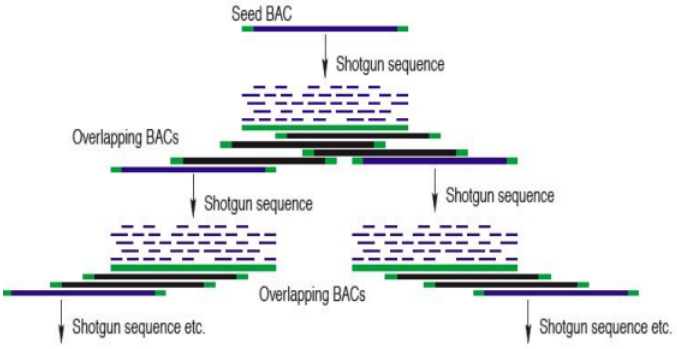


The main steps of the *Hierarchical technique* are the **sequencing phase**, which is automated, and the production of the physical map, which instead needs a lot of human labour. In fact, physical map are very difficult to produce; once they're done, however, they allow a **scalable approach on the genome** and make the finishing phase easier.

12.4 SEQUENCE-TAGGED CONNECTOR APPROACH

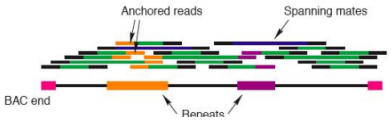
A variant of the *Hierarchical approach*, Sequence-tagged Connector **does not build the expensive physical map**. Instead, once the BAC library is built, a set of **seed BACs** are randomly selected from the library and sequenced using the Shotgun technique.

Then, other BACs overlapping the seeds are used to **extend the seeds on both directions**; the minimal overlapping BACs are therefore Shotgun sequenced.



12.5 CELERA GENOMICS: WHOLE-GENOME SHOTGUN SEQUENCING

Introduced by Celera Genomics to sequence the whole human genome, this technique applies the Double-barreled Shotgun sequencing with **support vectors of different sizes** (BACs, 2Kbp plasmid, 10Kbp plasmid). The different sizes of the various inserts and the mate pairs information are used to **resolve the repeated regions problem**: reads belonging to a repeated region have the repeat's colour and their mate is not in a repeat (**ancored mates**).



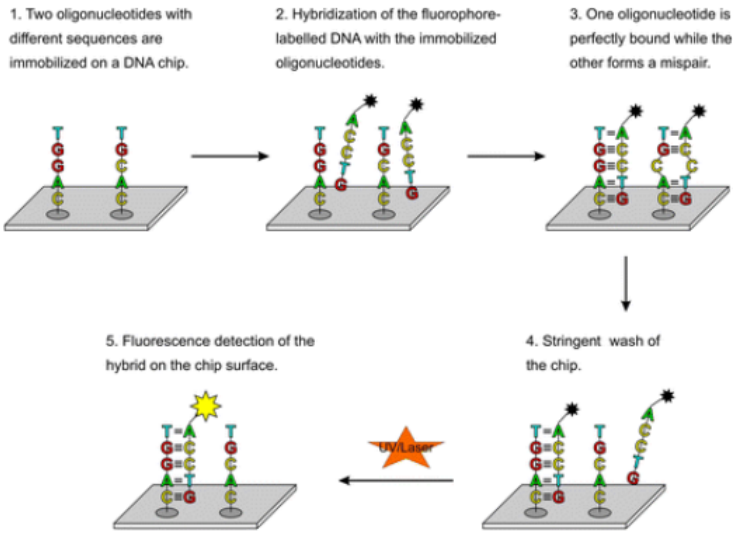
12.6 SEQUENCING BY HYBRIDIZATION (SBH)

Proposed in 1988 but actively used since 2005, SBH is a powerful sequencing method that **does not involve electrophoresis**; instead, it is based on the production of **DNA chips**. A DNA chip is a **matrix containing all possible DNA probes of a specific length**, each in a known location: for example, a DNA chip of size k contains all possible combinations of bases of size k, each with a specific position in the matrix. The approach works as follows:

1. **Produce a DNA chip** of a certain length, each with a known location;
2. Apply to the DNA chip a solution containing the strand of DNA we want to sequence: the DNA fragment hybridises with the complementary probes of the chip.

3. using a spectroscope, detect the probes that binded with the DNA strand obtaining a set of S **k-mers** (the **spectrum** of the original sequence). Notice that k-mers overlap between each other.

4. Finally, the original DNA is reconstructed using the spectrum.



Of course, the more k is large the better (since we obtain more probes); in the beginning k was 8, nowadays is a much larger value. Notice that this approach can have **errors**: first of all a k-mer might bind stronger than others (**Hybridization errors**); furthermore, we can't tell how many times a k-mer is occurred (we notice it's occurred but we can't say how many times).

12.6.1 *Example on SBH*

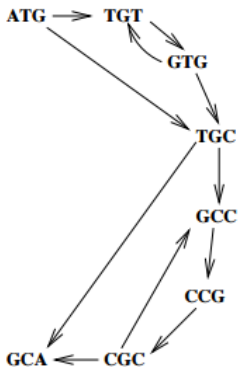
Suppose that the size of the probe is $k = 3$. The overall set of probes is:

```
AAA ACA AGA ATA
AAC ACC AGC ATC
AAG ACG AGG ATG
AAT ACT AGT ATT
CAA CCA CGA CTA
CAC CCC CGC CTC
CAG CCG CGG CTG
CAT CCT CGT CTT
GAA GCA GGA GTA
GAC GCC GGC GTC
GAG GCG GGG GTG
GAT GCT GGT GTT
TAA TCA TGA TTA
TAC TCC TGC TTC
TAG TCG TCG TTG
TAT TCT TGT TTT
```

The unknown target sequence is `TACACGGCGT`, which is the complement of `ATGTGCCGCA`. The resulting spectrum is $S = \{\text{ATG}, \text{CGC}, \text{CCG}, \text{GCA}, \text{GCC}, \text{GTG}, \text{TGC}, \text{TGT}\}$.

12.6.2 *Graphs of k-mers and reconstruction of the target sequence*

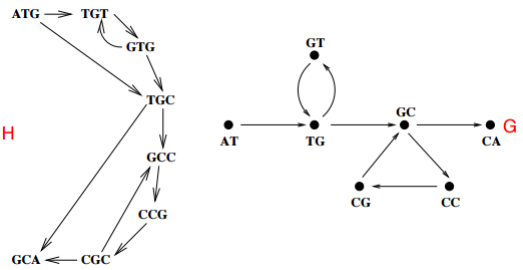
Once we have the spectrum it is possible to build the **graph of its k-mers**: nodes are k-mers and edges are represent a **suffix-prefix overlap of $k-1$ symbols**. For example, given the previous spectrum $S = \{ATG, CGC, CCG, GCA, GCC, GTG, TGC, TGT\}$, the corresponding graph is:



To infer the target DNA from this graph it is enough to **find the Hamiltonian path**, the **longest path traversing every node exactly once**. With small examples this is easy to do (in the previous example the unique Hamiltonian path produces the sequence `ATGTGCCGCA`); still, in general the problem of finding an Hamiltonian path is **NP-complete**: to solve this problem we need to switch to the corresponding dual problem. This is done as follows:

- **Transform the spectrum graph H into a De Bruijn graph** G. This is done by:
  - **Nodes**: each node/k-mer produces two nodes/$(k-1)$-mers, a prefix and a suffix of the original k-mer. For example, the 3-mer AAC produces AA and AC.
  - **Edges**: an edge is placed from u to v if there's a k-mer in the original graph H with u as a prefix and v as a suffix. For example, there's an edge between AA and AC because h contained AAC.
- Find the **Eulerian path of** G – the path that visits each **edge** only once.

This way the previous problem is solved in linear time. Getting back to our previous example, the "transformed" graph G is the following:



The Eulerian path of graph G produces the sequence ATGTGCCGCA. Notice that this solution **is not without errors**: the same graph, in fact, can have **many different Eulerian path** each corresponding to a different target DNA. To solve this problem graph theory has introduced **further conditions** on how Eulerian paths are found (conditions to make the Eulerian path unique and/or efficient algorithms to find the "best" Eulerian path in a graph).

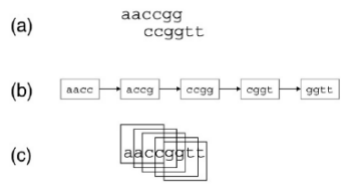
Figure 49: Three different target sequences: ACTAC, TACTA and CTACT

12.7 DE BRUIJN APPROACH FOR FRAGMENT-ASSEMBLY

The term "De Bruijn approach" refers to the fact that De Bruijn graphs can also be **used in the fragment-assembly phase of the Shotgun sequencing**. Suppose we already have a library of reads to assemble:

- **Decompose** each read (length n) into a set of $n - k + 1$ k-mers, storing for each of them the read it belongs to and its position within the read.
- Build the De Bruijn graph of the $(k-1)$-mers, labelling each edge with the pair (read, position).
- **Simplify the graph** according to some criteria (to reduce the number of problematic areas containing the errors of the reads).
- Perform some variant of the Eulerian path search to infer the sequence(s).

Reads with perfect overlaps induce a common path – a contig – which is detected implicitly without any pair-wise sequence alignment calculation.



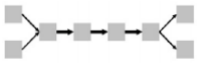
12.7.1 *The simplification step*

Even if errors are present, this procedure works rather well thanks to the **simplification phase** that reduces the number of problematic structures such as:

- **Spurs**, **short dead-end divergences** from the main path usually produced by **sequencing error** toward one end of a read.

- **Bubbles**, **paths that diverge and then converge**, usually produced by **sequencing error toward the middle of a read**.

- **Frayed ropes**, paths that converge and then diverge, and **cycles**, both usually produced by **repeats in the target DNA**.



All the described situations can occur together, creating a very complex graph structure.

12.7.2 *Variants of the Eulerian path (to handle repetitions)*

As said before, the De Bruijn approach uses a variant of the standard Eulerian path to take into account the **presence of repeated k-mers**. They can happen in the same read or in different reads: in the first case we can modify the Eulerian path by allowing multiple visits to the edges corresponding to repetitions; in the second case the corresponding edge contains significantly more fragment positions that is non-repetitive neighbours.

12.7.3 *Example on the De Bruijn approach*

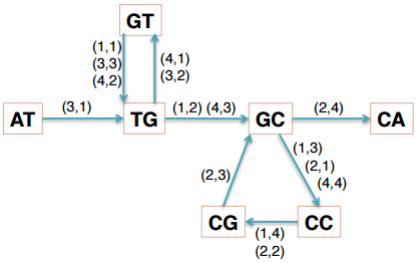
Let's consider the sequence `a = ATGTGCCGCA` that we want to study. The resulting reads are the following (notice that in this example we do not consider the complementary reads):

```
f1 = GTGCCG
f2 = GCCGCA
f3 = ATGTG
f4 = TGTGCC
```

For $k = 3$ the resulting 3-mers are:

```
ATG: (3,1)
TGT: (3,2),(4,1)
GTG: (1,1),(3,3),(4,2)
TGC: (1,2),(4,3)
GCC: (1,3),(2,1), (4,4)
CCG: (1,4),(2,2)
CGC: (2,3)
GCA: (2,4)
```

Notice that many reads have multiplicity (many copies of them on the whole sequence). The De Bruijn graph produced using the 2-mers is showed below.



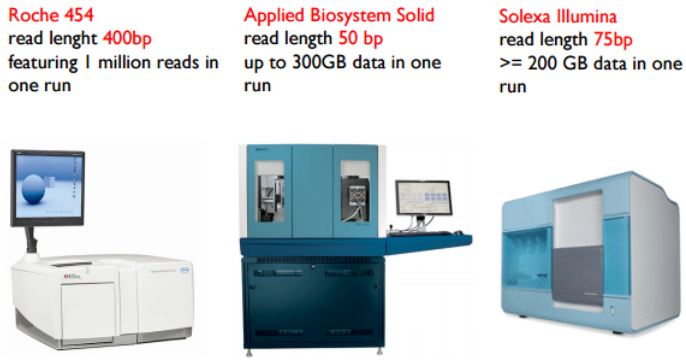
A possible Eulerian path within this graph, corresponding to a single contig, is:

```
AT -(3,1)-> TG -(3,2)-> GT -(3,3)-> TG -(1,1)-> TG
-(1,2)-> GC -(1,3)-> CC -(1,4)-> CG -(2,3)-> GC
-(2,4)-> CA
```

Of course this is not the only path we can find, for reasons listed before.

12.8 NGS: NEXT GENERATION SEQUENCING

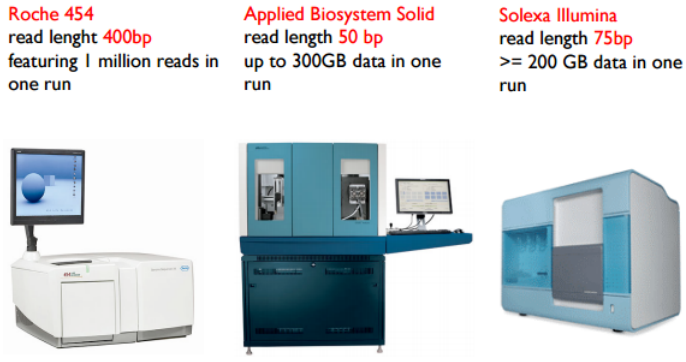
In recent years faster and cheaper techniques have been developed for genome sequencing; the **size of the reads** produced by these techniques varies sensibly, depending on the applied technique[1]:



The choice of the sequencing strategy depends on many factors, such as the **size and complexity of the target genome**. The complexity of the problem, in fact, depends linearly on the size of the genome!

12.8.1 *Phred and the quality of the sequence*

Phred is a tool, developed by Green and Ewing, devised to **assess the quality of a DNA sequence** assigning a quality value (***Phred-score***) to each base of a read. These scores work as reliability values, and are obtained using an idea we're already familiar with: the farthest you go from the starting site of a read, the lower the quality of the read; by using this notion, the electropherogram of the DNA sequence and other statistical techniques, the quality value is obtained.



1 For example reads with length $\leqslant 100$ are used mostly in hybridization techniques.

In practice a Phred-score is usually computed as the negative logarithm of the probability of error for each base, P:

$$\text{Phred-score} = -10\log_{10}P$$

For example, a Phred-score of 20 corresponds to a probability of error of around 1%. At the end of the analysis Phred reports in a file the Phred-score of each base.

12.9 ASSEMBLY PROGRAM: SANGER METHOD

The Sanger method can use several programs in the fragment-assembly phase. The most interesting one is perhaps ***Arachne***, a public domain software for whole genome assembly developed at the MIT. Efficient and accurate, Arachne uses base-pair information and adopts a very efficient overlap detection heuristic. Furthermore, it can be used in combination with other tools such as Phred and Consed. **Consed**, in particular, is a tool used to perform the **finishing phase**: for each contig it shows the multiple alignment of the reads and the proposed consensus sequence, visualizing also the quality of each base. Finally, another interesting tool is **Phrap** (Phragment Assembly Program), optimised for shotgun sequencing with BAC but also used for bacterial genomes. It's used in combination with Phred and Consed.

13

UNDERSTANDING THE GENOME

Sequencing a DNA strand is **only the starting point** of any analysis: once we know the sequence we must also understand its meaning. In other words, to make use of a sequenced genome we must understand its components: the **process of assigning identities and functions** to sequences within the genome is called ***genome annotation***.

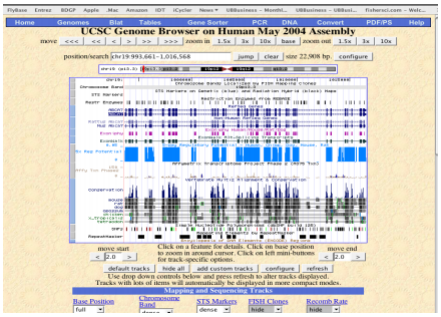
13.1 GENOME ANNOTATION

The annotation phase requires **many integrated tools** for the analysis and visualization of the sequenced genome. Its main goal is to **infer genes**; still, since genes are only a 2% of the human genome, researches also focus on the **detection of other information**: noncoding RNA areas (rRNA, tRNA, ...), gene promoters, structural and regulatory sequences to give the strand its specific shape, junk sequences with no apparent functions, ... It is still controversial how much of the genome is composed of any of these classes and it's still difficult to assign a meaning and a function to *every* possible fragment of DNA.

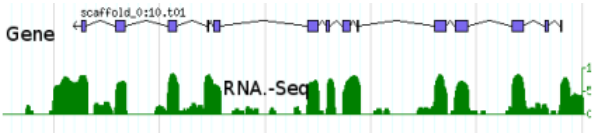
13.1.1 *Gene inference*

Protein-coding genes can be found ***ab initio*** – without knowing anything on the genome except its sequencing. Genes are found by looking at their **recognizable features**: known promoters or start/stop motifs, codon bias, ... Softwares (machine learning, artificial neural network) are used to scan the genome and identify these features: some of them work quite well, especially in bacteria and simpler eukaryotes with smaller and more compact genomes. With more complex eukaryotes, instead, the analysis is quite difficult (since genes can be found within introns of other genes, etc). Since these studies are very complex, **visualization is key**.

Another approach to find genes is to **gather transversal information** – a genome of a similar organism which has already been annotated, for example; by comparison, genes are found in the new



genome. Another powerful technique is called **RNA-seq**, where both the whole genome and the **transcriptome** (the RNA sequence produced by the transcription of a gene) are sequenced: the idea is to map the two to find the positions of each gene. This is done by using many heuristic information – one, for example, is the *coverage*:



As we can see, the parts where the transcriptome "insists" (green parts) correspond to a potential gene (blue parts). Of course, noise can be present so this analysis *it*'s not an easy task.

All in all, annotation uses the following information:

- BLAST searches for similarities;
- Hidden Markov Models to model of specific genes or gene families;
- **Context information** and hints to determine where genes are positioned;
- Experimental data about an organism's capacities can be used to decide whether the relevant functions are present in the genome;
- Biochemical pathway/subsystem information. If an organism has most of the genes needed to perform a function, missing components are probably present too;

13.1.2 *Non-coding RNAs*

Non-coding RNAs are shorted genes which are far **more difficult to detect since they have no associated signals**, so to speak. Their

search is based on the study of the secondary structure, homology (between alignment of related species) and practical experiments.

13.2 PROBLEMS WITH GENE ANNOTATION

Once genes have been identified we need to assign them names and functions. Since gene names have to be **recognizable by computers** we need to use a vocabulary with standardized words and punctuation – not an easy task since **a standard has yet to be developed**: since gene identification is based on experimental works and most annotations are in prose and definitely not standardized ("gene X in my organism is similar to gene Y in another organism that has been experimentally determined to have a such-and-such function"). Furthermore, we need **tools to assess the reliability of gene function predictions**. Finally, we need to determine **who's going to perform annotations**: reasonably, scientists are experts of only a handful of genes, not of all of them! The basic idea is to have experts annotate all the examples of the genes they're familiar with[1], and to do as much automated annotation as possible, with trained personnel examining only the hard cases.

13.2.1 *Gene ontology*

Quite often there is more than one word used to describe the same phenomenon, and the same word is often used to describe completely different phenomena. The **Gene Ontology (GO) consortium** is an attempt to **describe gene products** with a controlled vocabulary. Each GO term is given a number of the form `GO:nnnnnnn` (7 digits), as well as a term name; for example, `GO:0005102` is *receptor binding*. The terms are arranged in a hierarchy that is a directed acyclic graph. There are **3 root terms**: biological process, cellular component, and molecular function:

- Cellular component describes what larger structure the gene product is part of ("ribosome", "endoplasmic reticulum", "cytoplasm", etc).
- Molecular function describes activities that occur at the molecular level ("catalytic", "binding, etc).

1 Problematic: first we need to get the experts for *all* possible genes and second we need to keep them interested!

- Biological process describes the higher level activity that the molecular function contributes to ("signal transduction", etc).

Terms range in a hierarchy from very specific to very general. Still, this structure is difficult to use with unfamiliar gene functions.

14

PHYLOGENETIC TREES

The final topic we will focus on is phylogenetic trees – **tree representations of the evolutionary relations between species**.

14.1 PHYLOGENY AND THE HISTORY OF THE TAXONOMIC PROBLEM

The problem of relating different organisms has been known for centuries; in the beginning, when genetic or evolutionary information was not available, organisms were classified by comparing their physiology (**taxonomic problem**, completely based on the phenotypes of organisms). When genetic data became available, however, **genotype-based techniques** were proposed, using highly-conserved genes (which are universally present and rarely subject to duplications) or genes with slow evolutionary rates (16S rRNA, etc); furthermore, it is also possible compare whole genomes of different organisms (phylogenomics).
Phylogeny is the study of the **evolutionary relations among different organisms**; such relations are usually investigated using the genotypes of the organisms. Note that phylogeny has **two main hypotheses**:

- There is a **common ancestor between all organisms**;
- New species are born when a population spits into two separate communities, which become **different species by mutation**.

The evolutionary relations between organisms are represented using **phylogenetic trees**.

14.1.1 *Brief history of taxonomy*

Historically, the taxonomic comparison of organisms was born with **Linnaeus**, who is nowadays considered the father of modern taxonomy. Linnaeus was convinced that both the number and features of species were fixed. A rudimentary concept of evolution was later described by a French biologist, **Lamarck**, who first used a tree branch-

ing from a common ancestor to proposed the idea of animal evolution. It was Darwin, however, who first used an **unrooted phylogenetic tree** to describe his theory of evolution, causing a revolution in biology. Nowadays phylogenetic trees are widely used to describe the relations between groups of living being, just like the example below. Notice that trees can be much larger than this!

14.2 EVOLUTION AND MUTATIONS

The evolutionary process is the result of two different events:

- **Transmission of mutated genetic information**. Despite being very accurate DNA replication often produce mutations, caused either by environment factors altering the DNA sequence or by errors in the DNA replication process. Mutations are generally *destructive*: reproduction is usually impossible for the mutated creature; still, in the remote case that mutations are transmitted to all the descendants of the mutated creature, a new species might be born.
- **Survival of the fittest**. A mutated population must respond well to the environment, otherwise it is destined to disappear.

As we can see mutations are **the basis of the evolution process** that, starting from a primitive living organism, produced the present variety of living forms.
Mutations can be punctual substitutions (transitions or transversions), whole substitutions, deletions, insertions or inversions:

```
A   5′ -ATGGACCGGATGGATGATGGACCGTTAGGAT- 3′
                  punctual substitutions
B   5′ -ATGGACCGAATGGCTGACGGACCGTGAGGAT- 3′
                  deletions
C   5′ -ATGGAC.TGGCGACGGACCGTGAGGAT- 3′
                  insertion
D   5′ -ATGGAC.TGGCTGACGGAAGTCCGTGAGGAT- 3′
                  inversions
E   5′ -AGTGCA.TGGCTGACGGAAGTCCGTGAGGAT- 3′
.....................................................................
    5′ -ATGGACCGGATGGATGATGGA---CCGTTAGGAT- 3′
        |  |       ||| ||| |||    |||| |||||
    5′ -AGTGCA----TGGCTGACGGAAGTCCGTGAGGAT- 3′
```

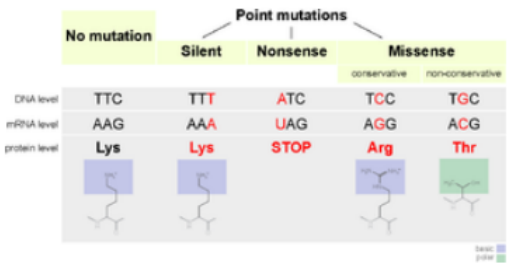
- **Punctual substitutions**.

Punctual substitutions are point mutations, involving a single nucleotide or amino acid. Interestingly, punctual substitutions tends to accumulate with time, eventually producing wildly different sequences. They are also divided into **transitions** and **transversions**:

- Transition: more common, a purine is substituted by another purine ($A \leftrightarrow C$) or a pyrimidine is substituted by another pyrimidine ($C \leftrightarrow T$);
- Transversion: less common, a purine is substituted by a pyrimidine or viceversa ($C/T \leftrightarrow A/G$).

Suppose punctual substitutions occur within the **coding region of a protein gene**: then the resulting gene may be classified in 3 ways depending on the "consequence" of the mutations:

- **Silent mutations**: a different code encodes the same amino acid;
- **Missense mutations**: different code encodes a different amico acid;
- **Nonsense mutations**: different code truncates the protein.



- **Insertions and deletions**

Insertions happens when one or more extra nucleotides are added into the DNA; deletions happen instead when one or more nucleotides are removed from the DNA. In both cases the result is **highly destructive** since the code is shifted, altering the gene product (**frameshifting**: codons are parsed incorrectly, which results in useless proteins).

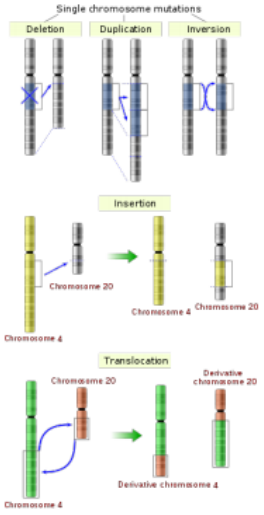
CTGGAG
CTGG**TGG**AG

- **Inversions**

With inversions, entire sections of DNA are reversed, involving either a small set of bases within a gene or larger regions containing several genes.

```
AGGTCTTA
AGTCTGTA
```

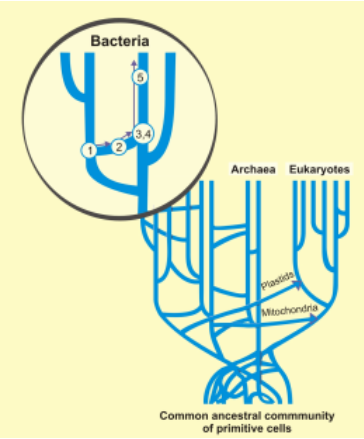
Mutations can also involve large parts of DNA (**large scale mutations**); the picture below shows 5 types of mutations of the chromosomal structure.



Finally, cells have **repairing mechanisms** that catch and repair most of the mutations that occur during DNA replication or from environmental damage. As we age, however, this function is less effective.

14.2.1 *Other forms of DNA transmission: horizontal gene transfer*

Reproduction is not the only way mutations can be transmitted; bacteria, in fact, also perform *horizontal gene transfer*, a transfer of genes between organisms without reproduction. This is one of the main reasons behind bacterial antibiotic resistance; furthermore, it means that bacteria **cannot be studied simply using phylogenetic trees**.



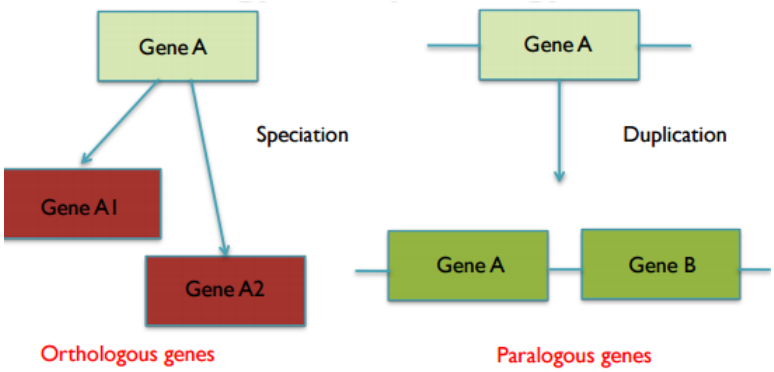
14.2.2 *Higher level: gene structure*

Let's now consider mutations from a gene point of view. What does it mean that a gene is mutated? Remember the following definitions:

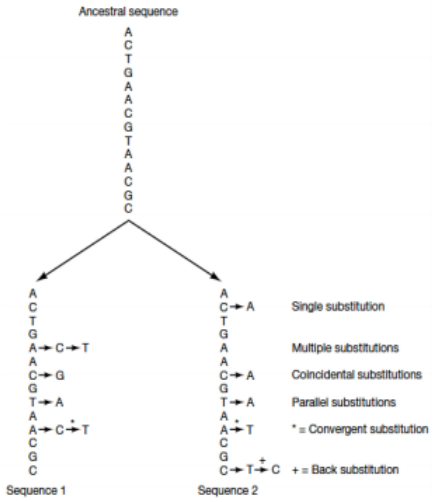
- Two genes are **homologous** if they are inherited from a common ancestor.
- Two genes are **orthologous** if they diverged after a speciation event: different species inherit the same gene of the ancestors, with small variations. They usually maintain the same function;
- Two genes are **paralogous** if they diverged after a duplication event. They usually have new functions, even if related to the original one.

14.3 GENETIC DISTANCE

It is usually **impossible to be sure of the evolutive relation inferred between a group of organisms**, even if their genomic sequences are completely known. To understand why, suppose we have an ancestral



sequence and two populations that inherited such sequence and then evolved in different ways, as in the following example:



When comparing the sequences of these communities we **inevitably underestimate the number of mutations**: for example, we overlook the mutation T $\rightarrow$ A because it happens in both sequence and we have now way of knowing it. Now, many methods used by phylogeny are based on the calculation of a "distance", proportional to the number of differences in the alignment of the sequence; it's clear that such **distance is a total guess**: the number of mutations is always greater than the observed differences.

14.4 MODELS TO COMPARE SEQUENCES

When comparing different organisms we can focus on both DNA (sequences of nucleotides) or proteins (sequences of amino acids). Each approach has a different structure:

- Models based on DNA are usually **mechanical models**, where substitution rates and distances are estimated for every data set analyzed;
- Models based on proteins are instead **empirical models** created by estimating parameters from a large data set, once and for all. In other words, the parameters are fixed and reused for every data set.

All in all, it seems that **nucleotide-based models are the best choice** for several reasons: they allow studying non-coding regions and changes in the DNA sequence that amino acid might hide due to silent mutations; furthermore, the DNA alphabet is made of only four symbols so **mathematical models are simpler**.

14.5 NUCLEOTIDE-BASED MODELS

To determine the evolution process and estimate the actual number of substitutions **stochastic models** have been developed. Each model is composed by a substitution score matrix, a definition of distance and few **unrealistic and *a priori* assumptions** used to simplify computations:

- **Molecular clock hypothesis**: different linages have the same constant substitution rate. In other words, the "rhythm" of evolution is the same for all species[1].
- **All sites evolve independently**: two mutations in different moments in time are not related.
- **All sites are equally subject to change**: mutations happen independently from the position on the sequence;

1 The existence of a molecular clock was first attributed to Emile Zuckerkandl and Linus Pauling (1962): starting from fossil evidence that the number of amino acid differences in hemoglobin between different lineages changes roughly linearly with time, they generalised this observation to assert that the rate of evolutionary change of any specified protein is approximately constant over time and over different lineages. This allows the estimation of the time when two proteins (or the corresponding genes) have diverged but, since it's a very strong assumption, it must be used with caution. Furthermore, the availability of complete genomes has shown that a **universal molecular clock does not exist**: every protein and gene has its own evolution dynamics, with accelerations or decelerations in the evolution of the same gene in different lineages. In homolog proteins or genes it is more common to find a **local molecular clock**, valid only for a specific taxonomical group.

- All sequences have the **same base composition**: the statistics for each symbol are always uniform.
- **Substitutions occur randomly** among the four types of bases.

Different models may have different assumptions; for example, by assuming the constant substitution rate and stationarity we can build one substitution probability matrix such as this one, where P_{AC} is the probability that A turns into C in a unit of time. Note that in many models the matrix is symmetric, that is $P_{AC} = P_{CA}$.

$$\begin{array}{c} \\ A \\ C \\ G \\ T \end{array} \begin{array}{c} \begin{array}{cccc} A & C & G & T \end{array} \\ \begin{bmatrix} P_{AA} & P_{AC} & P_{AG} & P_{AT} \\ P_{CA} & P_{CC} & P_{CG} & P_{CT} \\ P_{GA} & P_{GC} & P_{GG} & P_{GT} \\ P_{TA} & P_{TC} & P_{TG} & P_{TT} \end{bmatrix} \end{array}$$

14.6 NUCLEOTIDE: JUKES-CANTOR MODEL (1969)

The Jukes-Cantor model is the simplest of them all, with many simplifying **assumptions**:

- Every previous hypothesis is assumed;
- There's no difference between transversions and transitions;
- Every base has the same frequency, $f_A = f_C = f_G = f_T = \frac{1}{4}$;
- Mutation rates are equal.

Given these assumptions the substitution rate[2] α is the same for any nucleotide with any other. The substitution probability matrix is therefore:

$$\begin{bmatrix} p & \alpha & \alpha & \alpha \\ \alpha & p & \alpha & \alpha \\ \alpha & \alpha & p & \alpha \\ \alpha & \alpha & \alpha & p \end{bmatrix}$$

Notice that α is the same for every possible substitution and that the sum on each column must be 1. As for the definition of distance,

2 It's the **probability of one substitution**.

given two sequences A and B the expected number of mutations per site[3] is the following:

$$d_{AB} = -\frac{3}{4}\ln\left(1 - \frac{4}{3}f_{AB}\right)$$

...where $f_{AB} = \frac{\text{Number of different sites}}{\text{Total number of sites}}$ is **substitution frequence** between sequences A and B.
To sum up, this method works only when sequences are related (otherwise the distance cannot be computed and tends to infinity).

14.6.1 *Example of JC69*

Consider the following DNA sequences:

```
S1 = AACTGG
S2 = AACTGT
S3 = GACGGT
```

To compute their distances using the Jukes-Cantor model we first compute the ratio of sites that differ between every couple of sequences:

```
S1-S2: AACTGG
       .....T (1 substitution)

S1-S3: AACTGG
       G..G.T (3 substitutions)

S2-S3: AACTGT
       G..G.. (2 substitutions)
```

They corresponds to the following substitution frequencies:

- $f_{s1s2} = 1/6$
- $f_{s2s3} = 2/6 = 1/3$
- $f_{s1s3} = 3/6$

Therefore we simply compute the distances:

- $d_{s1s2} = -3/4\ln(1–4/31/6) = -3/4\ln(7/9) = 0,19$

3 This is estimated by assuming that is has a Poisson distribution in time.

- $d_{s1s3} = -3/4\ln(1–4/31/2) = -3/4\ln(1/3) = 0,82$
- $d_{s2s3} = -3/4\ln(1–4/31/3) = -3/4\ln(5/9) = 0,44$

...and the corresponding distance matrix:

```
   S1    S2

S2 0.19

S3 0.82  0.44
```

14.7 NUCLEOTIDE: KIMURA MODEL (1980)

The Kimura model is more realistic since it **distinguishes transitions from transversions**, assigning two different substitution rates (α for transitions, β for transversions). As a result the substitution probability matrix is:

$$\begin{array}{c} \\ A \\ C \\ G \\ T \end{array} \begin{array}{c} \begin{array}{cccc} A & C & G & T \end{array} \\ \begin{bmatrix} p & \beta & \alpha & \beta \\ \beta & p & \beta & \alpha \\ \alpha & \beta & p & \beta \\ \beta & \alpha & \beta & p \end{bmatrix} \end{array}$$

As for the definition of **distance**, given two sequences A and B the expected number of mutations per site is:

$$d_{AB} = -\frac{1}{2}\ln(1-2P-Q)-\frac{1}{4}\ln(1-2Q)$$

...where P is the **transition substitution frequence** between sequences A and B and Q is the **transversion substitution frequence** between A and B.

14.7.1 *Example*

Suppose we're given again the previous sequences:

```
S1 = AACTGG
S2 = AACTGT
S3 = GACGGT
```

To compute their distances using the Kimura model we first compute the ratio of sites that differ between every couple of sequences:

```
s1-s2: 1 transversion, 0 transitions
s1-s3: 1 transversion, 2 transitions
s2-s3: 1 transversion, 1 transition
```

Then we apply the distance formula:

- $d_{s1s2} = -1/2\ln[(1–1/6)–1/4\ln(1–1/3) = -1/2\ln(5/6)–1/4\ln(2/3) = 0,19$
- $d_{s1s3} = -1/2\ln[(1–21/6–2/6) - 1/4\ln(1–2/3) = -1/2\ln(1/3)–1/4\ln(1/3) = 0.82$
- $d_{s2s3} = -1/2\ln[(1–21/6–1/6) - 1/4\ln(1–1/3) = -1/2\ln(1/2)–1/4\ln(2/3) = 0.69$

The resulting distance matrix is:

```
   S1   S2

S2 0.19

S3 0.82  0.69 <- larger distance
```

Notice that given the same strings, the distance between S2 and S3 is larger with the Kimura model.

14.8 OTHER MODELS WITH NUCLEOTIDES

Other stochastic models have been developed to estimate distances:

1. **Tamura model (1992)**: it considers different substitution rates for transitions and transversions, taking into account the $C + G$ frequency of the considered sequences (θ parameter);
2. **Felsenstein model (1981)**: it extends the Jukes-Cantor model by considering the nucleotide frequency of the analysed sequences (π_1, π_2, π_3, π_4 parameters).
3. **Hasegawa et al. model (1985)**: it extends the Felsenstein model by considering a different substitution rate for transitions and transversions.
4. **Lanave et. al. model (1984)**: the most general model, it is known as the `REV` (reversible) model or `GTR` (General Time Reversible) model. It considers the nucleotide frequency of the

analysed sequences and it assigns a different probability to each possible substitution. It assumes the **reversibility of substitutions**.

Modello	Matrice delle probabilità delle sostituzioni nucleofile	Composizione in basi nello stato stazionario (f_i, i = A, C, G, T)	Numero parametri
Jukes & Cantor (1969)	$\begin{matrix} p_{11} & \alpha & \alpha & \alpha \\ \alpha & p_{22} & \alpha & \alpha \\ \alpha & \alpha & p_{33} & \alpha \\ \alpha & \alpha & \alpha & p_{44} \end{matrix}$	$\left[\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right]$	1
Kimura (1980)	$\begin{matrix} p_{11} & \beta & \alpha & \beta \\ \beta & p_{22} & \beta & \alpha \\ \alpha & \beta & p_{33} & \beta \\ \beta & \alpha & \beta & p_{44} \end{matrix}$	$\left[\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right]$	2
Tamura (1992)	$\begin{matrix} p_{11} & \theta\beta & \theta\alpha & (1-\theta)\beta \\ (1-\theta)\beta & p_{22} & \beta & (1-\theta)\alpha \\ (1-\theta)\alpha & \beta & p_{33} & (1-\theta)\beta \\ (1-\theta)\beta & \alpha & \beta & p_{44} \end{matrix}$	$\left[\frac{1-\theta}{2}, \frac{\theta}{2}, \frac{\theta}{2}, \frac{1-\theta}{2}\right]$	3
Tajima and Nei (1982)	$\begin{matrix} p_{11} & \pi_C\alpha & \pi_C\alpha & \pi_T\alpha \\ \pi_A\alpha & p_{22} & \pi_C\alpha & \pi_T\alpha \\ \pi_A\alpha & \pi_C\alpha & p_{33} & \pi_T\alpha \\ \pi_A\alpha & \pi_C\alpha & \pi_C\alpha & p_{44} \end{matrix}$	$[\pi_A, \pi_C, \pi_G, \pi_T]$	4
Hasegawa et al. (1985)	$\begin{matrix} p_{11} & \pi_C\beta & \pi_C\alpha & \pi_T\beta \\ \pi_A\beta & p_{22} & \pi_C\beta & \pi_T\alpha \\ \pi_A\alpha & \pi_C\beta & p_{33} & \pi_T\beta \\ \pi_A\beta & \pi_C\alpha & \pi_C\beta & p_{44} \end{matrix}$	$[\pi_A, \pi_C, \pi_G, \pi_T]$	5
Lanave et al. (1984) Saccone et al. (1990)	$\begin{matrix} p_{11} & \pi_C\beta_1 & \pi_C\alpha_1 & \pi_T\beta_2 \\ \pi_A\beta_1 & p_{22} & \pi_C\beta_3 & \pi_T\alpha_2 \\ \pi_A\alpha_1 & \pi_C\beta_3 & p_{33} & \pi_T\beta_4 \\ \pi_A\beta_2 & \pi_C\alpha_2 & \pi_C\beta_4 & p_{44} \end{matrix}$	$[\pi_A, \pi_C, \pi_G, \pi_T]$	9

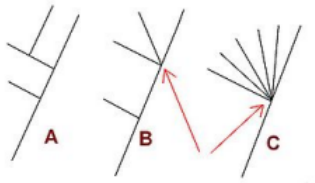
14.9 AMINO ACID-BASED MODELS

Despite being less precise, it is still possible to understand the evolutionary relations of different species using proteins. Empirical scoring matrices for amino acids have been already presented for pair alignment: **PAM and BLOSUM**. They have been determined from substitution probability matrices estimated once for all. The following matrices are roughly equivalent:

14.10 PHYLOGENETIC RELATIONSHIPS

Phylogenetic relationships are usually depicted as trees with the following features:

- **Binary**. Since speciation events where more than two different species are born are rare, a binary solution is efficient. Non-binary phylogenetic trees can still be used to model situations not fully resolved to dichotomies (a tree containing**polytomies**).



- **Nodes** represent taxonomic units. In particular, internal nodes represent *ancestral* taxonomic units, the unknown ancestors of the current species; leaves, instead, represent *operational* taxonomic units, the currently living species we want to study.
- **Edges** reflect the possible ancestor-descendant relationship between species;
- The **root**, if present, is the common ancestor of all the taxonomic units in the tree.

It is important to stress that phylogenetic trees can be **rooted or unrooted**: in rooted trees the direction of the evolutionary path is always specified while in unrooted trees there are many possible evolutionary paths (and are used to illustrate how different species are related, without assumptions on the common ancestry. Remember that it's always possible to **root an unrooted tree** by adding a so-called **outgroup** – a set of at least one taxonomic unit **completely unrelated** to anything else in the tree (the ingroup). The root is set on the **edge between the outgroup and the ingroup**.

14.10.1 *The problem with phylogenetic trees*

The task of reconstructing phylogenetic trees is complex for a very simple reason: given n species, it is **impossible to know the exact way in which they're related**. Naively, we could think of computing all the different rooted and unrooted trees for these n species and search for the best but this is unfeasible since the number of trees is exponential:

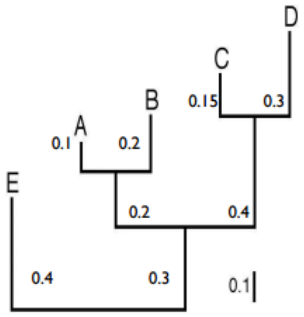
$$N^{unrooted}(n) = (2n-5)!! = \frac{(2n-5)!}{2^{n-3}(n-3)} \text{ with } n \geqslant 3;$$
$$N^{rooted}(n) = N^{unrooted}(n) - 1$$

Given $n = 10$ species, for example, we would obtain 2 million unrooted trees! As a result, many tree reconstruction algorithms and heuristics have been proposed: still, each approach yields a different phylogenetic tree even if the data is the same. Finally, data are inherently noisy, error-prone, and **hard to interpret**.

14.11 PHYLOGENETIC TREES, TIME AND NEWICK NOTATION

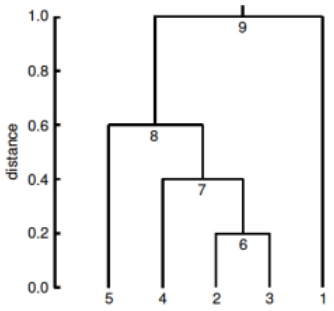
Using a general term, phylogenetic trees can also be called ***dendrograms***. ***Cladograms*** are instead phylogenetic trees where **time is not represented**: they're just a representation of **ancestral-descendant relationships** and nothing more.

If we're interested in representing some attribute of each taxonomic unit we can use **additive trees** or phylograms, where the lengths of the edges are proportional to some attribute of the species (for example time or genetic distance):



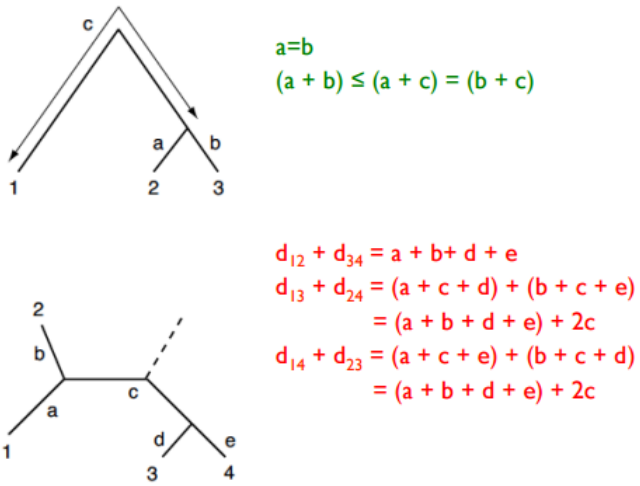
Notice that in additive trees the distance between any pair of leaves is the sum of their distances from their last common ancestor. Finally, if we're interested in representing time only, we can use **ultrametric trees** or chronograms, a special type of additive trees where leaves

are all equidistant from the root. These trees can be used to express evolutionary time or the amount of sequence divergence (molecular clock assumption).

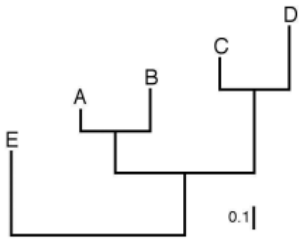


Notice that given a certain distance matrix D, we can construct an additive or ultrametric tree from it only if these conditions are valid:

- **Ultrametric trees**: three-point condition. D can be transformed into an ultrametric tree if and only if for any three leaves i, j, k, two of the distances d_{ij}, d_{kj}, d_{ki} are equal and $\geqslant$ than the third one.
- **Additive trees**: four-point condition. D can be transformed into an additive tree if and only if for any for leaves $1, 2, 3, 4$ two of the sums $d_{12} + d_{34}, d_{13} + d_{24}, d_{14} + d_{23}$ are equal and greater or equal than the third.

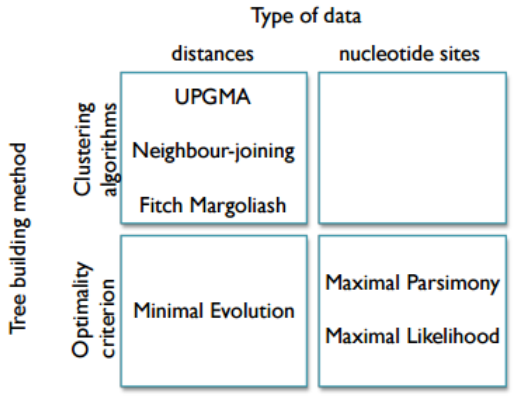


Furthermore, trees can be represented with a more compact notation, the **Newick notation**. For example the tree below can be written as: $((A, B), (C, D)), E)$. One can **add information** on edge lengths: $((A : 0.1, B : 0.2) : 0.2, (C : 0.15, D : 0.3) : 0.4) : 0.3, E : 0.4)$.



14.12 AN INTRODUCTION TO TREE CONSTRUCTION METHODS

There are many different ways to construct phylogenetic trees; organized into **methods using custering techniques** and **methods using some optimization criterion**. Notice that the second group of methods is intrinsically combinatorial and, therefore, heuristic.



We will now start with the first group, based on clustering.

14.13 CLUSTERING METHOD: UPGMA

Developed by Sokal and Michener (1958), the UPGMA – *Unweighted Pair Group Method with Arithmetic Mean* – approach is the **simplest clustering method** to build phylogenetic trees: since it assumes the molecular clock hypothesis it builds **rooted ultrametric trees**. Furthermore, UPGMA assumes that all pairwise distances contribute equally and groups are combined in pairs (dichotomies only).
The algorithm behind UPGMA is iterative. To understand how it works suppose we have four sequences, A, B, C, D with the following distance matrix D:

At each iteration the algorithm:

1. Determines the minimal distance in the distance matrix D and groups the corresponding sequences together: for example, if

$$\begin{array}{c} \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} d_{AB} & & \\ d_{AC} & d_{BC} & \\ d_{AD} & d_{BD} & d_{CD} \end{bmatrix} \end{array}$$

the minimal distance is d_{BC} then B and C are grouped. Notice that the distances, at least in the beginning, are computed as follows:

$$d_{ij} = \frac{1}{|C_i||C_j|} \sum_{p \in C_i, q \in C_j} d_{pq}$$

...where C_i and C_j are two different clusters (in the beginning corresponding to the single sequences).

2. Recomputes the distances between this new cluster BC and all the remaining sequences, A and D. In this step the distance between the new cluster C_k and the remaining sequences h is computed as follows:

$$d_{kh} = \frac{d_{ih}|C_i| + d_{jh}|C_j|}{|C_i| + |C_j|}$$

In this example $d_{A,BC}$ is the mean of the distances between A and B and A and C.

$$d_{(A,BC)} = \frac{(d_{AB} + d_{AC})}{1 * 2}$$
$$d_{(BC,D)} = \frac{(d_{BD} + d_{CD})}{1 * 2}$$

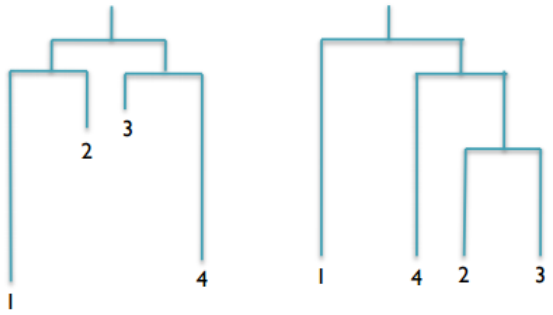
$$\begin{array}{c} \\ BC \\ D \end{array} \begin{array}{c} \begin{array}{cc} A & BC \end{array} \\ \begin{bmatrix} d_{(A,BC)} & \\ d_{AD} & d_{(BC,D)} \end{bmatrix} \end{array}$$

3. Add a new node BC with B and C as children and place it at height $d_{BC}/2$.

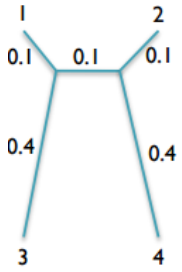
Once we're left with only two ungrouped taxonomic units, their union is the **root of the ultrametric tree**.
Time complexity is $O(n^2)$, where n is the number of considered taxonomic units. Note that this technique is rather good and simple; still, **if the matrix** D **is not ultrametric** the resulting tree will be distorted by UPGMA.

14.14 CLUSTERING METHOD: NEIGHBOUR-JOINING

Developed by Saitou and Nei (1987), the *Neighbour-joining* method does no assume the *molecular clock hypothesis*, producing **unrooted ad-**



ditive trees. Furthermore, the previous notion of distance is not used to cluster: sequences with the minimal distance are not necessarily neighbors. As an example, consider an initial situation where the distance matrix D represents each sequence as a singleton that can be potentially grouped with another one. It is represented as a general ***star-topology tree*** such as the one below:



Instead of clustering sequences using the notion of minimal distance, Neighbour-joining bases its clustering on an additional matrix H, where the element h_{ij} is the difference between the distance d_{ij}, the average distance of i from every other sequence and the average distance of j from every other sequence.

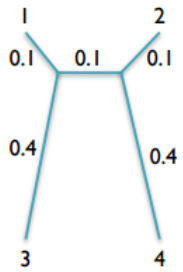
$$h_{ij} = d_{ij} - r_i - r_j$$

Where:

$$r_i = \frac{1}{|L|-2} \sum_{k \neq i \in L} d_{ik}$$

A cluster is therefore obtained by picking the couple of sequences with the **minimal** h **value** in the matrix. Getting back to our previous example, since we have these computations...

...the first clusters are composed by nodes 2-4 and 1-3. Interestingly, the distance between unclustered pairs 1-2 and 3-4 is lower than the distance of clustered pairs 2-3 and 1-3. To sum up, the algorithm works as follows:



In the example:

$r_1 = 1/2\ (0.3 + 0.5 + 0.6) = 0.7$
$r_2 = 1/2\ (0.3 + 0.6 + 0.5) = 0.7$
$r_3 = 1/2\ (0.5 + 0.6 + 0.9) = 1$
$r_4 = 1/2\ (0.6 + 0.5 + 0.9) = 1$

$h_{12} = 0.3 - (0.7 + 0.7) = -1.1$
$h_{13} = 0.5 - (0.7 + 1) = -1.2$
$h_{14} = 0.6 - (0.7 + 1) = -1.1$
$h_{23} = 0.6 - (0.7 + 1) = -1.1$
$h_{24} = 0.5 - (0.7 + 1) = -1.2$
$h_{34} = 0.9 - (1 + 1) = -1.1$

1. Given the distance matrix D, compute the matrix H where element h_{ij} is defined as above and pick the pair i, j with the minimal h_{ij} value;
2. Define a new node k and set the distances between k and every other sequence as:

$$d_k m = \frac{1}{2}(d_{im} + d_{jm} - d_{ij}) \tag{3}$$

3. Join k to i and j, with distances:

$$d_{ik} = \tfrac{1}{2}(d_{ij} + r_i - r_j)$$
$$d_{jk} = d_{ij} - d_{ik}$$

4. Remove i and j and add k to the clusters to consider.

The algorithm ends when only two sequences i, j are left: an edge of length d_{ij} is added between them.

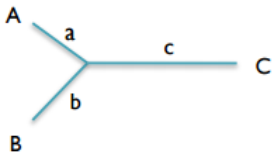
Again, to make this algorithm work the **distance matrix must be additive**; real data, however, are **at best approxmimately additive**. We can still apply the Neighbour-joining method but the reconstruction of the correct tree is no longer guaranteed.

14.15 CLUSTERING METHOD: FITCH & MARGOLIASH METHOD

A variant of Neighbour-joining, the *Fitch & Margoliash* method also rejects the molecular clock hypothesis, producing an unrooted tree from an additive distance matrix. The algorithm is rather **costly** since since it's combinatorial: it generates all possible unrooted trees and *a posteriori* finds the best one.

Chosen set of 4	Sum of distances
ABCD	$n_{AB} + n_{CD} = 22 + 18 = 40$
	$n_{AC} + n_{BD} = 39 + 41 = 80$
	$n_{AD} + n_{BC} = 39 + 41 = 80$
ABCE	$n_{AB} + n_{CE} = 22 + 20 = 42$
	$n_{AC} + n_{BE} = 39 + 43 = 82$
	$n_{AE} + n_{BC} = 39 + 41 = 82$
ABDE	$n_{AB} + n_{DE} = 22 + 10 = 32$
	$n_{AD} + n_{BE} = 39 + 43 = 82$
	$n_{AE} + n_{BD} = 41 + 41 = 82$
ACDE	$n_{AC} + n_{DE} = 39 + 10 = 49$
	$n_{AD} + n_{CE} = 39 + 20 = 59$
	$n_{AE} + n_{CD} = 41 + 18 = 59$
BCDE	$n_{BC} + n_{DE} = 41 + 10 = 51$
	$n_{BD} + n_{CE} = 41 + 20 = 61$
	$n_{BE} + n_{CD} = 43 + 18 = 61$

	A	B	C
A		22	39
B			41
C			



14.15.1 *The algorithm*

At each iteration the algorithm works with **three taxonomic units**, computing their distances from one another. Suppose we're given three sequences with the following distances:

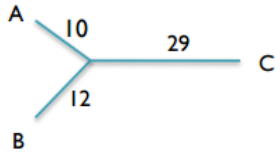
We're interested in finding out a, b, c. Since A and B are the closest units in D the lengths of their edges are expected to be shorter. To do so we associate to each distance in D a corresponding equation, obtaining:

1. $d_{AB} = a + b = 22$
2. $d_{AC} = a + c = 39$
3. $d_{BC} = b + c = 41$

By solving this system (subtract 3 from 2 and add 1: $a = 10, b = 12, c = 29$) we obtain the various distances and adjust the previous tree proportionally.

1. $d_{AB} = a + b = 22$
2. $d_{AC} = a + c = 39$
3. $d_{BC} = b + c = 41$

In the general case, with more than three sequences, A and B represent the closest pair of sequences while C is a composite with everything else. Then the algorithm is:



- Given a distance matrix D, A and B are the closest pair of sequences while C is the composite group;
- Compute d_{AC}, the average distance between A and all the OTUs in C, and d_{BC}, the average distance between B and all the OTUs in C, obtaining the distances "a" and "b" like in the previous example;
- Update the distance table D by computing the distances between the single group AB and all the remaining taxonomic units;
- Identify the next pair of most closely-related taxonomic units and iterate everything again.

This procedure is then repeated **starting from all possible pairs of taxonomic units**; by computing the distances between each pair of OTUs for each tree, we find the "best" tree – either the tree that **minimizes the sum of all paths** (maximal parsimony) or the three that has distances proportional/**corresponding to the original distance table**.

14.16 CLUSTERING METHOD: MINIMAL EVOLUTION METHOD

The *Minimal Evolution* method is a combinatorial approach that, given n sequences, produces all possible unrooted trees. Each of them is thus evaluated to find the best one. Notice that the algorithm associates to each tree a corresponding score L, which is computed as the sum of the lengths of all the $2n-3$ edges of the tree:

$$L = \sum_{i=1}^{2n-3} l_i$$

where l_i is the length of the ith edge of a certain tree. The best tree is the one with the lowest score L, provided that:

- $l_i > 0$, meaning all edges have positive length; this is not guaranteed in Fitch & Margoliash method.

- $p_{ij} \geqslant d_{ij}$, for all pairs of terminal nodes i, j (the distances on the tree, p_{ij}, can never be smaller than the directly observed distances d_{ij}).

Notice that with these two criteria, **both *maximal parsimony* and the correspondence with the initial distance matrix** D are considered.

14.17 SEQUENCING METHOD: MAXIMUM PARSIMONY SEQUENCE METHOD

Representing genetic relations using pairwise distances causes a **loss of information** with respect to simply aligning sequences; this realization is behind the *Maximum Parsimony* sequence method, which:

- Does not involve clustering or distance tables;
- Does not assume any model of evolution;
- Is based only on multiple alignment;
- Assumes only the **maximum parsimony criterion** which states that the phylogenetic trees describing best the relationship between n sequences it the most parsimonious – the one requiring the smallest number of changes to explain the observed differences between sequences[4].

As a result, the MP algorithm produces **cladograms**, unrooted trees with no information on the edges. The algorithm works as follows:

1. Perform a **multiple alignment** of the given sequence and select the **informative sites** – columns where there are at least two different symbols and each of them is repeated at least twice.
2. For each informative site **produce every possible unrooted tree** with every possible topology (sequences are distributed in every possible way) and **determine the number of substitutions** required by each of them. For example, given 4 sequences, they

4 Notice that Maximum parsimony can be justified in different ways:

- It is a general principle in science known as Occam's razor: there is no need to make more assumptions than necessary to explain the observations.
- It is appealing since it does not require any explicit assumption, thus it can be applied when the data cannot be easily modelled.

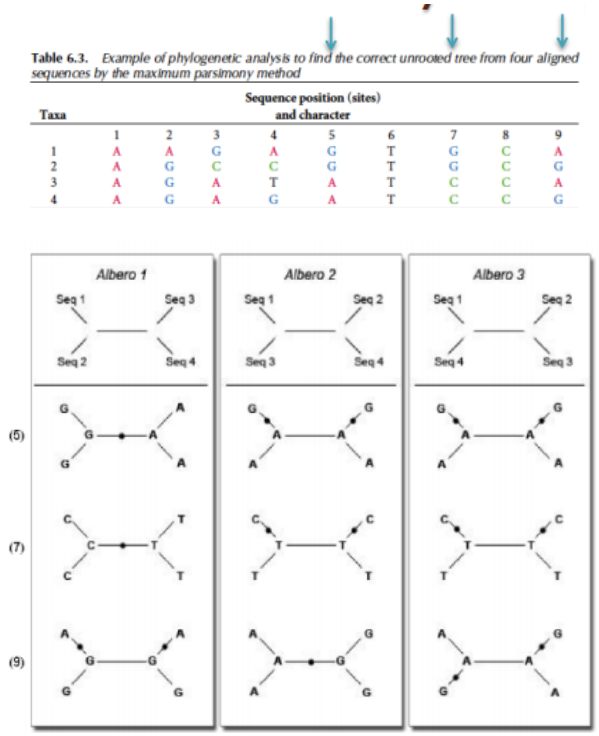


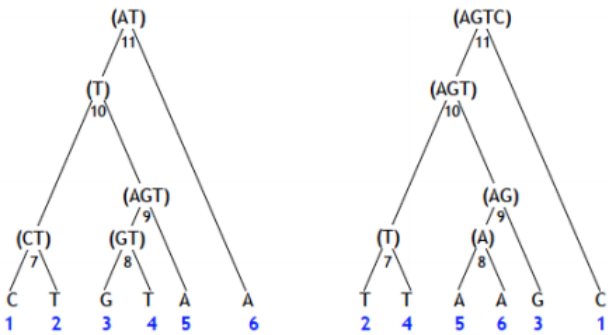
Table 6.3. *Example of phylogenetic analysis to find the correct unrooted tree from four aligned sequences by the maximum parsimony method*

Taxa	Sequence position (sites) and character								
	1	2	3	4	5	6	7	8	9
1	A	A	G	A	G	T	G	C	A
2	A	G	C	C	G	T	G	C	G
3	A	G	A	T	A	T	C	C	A
4	A	G	A	G	A	T	C	C	G

can be organized in three possible ways, obtaining 4 (left), 5 (center), 6 (right) substitutions.

3. Select the maximum parsimony tree – the one with the **minimum number of substitutions** for the informative areas.

Notice that with more than 4 sequences the **computation becomes so complex** that a heuristic search is generally done to compute the score of each tree (the number of substitutions). Usually the **Fitch parsimony** algorithm is used: the tree is rooted and then visited bottom-up, counting the number of changes from the leaves to the root. For example, the tree on the left returns 4 while the tree on the right returns 3: every time an ancestral node is the union of two different symbols we have to count a substitution.



To sum up, this method is very simple since it has **only one main assumption**, *Maximum parsimony*; this can be an advantage if nothing is known of the data set but also a disadvantage, since no other hypothesis is exploited. Furthermore, it often produces many equally parsimonious trees: *consensus methods* are applied in this case. Lastly, the method produces unrooted trees that can't show stronger evolutionary relations between sequences: the algorithm squeezes together everything (rates of change along all branches of the tree are assumed to be equal).

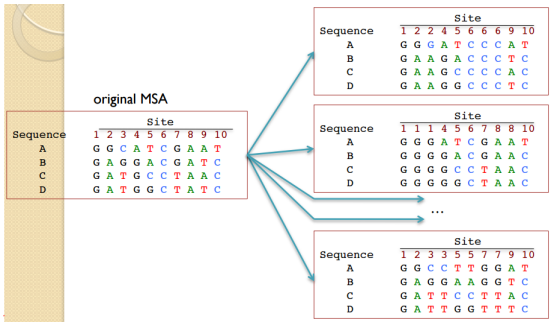
14.18 SEQUENCING METHOD: MAXIMUM LIKELIHOOD TECHNIQUE

Maximum likelihood is the best method in terms of quality of the reconstruction and quantity of information associated to the tree. It is a combinatorial technique where the **optimization criterion is the likelihood of the evolutionary model governing the data**; as for the assumptions, independence between sites is assumed and, similarly to *Maximum Parsimony*, it considers all topologies for each column of the MSA. Finally it is **computationally expensive**, hence heuristics are used.
To understand how *Maximum likelihood* works, consider that an evolutionary model specifies the probability of a *symbol mutation* everywhere on the tree, and can be defined as the couple $H = (T, \theta)$ where:

- T is the specific topology of the tree we're considering;
- θ is additional information specifying the probability of a symbol mutation in any place in the tree;

The **likelihood of a model** H for a data set D – representing a multiple alignment of sequences – is defined as the **probability of observing the data** D **given the model** $H = (\theta, T)$; in other words, the more an evolutionary model is capable of producing the data we're working on, the greater its likelihood. Now, the algorithm computes all possible trees T and evaluate them all to maximize the likelihood L. Since independence between sites is assumed, the likelihood is the product of the likelihoods of all sites of the tree:

$$L = \prod_{i=1}^{k} L_i$$



Notice, however, that likelihoods are generally very small so the logarithm of the likelihood is generally used – and we have to minimize it.

14.19 TREE ASSESSMENT AND THE BOOTSTRAP METHOD

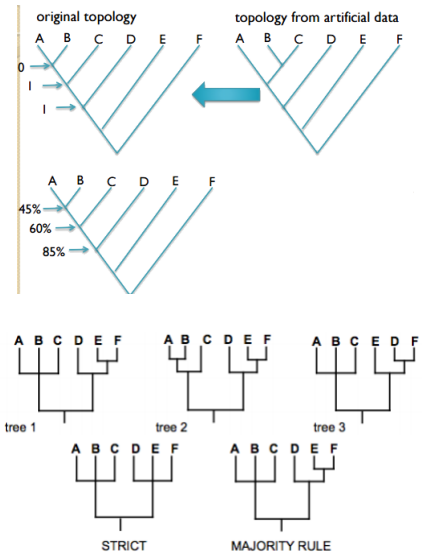
The output of every algorithm seen so far is a tree or a collection of trees. **How much should we trust these trees?** A statistical method called bootstrap can be employed to assess the significance of the result for methods based on sequences. Notice that the bootstrap method gives a measure of the **robustness of a tree but says nothing on its correctness**. The algorithm works by producing an artificial sample set and comparing our results against it.

14.19.1 *Bootstrap algorithm*

Given a dataset consisting of a multiple alignment of sequences:

1. Generate a new artificial dataset by picking randomly columns from the alignment at random (with replacement: a given column in the original dataset can appear several times in the artificial dataset);
2. Apply the tree building method to the new dataset;
3. repeat steps 1 and 2 some number of times, typically about 1000 times.

The **frequency of a phylogenetic feature** (a node of a branch) is taken to be a measure of the confidence we can have of this feature. More in details, each node of the original tree receives a measure of its presence in the artificial topologies. For branches in the original topology to be significant, the artificial data set should predict the same branches over a threshold (for example, $> 70\%$).



14.20 CONSENSUS TREE

We've seen that various techniques produce different trees for the same data. As a result, it is a common practice to apply several different tree building principles to a given data set and build the consensus of the obtained set of trees (the branch points that all or most of the trees have in common). The most common forms are:

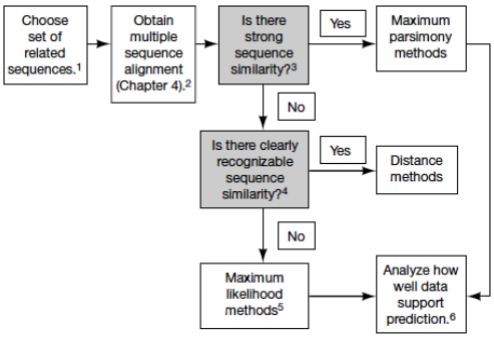
- **Strict consensus**: relies only on what is in the intersection of *all* the trees.
- **Majority consensus**: the criterion is more relaxed and includes what is present in more than 50% of the trees (a threshold is used). The edges are usually labelled with the percentage of the trees in which they are present.

For example, suppose we have the following different trees, each obtained with a different technique; then, depending on the type of consensus, we get different results:

In fact, ABC are all related by a common ancestor, and DEF also. In the majority rule consensus we obtain a different result since we consider the most common structures common of the trees.

A workflow to choose the tree buildin method:

It says that it depends on the info on the species you have.


Choose set of related sequences.[1]
Obtain multiple sequence alignment (Chapter 4).[2]
Is there strong sequence similarity?[3]
Yes
Maximum parsimony methods
No
Is there clearly recognizable sequence similarity?[4]
Yes
Distance methods
No
Maximum likelihood methods[5]
Analyze how well data support prediction.[6]

A

LANGUAGES AND GRAMMARS

Suppose we have the phrase "*colourless green ideas sleep furiously*": despite being nonsensical, it is a grammatically correct sentence in English. Chomsky was interested in how a brain or computer program could algorithmically determine whether a sentence was grammatical or not, and constructed "grammars" to enumerate the infinite number of sentences that belong to a language. Given a language and the corresponding grammar it is impossible to generate all the possible sequences; we can, however, decide whether a *certain* sequence belongs to that language or not: it all boils down to how well the grammar models the constraints on the language (i.e. how many grammatical sentences there are that the grammar fails to generate, and how many ungrammatical sentences the grammar generates erroneously).

A.1 GRAMMARS AND PARSINGS

A grammar – or *transformational* grammar – consists of:

- A set of symbols, either *terminal* or *nonterminal*. Nonterminal symbols are abstract symbols, meaning they don't appear in the actual sequence; terminal symbols are instead those of the sequence.
- A set of *production rules* (also called *rewriting rules*) in the form of $\alpha \to \beta$, where α and β are both strings of (one or more) symbols. If several rules have the same symbol on the left, the resulting productions can be grouped into a single production rule using the symbol |, or.

The easiest way to understand how a grammar works is by example. Suppose we have an alphabet $\{a, b\}$ and the production rule $S \to aS|bS|\epsilon$. To generate a string of as and bs such as abb we perform a *derivation* from the grammar such as the following:

$$S \to aS \to abS \to abbS \to abb$$

When grammars are used for a sequence analysis problem, we usually already have a sequence in mind; the question is whether the sequence "matches" (can be generated by) a certain grammar: we need to **work backwards** to determine whether a derivation exists for the string. Finding a valid derivation for a given sequence is called ***parsing*** while the derivation is also called a *parse* of the sequence.

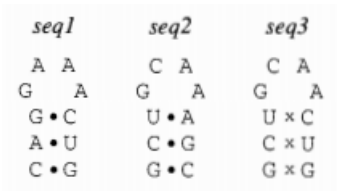
A.2 CHOMSKY'S HIERARCHY

In 1959 Chomsky dined a hierarchy of **four different types of grammars**. In the following pages we will focus on type-2 and type-3 grammars, which can be defined as follows:

- **Type-2: context-free grammars**. Generate context-free languages. Only production rules of the form $W \to \alpha W$ or $W \to \alpha$ are allowed. Typical of palindrome languages and especially **useful when dealing with RNA sequences**, these grammars permit additional rules that alow the grammar to create nested, long-distance pairwise correlations between terminal symbols (since the right side of any production rule can be a combination of terminal and nonterminal symbols). Note that context-free grammars can be **ambiguous**: sequences can have **more than one possible derivation**.
- **Type-3: regular grammars**. Generate regular languages, where strings are produced from left to right. Only production rules with a single nonterminal symbols on the left (i.e. $W \to \alpha W|\alpha$) are allowed.

A.2.1 *A context-free grammar or an RNA stem loop*

To understand how context-free grammar can be used to analyze molecule sequences, consider the following example. We have three sequences of amino acids: $seq1$ and $seq2$ fold into the same RNA secondary structure while $seq3$ cannot fold into a similar structure.



These differences can be recognized and described by a context-free grammar:

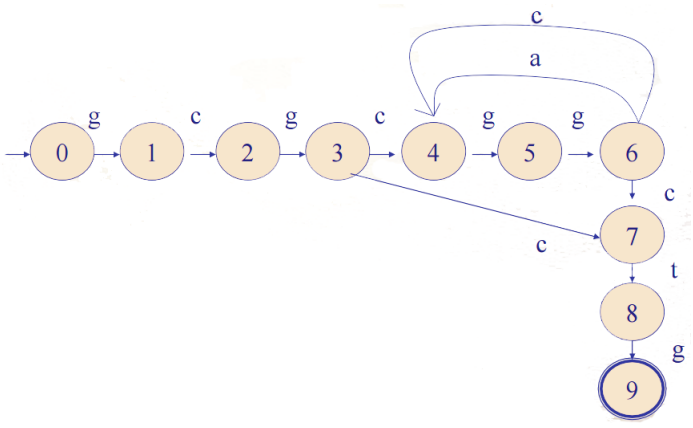
```
S -> aWu | cWg | gWc | uWa,   W -> aVu | cVg | gVc | uVa,
V -> aTu | cTg | gTc | uTa,   T -> gaaa | gcaa
```

A.3 AUTOMATA

Each grammar has a corresponding abstract computational device called *automaton* – parsers that accept or reject a given sequence. Regular grammars produce ***finite automata*** – simple automata reading one symbol at the time from an input string. If the symbol is accepted the automaton enters a new state, otherwise the automaton rejects the string. Automata are build to give a more concrete idea of how we might recognise a sequence using a formal grammar. Interestingly, we can see an application of finite automata on molecule biology straight away.

A.3.1 FMR − 1 *triplet repeat region*

The human genetic sequence `FMR-1` in the X chromosome contains the pattern CGG (with an accepted variation, AGG) which is repeated a number of times. The number of triplets is highly variable between individuals, and a high number of triplets is associated with *fragile X syndrome*, a genetic disease. The occurrence of the triplet can be represented as follows:



As we can see, the string we're analyzing starts with `gc` repeated a certain number of times and ends with `ctg`:

```
gcg ( cgg + agg )* ctg
```

A.3.2 *Regular expressions*

Multiple alignments of fragments of sequences (with similar functions) can be expressed as *regular expressions*. There are databases containing such fragments – PROSITE (1997) for example, where every entry is a relevant pattern shared by a group of proteins. An example of common **pattern RNP-1** shared by several proteins is the following:

```
RU1A_HUMAN: SRSLKM*RGQAFVIF*KEVSSAT
SXLF_DROME: KLTGRP*RGVAFVRY*NKREEAQ
ROC_HUMAN:  VGCSVH*KGFAFVQY*VNERNAR
ELAV_DROME: GNDTQT*KGVGFIRF*DKREEAT
```

Despite te different contexts, the pattern can be described with this ***regular expression***:

[RK]-G-{EDRKHPCG}-[AGSCI]-[FY]-[LIVA]-x-[FYM]

In fact, we have:

- Position 1: either `R` or `K`, represented with [];
- Position 2: always `G`;
- Position 3; any other amino acid except {EDRKHPCG};
- Position 4: one of the following symbols: `AGSCI`;
- Position 5: one of `FY`;
- Position 6: one of `LIVA`;
- Position 7: x, representing *any* amino acid;
- Position 8: one of `FYM`.

A.4 CHOMSKY NORMAL FORM

We will see in the following pages that the most popular parsing algorithm – the Cocke-Younger-Kasami algorithm – works exclusively on context-free grammars given in Chomsky normal form (CNF). In other words, we need context-free grammars where every production rule has the following structure:

- $A \to BC$

- $A \to a$
- $S \to \epsilon$

In other words, on the left side we only have a single nonterminal symbol; on the right side, instead, we have either two nonterminal symbols or a terminal symbol. Note that the starting point of the grammar is always the empty string, as stated above. **Any context-free grammar can be transformed into a Chomsky normal form grammar** and viceversa. Furthermore, we will also see that the algorithms we will study don't allow ϵ production, meaning that the grammar we want to work on has only production of the first and second kind.

A.5 CONVERT A CONTEXT-FREE GRAMMAR INTO A CHOMSKY NORMAL FORM GRAMMAR

To convert, we have the following steps:

1. **Starting point**. Create a new start symbol linking to the original one.

   $$S_0 \to S$$

2. **Terminal *or* nonterminal**. If a rule contains **both terminals and nonterminals** on the right, split it into several other rules with only terminals or only nonterminal symbols. For example, suppose we have $A \to XYaZ$. Since we have both terminal and nonterminal symbols here, we create a new rule $N \to a$ and modify the previous rule as follows:

   $$A \to XYNZ$$

3. **Only two nonterminal**. If a rule contains more than two nonterminal symbols, split it into rules with up to two nonterminals. For example, if we have $A \to BCDE$ we must split it into $A \to BC$ and $A_1 \to DE$.

4. **No empty production**. Delete any rule such as $A \to \epsilon$ (where A is **not the grammar's starting symbol**). To do so we must first determine which rules eventually derive ϵ (*nullable rules*). We follow these rules:

   - if $A \to \epsilon$ exists, then A is nullable;

- if $A \to X_1..X_n$ exists and each X_i is nullable, then A is nullable too.

For example, suppose we have this intermediate grammar:

```
S0 -> AbB | C
B -> AA | AC
C -> b | c
A -> a | ϵ
```

Since A is nullable then B is also nullable. Once A is deleted we modify the grammar as follows:

```
S0 -> AbB | Ab | bB | b | C
B -> AA | A | AC | C
C -> b | c
A -> a
```

5. **Only two nonterminals, reprise**. If a rule has a single nonterminal symbol on the right, delete it. To remove them, suppose we have $A \to B$; then we need to eliminate id and $\forall B \to U$ add $A \to U$ to the grammar.

With these modifications we obtain a grammar with more productions, but it will be binary.