Bioinformatics notes

Indice

Biological background **1**

DNA and RNA . . . 1

Proteins . . . 2

Protein synthesis . . . 2

Viruses . . . 2

Cellular signaling and metabolic processes . . . 2

Sanger Sequencing (low throughput) . . . 3

Illumina Sequencing (high throughput) . . . 3

Nanopore Sequencing (long read) . . . 3

Single and paired end sequencing . . . 3

Genome assembly **3**

Basics . . . 3

Modeling sequence data . . . 3

k-mers and genome length . . . 4

Contigs . . . 5

Graph-based genome assembly . . . 6

Hamiltonian path based approach . . . 6

De Brujin graph based approach . . . 6

Existence of Eulerian cycles and paths . . . 6

Finding Eulerian paths . . . 6

Practical problems . . . 7

Sequence alignment **7**

Strings and alphabets . . . 7

Comparisons of biological sequences . . . 7

Global sequence alignment . . . 7

Local sequence alignment . . . 10

Read mapping **11**

Tries . . . 11

Suffix tries and trees . . . 12

Suffix arrays . . . 13

Burrows Wheeler Transform . . . 13

FM index . . . 14

Biological background

DNA and RNA

DNA (deoxyribonuclic acid) is composed of 2 strands intertwined with each other forming a double helix. Each strand has a sugar-phosphate backbone and is a polymer of groups called nucletides. A nucleotide is made of a phosphate group, a pentose sugar (Deoxyribose or Ribose) and a base. Each nucleotide is linked to the previous by a covalent bond between the phosphate group on the 5' carbon and the hydroxyl group on the 3' carbon of the next nucleotide. There are 4 types of bases: Adenine, Cytosine, Guanine, Thymine. Each base can have 1 ring (pyrimidine) or 2 rings (purines). The two strands are held together by hydrogen bonds between complementary bases: A-T, G-C.

RNA differs from DNA by being single stranded and having Ribose as opposed to Deoxyribose in its nucleotides. Moreover in RNA Thymine is replaced with Uracil. Being single stranded doesn't imply that base pairing doesn't happen: bases can pair intra-molecularly and generate complex shapes.

Each strand has two ends: a 3' end (an hydroxyl group) and 5' end (a phosphate group). The two strands run in opposite directions and are complementary. This means that we can reconstruct the other strand by knowing one of the strands. Each strand is read from the 5' end to the 3'.

Genetic material is compacted into chromosomes. Bacteria don't have chromosomes, they have only one circular DNA molecule. Each species can have a different numbers of chromosomes. Humans have 46 chromosomes.

Every living organism is encoded by one or more DNA molecules that are continuously read and interpreted by each cell. The central dogma (now proven wrong but generally it works) mandates that information that determines a protein's structure contained in DNA is transcribed to RNA and the RNA is translated into proteins.

Proteins

Proteins are chains of amino acids (polypeptides). An amino acid is composed of an amino group, an R group and a carboxyl group. Amino acids are linked together by the peptide bond between the hydrogen of carboxyl group and the amino group. Each protein start with the amino group and ends with and the carboxyl group.

Proteins tend to form different structures classified in primary (the amino acid chain), secondary (alpha helix and beta sheet), tertiary (three dimensional structure) and quaternary structures.

Proteins are involved in most tasks essential for life: structural, receptors, signalers enzymes and transcription factors.

Protein synthesis

Synthesis starts with transcription: one strand of DNA is copied into a reverse complementary RNA molecule (in prokaryotes mRNA, in eukaryotes pre-mRNA). DNA is transcribed by an enzyme: RNA polymerase. Transcription is aided by transcription factors bound on the promoter of the gene (transcription start site) or to a distante enhancer. In eukaryotes, the transcribed RNA is often spliced, before it is exported from the nucleus to the cytoplasm: introns are spliced out while exons remain to form mature mRNA. A 5'-cap and e 3' poly-A tail are often added to stabilize the molecule.

mRNA transcripts are translated into proteins by ribosomes. Each amino acids corresponds to triplet of bases (codon) and are transported by tRNA that exposes the complementary triplet of bases of te codon. The first codon is always AUG, while coding stops with either UAA, UAG or UGA.

Viruses

Viruses are organisms at the "edge of life". They need to exploit the host cell for replication. The genetic material may be DNA or RNA, single stranded or double stranded, linear or circular. Some viruses integrate genetic material into the host genome.

The virus envelope/capsule may contain additional viral proteins.

Cellular signaling and metabolic processes

An environmental signal binds to a receptor on the cell's surface. The receptor then stimulates a behaviour in the cell by the synthesis of some proteins.

We can divide metabolic processes into two categories:

1. Catabolic: breaking down of components;
2. Anabolic: synthesis of new components.

We will avoid specifics as it can get absurdly complicated.

Sanger Sequencing (low throughput)

Developed by Fred Sanger, it's a really precise method with low throughput for sequencing DNA. It is based on sequencing by synthesis: start from single stranded DNA, reconstruct the reverse strand and check which bases you have to add to reconstruct it. The ingredients needed are:

1. single stranded DNA template
2. DNA primer
3. DNA polymerase
4. normal deoxynuclotide-triphosphate (dNTPs)
5. chain terminating dideoxynuclotide-triphosphate (ddNTPs) marked with fluorescence.

We can make 4 experiments, one for each base: we add ddNTPs to the mixture and we observe how many sequences end with that specific nucleotide. We can then order strands by length (filtering) and observe the ending strand (separating by terminating base). At the end we can read in reverse the experiment results and generate the sequence.

Illumina Sequencing (high throughput)

Illumina sequencing is most suited to shorter sequences and is divided in four basic steps:

1. DNA & library preparation: make random DNA fragments and append adapters
    - Fragment the genomic DNA
    - Repair ends and add an A overhang
    - Ligate adapters to DNA fragments
    - Select ligated DNA
2. Chip/flowcell preparation: attach fragments to surface and amplify
3. Sequence: massively parallel DNA sequencing
4. Analyze.

Nanopore Sequencing (long read)

Nanopore sequencing is a new method for reading longer sequences. It works by passing a strand of DNA through a microscopic pore one base at a time. For each pair you apply a voltage and observe the current through te nanopore. Each base has a different current. So by following current changes we can determine the sequence.

Single and paired end sequencing

The DNA fragments inserted between the adapters are usually longer than the maximum read sequence length. We can either sequence only one end (single-end) of the fragment or both ends (paired-end). Paired-end sequencing can detect errors and mutations due to the fact that it reads both strands.

Genome assembly

Basics

Genome assembly reconstructs the genome from short sequencing reads, while read mapping "maps" the reads to a known reference genome. Since sequencing is now cheap, can we sequence and *de novo* assemble a large genome with the short reads of NGS protocols (50-250 bp)?

> The process of puzzling together a complete genome sequence of an organism for which "shotgun sequencing" has been performed is referred to as genome assembly.

There are two major classes of assembly algorithm: overlap-layout consensus (OLC) and de Brujin graphs (DBG). OLC was widely used back when sequencing was performed by low-throughput long-read methods; DBG-based methods have dominated since the introduction of NGS.

Modeling sequence data

Let us consider an idealized genome that represents a long random sequence of four bases that does not contain repeats or other complex structure. Consider a simple and error free sequencing strategy: single-end, whole-genome sequencing

(WGS). That is, we sample equal length fragments (the reads) with starting points randomly distributed across the genome. Thus, our "shotgun sequencing" can be compared to a process that samples bases from all genome positions at random. The chance that any particular base is sampled in a single read is very low. However we perform the sampling process a very large number of times.

A nice model for this situation is the Poisson distribution. The Poisson distribution expresses the probability of a given number of events occurring in a fixed interval of time/space if these events are independent and identically distributed.

$$f(k, \lambda) = P(X = k) = \frac{e^{-\lambda}\lambda^k}{k!}$$

Where k refers to the number of reads that overlap a certain genomic position (coverage) and λ is the mean sequencing depth, the average number of reads covering each base in the genome.

The number of sequenced bases is $n_b = N \cdot L$ (number of reads times the read length). Similarly the average coverage depth per base is $\lambda = n_b/G$ with G as the genome size.

k-mers and genome length

Lets introduce the concept of k-mers: a k-mer is a short subsequence of k nucleotides. In general there are $L - k + 1$ k-mers in a sequence of length L ($L > k$). To know the total number of k-mers in our WGS data, since we have N reads, we can calculate it as such: $n_k = N \cdot (L - k + 1)$. The coverage depth for k-mers is then $d_k = \frac{n_k}{G}$. The ratio between average coverage depths is then:

$$\frac{\lambda}{d_k} = \frac{n_b/G}{n_k/G} = \frac{L}{L - k + 1}$$

Using k-mers we can estimate the total genome size. The number of k-mers in the WGS reads can be directly counted. So the mean coverage depth of k-mers can be estimated from the peak value of the empirical k-mer coverage depth distribution curve.

But how can we find the k-mer coverage depth distribution? Since we can assume k-mers to be from unique genome positions in most cases, we can the interpret the count (number of matches) of a specific k-mer as the coverage depth at that position. We the plot the distribution over all k-mers and identify the peak as the mean coverage depth.

With this data in hand we can estimate the genome size to be:

$$G = \frac{n_k}{d_k}$$

And we can estimate the average base coverage by:

$$\lambda = \frac{L}{L - k + 1} \cdot d_k$$

So if we want to estimate the mean read required such that at least 99% of the genome is covered once we have

$$P(X > 0) = 1 - e^{-\lambda} = 0.99$$
$$\lambda = 4.605$$

Thus we need to sequence to an average depth of at least 4.6 to get at lest 99% of the genome covered at least once.

Contigs

Now let us consider "contigs": combinations of overlapping reads that represent continuous sequence. The contigs are assumed to be the best possible representation of the original DNA sequence. However the actual locations of the contigs and their orientation to one another are unknown to us.

The initial steps of genome assembly are basically an attempt to find contigs. The result of assembly is a set of contigs with gaps.

In a genome of length G, a read of length L can start anywhere (except at the ends of the chromosomes, we will ignore this). An approximation for the expected number of reads starting at a specific genomic location i is $N_i = N/G$. The probability of reads to start in a given interval is (assuming Poisson distribution):

$$P(X > 0) = 1 - P(X = 0) = 1 - e^{-\lambda}$$

Now lets consider a nucleotide at position i. This nucleotide is in a gap between contigs if no read starts in the interval $[i - (L - 1), i] = [i, i - L + 1]$. This interval has length L and thus the probability that no read starts in it is $e^{-\lambda}$. We can estimate the number of nucleotides in gaps across the entire assembly as $G \cdot e^{-\lambda}$. Correspondingly, the number of nucleotides included in contigs is $G \cdot (1 - e^{-\lambda})$. This means that the fraction of genome which will be covered by contigs depends on the "sequencing depth" (i.e. how much sequence data we produce and thus what per-base coverage we have).

How many contigs can we obtain? Each contig has a unique rightmost read R. The probability that a given read is the rightmost read is the same as the probability that no other read start within that read: if the read starts at position i, this is the probability that no read start within $[i, i + L - 1]$, which is $e^{-\lambda}$. The number of contigs must be equal to the number of rightmost reads. Since there are a total of N reads, each of which with probability $e^{-\lambda}$ of being and R read, the expected number of contigs is:

$$N \cdot e^{-\lambda}$$

And the expected number of reads per contig is:

$$\frac{N}{Ne^{-\lambda}} = \frac{1}{e^{-\lambda}}$$

The expected size of a contig can be calculated to be:

$$\frac{\left(1 - e^{-\lambda}\right) G}{Ne^{-\lambda}}$$

How much of an overlap is required to connect two reads in a contig? Let θ refer to the minimum portion of L that is required to detect an overlap. This means that we will combine a group of reads to a contig if they are connected by overlaps of length grater than θL.

Let us calculate the number of expected contigs taking into account θ. We now need to calculate the probability that zero reads start in the leftmost portion of the interval $[i, i + L - 1]$ ($(1 - \theta)L$). Similarly to what we did before, we can calculate the expected number of reads that start in $(1 - \theta)L$ to be $(1 - \theta)\lambda = (1 - \theta)LN/G$. The expected number of contigs is then N times the probability that there are no reads starting in $(1 - \theta)L$:

$$\#contigs = Ne^{-(1-\theta)LN/G}$$

This means that the expected number of contigs depends on the average coverage and the degree of overlap.

The N50 measure is used to estimate the quality of a genome assembly. To calculate the N50 index we:

1. Arrange the contigs from largest to smallest;
2. Find the position where the contigs cover 50% of the total genome size;
3. The length of the contig at this position is defined as the N50.

The longer the better the assembly.

Graph-based genome assembly

Our goal is to find an algorithm that will allow us to take a collection of short NGS sequence reads and to output a longer string representing the genome.

Hamiltonian path based approach

Lets start with a simple genome we sequenced. Lets say we have a function called `composition_k()` that takes a DNA sequence and returns a set of all k-mers contained in it. We do not know the original order of the k-mers in the genome.

The challenge is reconstructing the original string with an unordered set of k-mers. Let us now put each of the k-mers into the nodes of a graph. To connect the nodes we can search for overlaps between k-mers. We connect the i-th k-mer with the j-th k-mer if `suffix(k-mer_i) = prefix(k-mer_j)`. We can extract the original string by walking a path that visits each node exactly once.

A Hamiltonian path is a path in an undirected or directed graph that visits each vertex exactly once. If this path is a cycle, it is called a Hamiltonian cycle.

Determining whether Hamiltonian graphs/cycles exist in a given graph is a NP-complete problem.

De Brujin graph based approach

Now let us consider the same simple genome and k-mers. However, instead of labelling the nodes with k-mers, let us label the edges. We will label the nodes with the $(k-1)$-mers suffixes and prefixes of the edge. Let us now merge identically labeled nodes in this graph whilst retaining the edges. The resulting graph is called a de Brujin graph. To reconstruct the original sequence we need to find a path that visits all edges once.

An Eulerian path is a path that visits each edge exactly once.

Determining whether Eulerian path are present in a given graph can be solve with efficient algorithms.

Some post-processing we can do to the graph is "compress" it: we can collapse unambiguous edges.

Existence of Eulerian cycles and paths

Do we have a way to check if a graph can have an Eulerian path or cycle? Yes, we need to look at the degree of each node: the degree of a node is the number of edges in ad out of it. The two conditions are:

1. When the degree of all vertices is even, the graph is traversable (there is a Eulerian cycle). For directed graphs, we need all vertices to have the same number of incoming and outgoing edges.
2. When there are exactly two nodes of odd degree, there is a Eulerian path starting at one of the odd vertices. In directed graphs we need to have only two vertices that have respectively:

$$degree_{in} - degree_{out} = 1$$
$$degree_{in} - degree_{out} = -1$$

   All other vertices will need to have the same number of incoming and outgoing vertices.

Finding Eulerian paths

For undirected graphs, we start at one of the two veritces with odd degree. For directed graphs we start at the vertex with one more outgoing edge and we will end at the ne with one more incoming edge. If the graph has a Eulerian cycle, you can find it with the same algorithm.

We will use Hierholzer's algorithm:

```
tpath <= Stack() # temporary
epath <= Stack() # eulerian path starting at epath.top() and ending at
                 # epath.bottom()
v <= chooseSuitableStart()
tpath.push(v)
```

```
do
  u <= tpath.top()
  if allOutgoingEdgesVisitedFrom(u) then
    u <= tpath.pop()
    epath.push(u)
  else
    e <= selectRandomOutgoingFrom(u)
    tpath.push(e.end)
    u.outgoingEdges.remove(e);
while (!tpath.empty())
```

Practical problems

1. How do we choose a value for k in real-life? The answer is "big enough": the k-mers should be big enough to be unique in the genome. However, memory usage for computation grows $\mathcal{O}(nk)$: for about 3×10^0 nucleotides with $k = 27$ requires about 20 GB of memory to store the nodes alone! Plus, we are completely ignoring the problem of repeating patterns/structures.
2. Sequencing errors: there is a high number of erroneous k-mers due to sequencing errors, with only few copies each. Therefore we need to remove low-frequency k-mers in preprocessing.
3. Repetitive regions.
4. Strand ambiguity: each read, and hence the k-mers, could be from either the forward or the reverse strand.

Sequence alignment

Information is encoded in living cells mostly in the form of "sequences": DNA/RNA sequences and protein sequences. Biological function largely depends on the 3D structure of large molecules, but many characteristics and functions can be inferred from the 1D structure.

Strings and alphabets

Lets recall the notions of alphabet and string:

- An alphabet is a finite set of symbols Σ
- A string over an alphabet Σ is any finite sequence of symbols from Σ. The empty string is denoted as ϵ
- A sequence is a set of elements ordered so that they can be labeled/enumerated with positive integers

The DNA/RNA alphabet has 4 letters (the bases), the protein alphabet has a 20-letter alphabet, one letter for each amino acid.

The set of all strings of any length is $\Sigma^\star$ (the Kleene closure). Strings can be concatenated, reversed etc. A substring s of a string t is a string so that $t = usv$ such that u and v exists (also empty). A string on length n has $1 + \frac{n(n+1)}{2}$. A string $s = uv$ is said to be a rotation of t if $t = vu$. The number of rotation of a string is equal to its length.

Comparisons of biological sequences

We need to define suitable measures of distance or similarity, as well as efficient algorithms to compute them. The measures should reflect as much as possible the different ways in which the sequence can be similar or different, and the processes or factors underlying the similarities and differences: evolution of sequences, errors in data production (sequencing data), errors made by the cells during DNA replication.

Global sequence alignment

Hamming distance The hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different. It is not a good measure for dissimilarity, but we can formalize "substitutions" in otherwise identical sequences.

Edit distance The edit distance is a way of quantifying how dissimilar two strings of different lengths are to one another by counting the minimum number of operations required to transform one string into the other. Different definitions use differente string operations, we will use the Levenshtein distance which permits these operations: removal, insertion and substitution of a character in the string. We can formally define the edit distance (from now on we use edit distance interchangeably with Levenshtein distance) with recursion as:

$$lev(a,b) = \begin{cases} |a| & if |b| = 0, \\ |b| & if |a| = 0, \\ lev(tail(a), tail(b)) & if a[0] = b[0], \\ 1 + min \begin{cases} lev(tail(a), b) \\ lev(a, tail(b)) \\ lev(tail(a), tail(b)) \end{cases} & otherwirse \end{cases}$$

Where:

- $tail(x)$ is the string x without its first character
- $x[n]$ is the n-th character of the string x
- In the minimum term:
  - First element corresponds to a deletion/removal of a character from a
  - Second element corresponds to an insertion of a character to a
  - Third corresponds to a replacement/substitution.

From the mathematical definition, it is evident that calculating the edit distance implies solving a minimization problem. In theory, the number of solutions is infinite, however in practice we would not need to test all of them because any solution with more than $max(|a|, |b|)$ operation can be excluded, se we could discard partial solutions as soon as we exceed this number of operations.

Can we exploit the edit distance to fully "align" two sequences with respect to their similarities and differences? Let's use dynamic programming and find an algorithm to compute the edit distance between two strings.

1. Denote as $E(a, b)$ the edit distance between two string a and b
2. For every string a, we have $E(a, \epsilon) = |a|$
3. Let $a = a_1 a_2 \dots a_n$ be a string of length $|a| = n$ and $b = b_1 b_2 \dots b_m$ a string of length $|b| = m$. We denote the prefixes of length i of a and b respectively $a(i)$ and $b(j)$
4. To compute the edit distance $E(a, b)$ between a and b, let's first assume that for given i and j we (somehow) computed:
   - $E(a(i-1), b(j-1))$
   - $E(a(i), b(j-1))$
   - $E(a(i-1), b(j))$
5. It can be proven that the following is true:

$$E(a(i), b(j)) = min \begin{cases} 1 + E(a(i), b(j-1)) \\ 1 + E(a(i-1), b(j)) \\ \begin{cases} E(a(i-1), b(j-1)) & \text{if } a_i = b \\ 1 + E(a(i-1), b(j-1)) & \text{if } a_i \neq b \end{cases} \end{cases}$$

This is a recursive definition: we can initially set $i = n$ and $j = m$. Thus, to compute $E(a, b)$, we can start with:

$$E(a(1), \epsilon) = 1 \quad E(\epsilon, b(1)) = 1 \quad E(a(1), b(1)) = \begin{cases} 0 & \text{if } a_i = b \\ 1 & \text{if } a_i \neq b \end{cases}$$

We can then prepare a matrix with $|a| + 1$ rows and $|b| + 1$ columns, each row/column labeled with on character from a and b respectively. The first row/column are labeled with a gap (-). We can use this table, for each i and j, the edit distance for the prefixes $a(i)$ and $b(j)$. Our solution will be in the $(n; m)$ cell.

To compute what changes are optimal, we need to backtrack from the previous computation. We start, from (n, m) and we move to the adjacent cells which has the minimum number of steps. Moving diagonally with no increase of cost means we have a match, moving diagonally with an increased score means a mismatch/substitution and lastly moving horizontally/vertically means we have a gap.

For a more efficient backtracking, we keep track of what choice we mad every time we fill a cell in the starting computation. Every cell will now have a number of pointers to the previous cells.

Each step of the computation takes constant time. In the worst case there is no overlap between the two sequences, meaning the alignment will have a length of $|a| + |b| = n + m$ symbols. Thus the complexity of backtracking is $\mathcal{O}(n + m)$ and the overall complexity is dominated by the construction of the table: $\mathcal{O}(nm)$.

It is possible that for a single edit distance, more backtracking options could be available. From an algorithmic point of view, every backtracking valid. However, the best solutions depends of the likeliness of the probability of the modification in our problem/model! Thus, we need to add some weighting.

Weighted edit distance Let's reformulate the minimization problem: given two strings a and b on an alphabet Σ, the edit distance $E(a, b)$ can be also defined as the minimum-weight series of edit operations that transforms a into b.

$$E(a(i), b(j)) = min \begin{cases} w_{id} + E(a(i), b(j-1)) \\ w_{id} + E(a(i-1), b(j)) \\ \begin{cases} E(a(i-1), b(j-1)) & \text{if } a_i = b \\ w(a_i, b_j) + E(a(i-1), b(j-1)) & \text{if } a_i \neq b \end{cases} \end{cases}$$

This definition requires the definition of a weight matrix that defines the likeliness of each change for each letter.

Sequence similarity Until now, we calculated the distance between two strings. Now, we need to define a measure and a way to calculate similarity.

One simple way, is to use a "percent identity": after alignment, we determine the percentage of the alignment which does not indicate an edit operation. The percent identity may differ between different alignments even when they have the same edit distance.

One better method, is finding the "longest common subsequence". For subsequence we mean a string derived by deleting some, all or no characters from the original string. We can see each subsequence as the result of applying a binary mask to a string. This means that we have a total of 2^n subsequences. An algorithm for solving the LCS problem is:

1. Let a and b be two strings with $|a| = n$ and $|b| = m$.
2. Let $LCS(a, b)$ be the length of the longest common subsequence.
3. We can compute $LCS(a, b)$ using:

$$LCS(a(i), b(j)) = max \begin{cases} LCS(a(i), b(j-1)) \\ LCS(a(i-1), b(j)) \\ \begin{cases} 1 + LCS(a(i-1), b(j-1)) & \text{if } a_i = b \\ LCS(a(i-1), b(j-1)) & \text{if } a_i \neq b \end{cases} \end{cases}$$

   This is very similar to computing the edit distance. We setup the table similarly and then backtrack to reconstruct the longest substring: we find a character/symbol belonging to the solution every time we have a diagonal move with a score increased by 1.

Combining distance and similarity We have seen two symmetrical approaches for comparing sequences/strings. We can then define a measure for similarity such that "differences" have a negative effect" and conserved letters have a positive effect.

We can combine the distance measure with the similarity measure as follows:

$$E(a(i), b(j)) = max \begin{cases} w_d + E(a(i), b(j-1)) \\ w_d + E(a(i-1), b(j)) \\ \begin{cases} w_m + E(a(i-1), b(j-1)) & \text{if } a_i = b \\ w_s + E(a(i-1), b(j-1)) & \text{if } a_i \neq b \end{cases} \end{cases}$$

Where $w_d < 0$ is the gap penalty for insertions/deletions, $w_s < 0$ is the negative mismatch penalty for substitutions and $w_m > 0$ is the positive match score. Since this is also similar to the calculation of edit distance and LCS, similar methods apply:

1. Prepare a matrix with cells $(0,0) = 0$ and $(i,0) = i \cdot w_d$ and $(0,j) = j \cdot w_d$
2. Fill the cells as usual
3. Traceback and slignment

Knowing the weights used during the calculation, we can extract the final score from the alignment: each column of the alignment represents a match or an edit operation; associate each column with its score and sum them.

This algorithm is called the Needleman-Wunsch algorithm. This algorithm is used for (and also called) global sequence alignment.

Note about gap penalties We can have different models of gap penalties:

1. Linear gap penalty: the total gap penalty is linear to the number of gaps;
2. Constant gap penalty: the total gap penalty is constant;
3. Affine gap penalty: combines the two previous methods by defining:
   - Constant gap opening penalty: penalty for opening a gap
   - Linear gap extension penalty: penalty for extending the length of an existing gap by 1

   Usually we have a greater gap opening penalty than extension.

Simplifying By defining a substitution matrix θ, we can simplify our calculations. Given an alphabet Σ, the substitution matrix is a $|\Sigma| \times |\Sigma|$ matrix which gives for every pair of symbols in the alphabet the weight of the substitution in the alignment. The matrix is symmetrical.

$$E(a(i), b(j)) = max \begin{cases} w_d + E(a(i), b(j-1)) \\ w_d + E(a(i-1), b(j)) \\ \sigma(a_j, b_j) + E(a(i-1), b(j-1)) \end{cases}$$

Uses of global alignment Global sequence alignment is the de facto standard for comparing biological sequences related by evolution. However, differente types of macromolecules evolve at different speeds: protein-coding genes evolve faster than proteins. Also, within a genome some regions tendo to be conserved by evolution. Thus in many cases an evolutionary relationship can be identified for only part of the sequences but not globally.

Local sequence alignment

Local sequence alignment identifies similar region within two sequences, without requiring full alignment.

Let a, b be two string over Σ, let $\sigma(s_i, s_j)$ be a substitution matrix for Σ and let w_d the gap penalty. Our problem is the following: "Find two substrings, one from a and one from b that produce the alignment with the maximum possible score".

If two sequences are not identical, there must be operations with negative score. This brings two special cases:

1. Since negative scores are possible, if there is no pair of substrings with an alignment score greater than 0, then the best solutions is ϵ

2. If we have no negative scores in the global alignment then the best solution would be the full global alignment.

Since two substrings with maximum alignment need not be of the same length, we need to compare and align all possible pairs of substrings from a and b. This means that naively enumerating all possible substrings would require to compute $\mathcal{O}(n^2m^2)$.

Smith-Waterman algorithm This algorithm guarantees to find the best solution. It is based on the dynamic programming matrix we used before: it already contains scores which potentially consider all possible substring pairs.

How can we identify which possible sub-path has the highest increase in score? We can let cell (i, j) contain the maximum alignment score of all substring pairs which end at position i and j respectively not from the beginning, but from an arbitrary place (a, b). We can achieve this by adding a new choice for each step:

$$M(a(i), b(j)) = max \begin{cases} 0 \\ w_d + M(a(i), b(j-1)) \\ w_d + M(a(i-1), b(j)) \\ \sigma(a_j, b_j) + M(a(i-1), b(j-1)) \end{cases}$$

Computing a value for a cell is equivalent to extending an alignment. But if all possible extensions lead to a negative score, then it's better to "reset" everything and start a new alignment. This implies that when the choice is 0, no traceback pointer will be associated with the cell.

To apply the algorithm we do as before: we initialize the matrix with zeroes. We put a traceback pointer only when score is greater than 0; one the table is complete, we start in the cell with the highest positive score in the entire matrix and we traceback.

This algorithm can generate multiple optimal solution. We can report only one of them or all of them. Once we have found the best local alignment, we can iterate by finding the second "best" score in cells which haven't been used for previous alignments and trace back to find the second best.

Read mapping

Considering DNA sequencing data, reference-based assembly follows the goal of finding the differences between an individual's genome and the reference genome for corresponding species, rather than characterizing the genome of that species in the first place. This requires reads to be aligned/mapped to the reference genome.

Why not use de novo assembly? De novo assembly algorithms are demanding and error-prone. We would like to leverage our knowledge of the human genome.

One approach is using local-sequence alignment. We would have $\mathcal{O}(LG)$ complexity where L is the read length and G the number of bases. The problem is that we'd need to do this for each read, that requires a huge matrix due to the size of the human genome, Plus for NGS this means hundreds of millions of times.

Faster algorithms are based on preprocessing of the text to build a substring index. Using the resulting data, occurrences of a pattern can be found quickly.

A substring index is a data structure which gives substring search in a text or collection of text in sublinear time. String searching algorithms identify the positions where one or multiple strings are found as substrings of a larger string. In the following discussion of string searching we will assume that the search pattern matches perfectly to a substring int the full text

Tries

A trie is a multi-way tree structure useful for storing strings over an alphabet. A trie is defined as the smallest tree over an alphabet such that:

1. Each edge of the trie is labelled with one character $c \in \Sigma$
2. A node has at most one outgoing edge labelled for each $c \in \Sigma$
3. Each key is "spelled out" along some path starting at the root.

We can then construct a trie from the reads and “slide” the trie down the genome once.

We define \$ as a symbol that is lexicographically less than every other symbol and represents the end of a sequence. The \$ enforces a lexicographic rul that we know from dictionaries: “over” comes before “overture”. The terminator also ensures that no suffix will be considered as a prefix of anny other suffix.

Using a trie this way we now use $\mathcal{O}(L'G)$ for matching (L' is the maximum length of any read) and $\mathcal{O}(n_b)$ for trie construction (n_b is the total length of the reads).

Suffix tries and trees

In the previous “method” we made a trie out of the reads and slid this tries across our genome to search for matches.

Lets create now a trie where each path from the root to a leaf represents a suffix, and each suffix is represented as a path from the root to a leaf. Each substring is a prefix of some suffix of a string T. Thus to search for a substring, start at the root and follow the edges labeled with the characters of our substring. If at some point there is no outgoing edge for the next character of the substring, the our read is not a substring of T.

Using this data structure, we can also easily find:

1. how many times does S occur in T: S occurs the same number of time as the number of leaf nodes rooted at the ending node
2. what is the longest repeated substring S of T: the deepest node with 2 or more children.

An algorithm for constructing a suffix trie is:

```
T += $
root = {}
for i = 1 to i == length(T) do
  n = root
  for c in T[i:]
    if c notin n then
      n[c] = {}
    end if
    n = n[c]
  end for
end for
return root
```

Lets analyze the size complexity of the suffix tree. A suffix trie has depth equal to the length of the longest suffix plus 1 an breadth equal to the maximum number of suffixes. The worst case is $\mathcal{O}(m^2)$. This is too big to be practical.

We can optimize the suffix trie into a suffix tree. We can:

1. combine non-branching paths into a single edge with a string label
2. replace the string label with $\mathcal{O}(1)$ references to the original string

Doing things this way we have reduced the number of nodes to $\mathcal{O}(m)$. However the length of the edge labels is still $\mathcal{O}(m^2)$. Searching in a suffix tree also is reduced to $\mathcal{O}(n+k)$ where n is the length of the substring and k is the number of matches in the string.

We can optimize further: instead of saving the labels, we can simply store the offset length of the original labels in the original string. This makes the edges $\mathcal{O}(1)$. Additionally we can store the offset of the full suffix in the leaves.

It may be useful to build a combined suffix tree from more than one input string. Instead of using one termination character, we use k different termination characters, one for each input string. Each leaf will be labeled with two integers to identify the input string from which the suffix originates and the position of the suffix within that string, also each edge is additionally labeled with an integer to identify the input string. The generalized suffix tree provides the best solution to different problems related to string comparison: longest common substring problem, longest palindrome substring, longest reverse complementary pair of substring in a DNA sequence and others.

The problem with suffix tries is that although they have $\mathcal{O}(n)$ size complexity, they have a very high “hidden constant”, making them unpractical.

Suffix arrays

The suffix array, at least in its simplest incarnation, requires only 4 bytes per character of the input sequence. A naive implementation of the suffix array basically manipulates an array of pointers to the various suffixes.

Naively, to construct a suffix array we first identify all the suffixes and keep track of their indexes. We then sort lexicographically the string (remember, $ comes before any strings). Ordering highlights repeated suffixes.

Using suffix arrays efficiently solves the problem of finding the longest repeated suffix: we simply scan through the list to find neighbours with the longest common prefix.

The Manber-Myers algorithm is an algorithm that efficiently creates new suffix arrays. It divided into 2 steps:

1. First we sort the array using only the first character of the suffixes
2. Recursive phase: given an array of suffixes sorted on the first 2^{i-1}, we create an array of suffixes sorted on the first 2^i.

Burrows Wheeler Transform

The BWT applies a reversible transformation to a block of input text. Thew transformation does not itself compress the data, but reorders it to make it easy to compress with simple algorithms.

To create a Burrows Wheeler matrix we do:

1. form all rotations of the input text T
2. sort the rotated strings lexicographically

The BWT of T is simply the last column of our matrix. Since the BWT contains a lot of runs of the same identical characters, we can use run-length compression to shorten it.

We can write an algorithm to create $BWT(T)$ from the suffix array of the same string by noting that position i of the BWT corresponds to the character that, int the original string, is just left of the i-th suffix in the array. We can now construct the BWT as follows:

$$BWT(T)[i] = \begin{cases} T[SA[i]-1] & SA[i] > 0 \\ \$ & SA[i] = 0 \end{cases}$$

If we want to get the original string back, we would need to reverse the compression procedure and get the original string back from the BWT. The reversibility of the BWT depends on the LF mapping property.

> LF mapping property:
>
> For any character, the T-ranking of characters in the first (F) column of the BW matrix is in the same order as the same characters in the last (L) column.
>
> For T-ranking we mean the number of times that an identical character has preceeded it in T.

Additionally, we introduce the B-ranking, calculated in a similarly to the T-ranking: it is the number of times the same character has occurred in given column before. The B-ranking is like a cumulative count of the characters. Since it's just a relabeling of identical characters, the same character in the string has the same the same ranking in the F and L columns.

Observing the B-rankings of the LF columns, we can note that:

1. F is made of chunks of identical characters with ascending B-ranks
2. In L the B-ranks of any character are arranged in ascending order.

During the reversing process we will exploit the cumulative index property of the BW matrix: for each character in L, its index is equal to the sum of the max B-rank of previous characters plus one, and the character's own b-rank. So for example in $a_0r_0d_0\$_0r_1c_0a_1a_2a_3a_4b_0b_1$ we can calculate the index of c_0 to be $(0+1)_\$ + (5+1)_a + (1+1)_b + 0_c = 8$.

For reconstructing the original string, we will be working backwards. Starting from the BWT, we first construct the F column lexicographically sorting L. Then, starting at the first character of F:

1. Prepend that character to our result
2. We know that the last character that precedes our current L character is the one with the same index in L.

3. Using the cumulative index property we jump to the precedent in F.
4. We repeat from step 1 until the we hit \$.

FM index

The "Full-Text index in Minute space" index uses the BWT and some auxiliary data structure to generate a fast end efficient index for searching small patterns within a larger string T.

The main data structures of the FM index are the F and L columns of the BW matrix. The F column is represented as an array of integer containing the number of occurrences of that letter. The L column is also compressed.

A high-level overview of a possible substring search algorithm is the following:

1. Starting with the last character of our prefix P, we get the chunk of F corresponding to our current letter.
2. We look in L to find the chuck corresponding to the precedent of the current letter. Using the B-rank if the letters in we can find the position of this chunk in F
3. We repeat from 1 until we either: consume all the prefix or have a mismatch.

This high level overview glosses over 3 important issues:

1. How do we efficiently find the preceding character? In the worst case we would have to scan all the L column.
2. We do not want to store the B-ranks of the character, so we have to find a way of deducing it.
3. We still need a way of figuring out at what position the matches occur in T.

For the first two issues, we can construct a tally table consisting of the pre-calculated number of each specific character is L up to every row.

F	L	a	b	c	d	r
$\$_0$	a_0	1	0	0	0	0
a_0	r_0	1	0	0	0	1
a_1	d_0	1	0	0	1	1
a_2	$\$_0$	1	0	0	1	1
a_3	r_1	1	0	0	1	2
a_4	c_0	1	0	1	1	2
b_0	a_1	2	0	1	1	2
b_1	a_2	3	0	1	1	2
c_0	a_3	4	0	1	1	2
d_0	a_4	5	0	1	1	2
r_0	b_0	5	1	1	1	2
r_1	b_1	5	2	1	1	2

In general we can subtract one from the tally table to obtain the zero based B-rank. For example, let's say we search for *abra*; after we found the range of rows $[i; j]$ that start with a, we can simply look at the number of r at $i - 1$ and at j to know that preceding characters are r_0 and r_1.

A problem with the tally table is its space complexity: $\mathcal{O}(|T| \cdot |\Sigma|)$ integers. We can mitigate this by storing every k-th row. This means that for a hit on a non stored index, we need to start form the nearest "checkpoint" and count the occurrences of that letter from the checkpoint to the queried index.

For issue 3, we can exploit that a sorted suffix array is equivalent to the BWM. For instance, if we used our searching algorithm and landed on a character, we can lookup the corresponding index in the suffix array to get starting index in the original string. Like with the tally table, for space efficiency, we will store only each k-th *value in the original string* (not row of the suffix array). When we hit an unsaved row, we simply continue reconstructing the prefix until we hit a saved row. We then subtract from that prefix the amount of characters we "extended" our prefix.

> Note that the fact that we are storing each k-th value ensures that we need at most $k - 1$ hops to retrieve the index we are looking for.