1 Introduction to Genetics

1. Describe the structure of the *deoxyribonucleic acid* (DNA), and highlight the ways in which it differs from the *ribonucleic acid* (RNA). Distinguish the concepts of a gene and a genome with respect to DNA structure.

   DNA is a double helix structure: it consists of two sequences of bases (abbreviated `ATCG`) which are bound together in complimentary pairs: `A` binds with `T`; and `C` binds with `G`. A gene is a sequence of DNA which represents a protein which perform a particular function. The genome is the entirety of all the DNA in an organism. This consists of a set of chromosomes – each of which is composed of thousands of genes.

   Rather than having the bases `ATCG`, RNA has the bases `AUCG` where `A` binds with `U`. In the context of eukaryotic cells, DNA is found in the nucleus while RNA is found around the cell. There are 3 types of RNA. The most important of which is mRNA; which is read by ribosomes and used to guide them which proteins to synthesise.

2. Explain, with the aid of a diagram, the process of *gene expression* (synthesis of a protein based on the genetic information contained within DNA). Your answer should contain the following terms:

   - DNA
   - messenger RNA
   - codon
   - protein
   - transcription
   - translation

   We have DNA in the nucleus. It's transcribed into mRNA by RNA polymerase enzymes. This mRNA then leaves the nucleus. It is then found by some ribosomes. They then inspect the codons (sequences of 3 bases) and translate this into the corresponding proteins.

3. How are different genes delimited within the DNA molecule? Can you correlate this to a similar concept used within a programming language (covered within the Tripos)?

   Genes are preceded by a regulatory region. They have buffer space at the end to reduce the probability that during transcription, ribosomes will overread the gene and read into another gene. This is related to the memory representation of arrays in languages such as `C` when running with ASan. Arrays are started by a pointer and terminated by a null pointer. Furthermore, at the end of each array there is a set of uninitialised memory to act as a buffer to reduce the risk of accidental overflow.

4. How many different codons exist? How about different amino acids? Provide an *evolutionary* explanation for the discrepancy between your two answers.

   There are 64 different codons (4^3). There are 20 amino acids which are represented by codons (plus STOP). This gives redundancy and tries to reduce the probability that a protein will be badly mis-synthesised. Furthermore, most of the redundancy is in the third base in the codon: this is because the third base is the most likely to mutate. This happened because those animals which had fewer amino acids were less likely to suffer from frequent mis-synthesis and so their bodies worked better and they ended up with a higher survival rate.

2 Sequence Alignment

1. Describe in detail the dynamic programming algorithm used for computing a *global alignment* between two DNA sequences, noting its inputs, outputs and time complexity. Explain the significance of the *score matrix* in this context, and provide a score matrix that would convert this problem into the familiar LCS (*longest common subsequence*) problem.

   Definition 1 (Score Matrix)**.** A score matrix S is a similarity matrix where the element $S_{i,j}$ indicates how similar the i^{th} symbol is to the j^{th} symbol. This allows us to model similarity between codon and represent that dissimilar mutations are unlikely, leading to substantially better approximations with no additional complexity.

   Question 1 (Symmetry of Score Matrices)**.** Is there ever a situation where we would want a score matrix that is *not* symmetric? Why would this be useful?

   Definition 2 (Needleman-Wünsch algorithm)**.** The simplest possible algorithm to solve the Global Alignment problem. It takes as input two strings $v, w \in \Sigma^*$ and a score matrix S for the language $\Sigma \cup \{\_\}$ and provides as an output an alignment of v and w whose alignment score is maximal.

```python
def needleman_wunsch(v: str, w: str, S: dict[tuple[str, str], float]):
    mem = [[None for _ in range(len(w))] for _ in range(len(v))]
    mem[0, 0] = 0
    ptr = [[None for _ in range(len(w))] for _ in range(len(v))]
    mem[0, 0] = "•"

    for i in range(1, len(v)):
        mem[i, 0] = mem[i - 1, 0] - S["_", v[i - 1]]
        ptr[i, 0] = "↑"

    for j in range(1, len(w)):
        mem[0, j] = mem[0, j - 1] - S["_", w[j - 1]]
        ptr[0, j] = "←"

    for i in range(1, len(v) + 1):
        for j in range(1, len(w) + 1):
            mem[i][j] = max(
                mem[i - 1][j - 1] + score[v[i - 1], w[j - 1]],
                mem[i - 1][j] + score["_", w[j - 1]],
                mem[i][j - 1] + score["_", v[i - 1]],
            )
            match mem[i][j]:
                case mem[i - 1][j - 1] + S[v[i], w[j]]:
                    ptr[i][j] = "↖"
                case mem[i - 1][j] - S[\_, v[i - 1]]:
                    ptr[i][j] = "↑"
                case mem[i][j - 1] - S[\_, w[j - 1]]:
                    ptr[i][j] = "←"

    return mem, ptr
```

A matrix which would convert this problem into LCS is given below:

$$\begin{pmatrix} 1 & -\infty & \cdots & -\infty & 0 \\ -\infty & 1 & \cdots & -\infty & 0 \\ -\infty & -\infty & \cdots & -\infty & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ -\infty & -\infty & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 0 \end{pmatrix}$$

2. Implement your algorithm from question 1 in a language of your choice and use it to compute the global alignment (and alignment score) of the sequences `CGTGAA` and `GACTTAC`, with the following parameters:

   - Match: $+5$
   - Mismatch: -3
   - Insertion / deletion: -4

   You may also verify your result by computing it manually – it is good practice for the examinations.

   My implementation computed that the maximum alignment between the two strings according to the score matrix given is -3. This is validated by manually carrying out the computation.

		C	G	T	G	A	A
	0	−4	−8	−12	−16	−20	−24
G	−4	−3	1	−3	−7	−11	−14
A	−8	−7	−6	−2	−6	−2	−6
C	−12	−3	−7	−7	−5	−6	−5
T	−16	−7	−6	−2	−6	−9	−9
T	−20	−11	−10	−1	−5	−9	−12
A	−24	−15	−14	−5	−4	0	−4
C	−28	−19	−18	−9	−8	−4	−3

3. Outline the key transformations that need to be made to the algorithm in order to find optimal *local alignments*, as well as incorporating *affine gap penalties*. why are these features useful? Have you changed the time complexity of the algorithm by doing so?

   Definition 3 (Local Alignment Problem)**.** In the Local Alignment Problem, we take two strings v and w as input, and a score matrix. The output is then substrings s_v, s_w of v, w such that the global alignment between s_v and s_w is maximised.

   We can augment the Needleman-Wünsch Algorithm by initialising the edges of the graph to 0. This has the effect of ignoring the cost of any regions at the start or end of strings with low alignment. This is now the Smith-Waterman Algorithm for solving the Local Alignment Problem. This has both the same time and space complexity as the Needleman-Wünsch Algorithm.

Definition 4 (Affine Gaps Problem)**.** In the Affine Gaps Problem, we model the cost of a gap as of length g as $d + e \times (g - 1)$. The reasoning behind this is that larger gaps are probably caused by the same event and so should not be penalised as heavily. The input is two strings u, v and a score matrix. The output is an Alignment which has a minimal cost assuming we use Affine Gap Penalties.

There are two algorithms given for the Affine Gaps problem. One is an exact solution and the second is an inexact solution which the paper which Pietro extracted it from claims that it has small impact in most cases: and is correct in the case where $-d-e < \min(S)$. I provide the augmentations required to implement both.

In the exact solution, we have three matrices: `lower` stores the best cost alignment that ends in insertions; the best overall alignment; and the best alignment which ends in deletions. The exact algorithm is implemented by the following algorithm:

```
def affine_gaps(v: str, w: str, score, e, d):
    lower = [[None for _ in range(len(w) + 1)] for _ in range(len(w) + 1)]
    middle = [[None for _ in range(len(w) + 1)] for _ in range(len(w) + 1)]
    upper = [[None for _ in range(len(w) + 1)] for _ in range(len(w) + 1)]

    lower[0][0] = -e
    middle[0][0] = 0
    upper[0][0] = -e

    for i in range(1, len(v) + 1):
        lower[i][0] = -e - d * (i - 1)
        middle[i][0] = -e - d * (i - 1)

    for j in range(1, len(w) + 1):
        middle[0][j] = -e - d * (i - 1)
        upper[0][j] = -e - d * (i - 1)

    for i in range(1, len(v) + 1):
        for j in range(1, len(w) + 1):
            lower[i][j] = max(
                lower[i - 1][j] - d,
                middle[i - 1][j] - e,
            )
            upper[i][j] = max(
                upper[i][j - 1] - d,
                middle[i][j - 1] - e,
            )
            middle[i][j] = max(
                lower[i][j],
                upper[i][j],
                middle[i - 1][j - 1] + S[v[i - 1], w[j - 1]]
            )

    return middle[-1][-1]
```

The second algorithm gets an inexact solution. It only uses two matrices. The first `F` stores the optimal alignment; and the second `G` stores the optimal alignment which ends in a gap. The problem with this algorithm is that we consider a gap in one string followed by a gap in the *other* as a single gap and thus provide the affine penalty when creating it. The justification for this algorithm is that it only inexact in regions which have bad alignment anyway. I don't understand why this justifies a broken algorithm.

```
def affine_gaps(v: str, w: str, score, e, d):
    F = [[None for _ in range(len(w) + 1)] for _ in range(len(v) + 1)]
    G = [[None for _ in range(len(w) + 1)] for _ in range(len(v) + 1)]

    F[0][0] = 0
    G[0][0] = 0

    for i in range(1, len(v) + 1):
        F[i][0] = -e - d * (i - 1)
        G[i][0] = -e - d * (i - 1)

    for j in range(1, len(w) + 1):
        F[0][j] = -e - d * (i - 1)
        G[0][j] = -e - d * (i - 1)

    for i in range(1, len(v) + 1):
        for j in range(1, len(w) + 1):
            G[i][j] = max(
                G[i - 1][j] - d,
                G[i][j - 1] - d,
                F[i - 1][j] - e,
                F[i][j - 1] - e,
            )
            F[i][j] = max(
                G[i][j],
                F[i - 1][j - 1] + S[v[i - 1], w[j - 1]]
            )

    return F[-1][-1]
```

4. Provide an explanation (accompanied with a brief pseudocode and diagram) of how the storage complexity of the global alignment algorithm can be significantly reduced, while keeping time complexity the same. Provide an informal proof that the asymptotic time complexity does not increase.

   In the main iteration step, when we fill in the table, the current row only ever depends on the previous row. This means we can find the minimum cost alignment using only two rows: we deallocate the previous row when we have finished computing the next one since we will never use it again! However, this algorithm only computes the cost of the optimum global alignment.

   To get the alignment as well as its cost, note that we can simultaneously both work forwards to the middle of the string and backwards from the end of the string. When these two algorithms meet in the middle at index `mid` of the first string, we have two columns: `prefix` and `suffix`, where `prefix[i]` is "the greatest alignment between v[: `mid`] and w[: `i`]"; and `suffix[i]` is "the greatest alignment between v[`mid` :] and w[`i` :]". So notice that `length[i]prefix[i]+suffix[[i]]` is the cost of the best alignment which passes through the vertex [`mid`, `i`]. We can then divide the size of the problem in half and solve each case recursively. If we resolve duplicates consistently, then we can compute the optimal alignment in quadratic time and polynomial space. Essentially, we compute a node in the middle of the alignment in quadratic time and linear space. However, since we halve the size of the problem each time, the complexity is still quadratic.

$$T(n) = n^2 + 2 \cdot T\left(\frac{n}{2}\right) \implies T(n) = 2 \cdot n^2 \in \mathcal{O}\left(n^2\right)$$

A strange algorithm where you only compute half of the nodes and instead run an algorithm to compute the successor. This doesn't decrease the real time or the complexity. Furthermore, it doesn't make sense since we can easily compute the predecessor as well, which would reduce the number of nodes we have to compute by the exact same amount! Thus this optimisation appears not to give any advantage: and even if it did it would only be exploiting half the potential for optimisation.

I provide a working python implementation in the next question so shall not provide pseudocode

5. Implement the reduced storage variant of the global alignment algorithm in a language of your choice, and verify that it provides the same result for the inputs in question 2.

```
def middle_node(lo, hi, left, right, u, v, score):
    mid = (left + right) // 2

    # prefix
    prefix = [0]
    for i in range(lo, hi):
        prefix.append(prefix[-1] + score[u[i]]["_"])
    for i in range(left, mid):
        prefix_tmp = [prefix[0] + score[v[i]]["_"]]
        for j in range(lo, hi):
            prefix_tmp.append(
                max(
                    prefix[j - lo] + score[v[i]]["_"],
                    prefix[j - lo] + score[v[i]][v[j]],
                    prefix_tmp[-1] + score[u[j]]["_"],
                )
            )
        prefix = prefix_tmp

    # suffix
    suffix = [0 for _ in range(hi - lo + 1)]
    for i in range(hi - 1, lo - 1, -1):
        suffix[i - lo] = suffix[i - lo + 1] + score[u[i]]["_"]
    directions = ["" for _ in range(hi - lo + 1)]
    for i in range(right - 1, mid - 1, -1):
        suffix_tmp = [0 for _ in range(hi - lo + 1)]
        suffix_tmp[-1] = suffix[-1] + score[v[i]]["_"]
        for j in range(hi - 1, lo - 1, -1):
            suffix_tmp[j - lo] = max(
                suffix[j - lo + 1] + score[v[i]][u[j]],
                suffix[j - lo] + score[v[i]]["_"],
                suffix_tmp[j - lo + 1] + score[u[j]]["_"],
            )
            if i == mid:
                if suffix_tmp[j - lo] == suffix[j - lo + 1] + score[v[i]][u[j]]:
                    directions[j - lo] = "searrow"
                elif suffix_tmp[j - lo] == suffix[j - lo] + score[v[i]]["_"]:
                    directions[j - lo] = "to"
                elif suffix_tmp[j - lo] == suffix_tmp[j - lo + 1] + score[u[j]]["_"]:
                    directions[j - lo] = "darr"
                else:
                    assert False
        suffix = suffix_tmp

    m = max(range(hi - lo + 1), key=lambda x: prefix[x] + suffix[x])
```

```
    return (mid, m + lo), directions[m]  # middle node!


def linear_space_alignment(lo, hi, left, right, u, v, score):
    if hi == lo or hi == lo + 1:
        return [(x, lo) for x in range(left + 1, right)]
    node, edge = middle_node(lo, hi, left, right, u, v, score)
    (xmid, ymid) = node
    left_alignment = linear_space_alignment(lo, ymid, left, xmid, u, v, score)
    if edge in {"to", "searrow"}:
        xmid += 1
    if edge in {"darr", "searrow"}:
        ymid += 1
    right_alignment = linear_space_alignment(ymid, hi, xmid, right, u, v, score)

    return left_alignment + [node, (xmid, ymid)] + right_alignment


def reduced_storage_global_alignment(u, v, score):
    alignment = [(0, 0)]
    alignment.extend(linear_space_alignment(0, len(u), 0, len(v), u, v, score))
    return alignment
```

6. Describe the Nussinov algorithm for RNA secondary structure prediction, its underlying assumptions and time/storage complexities. What should be the output of the algorithm on the sequence `GCAACGUCG`? (this is a test of understanding – do not actually perform the algorthm on this sequence!)

   Definition 5 (Nussinov Algorithm)**.** A dynamic programming algorithm for predicting how RNA will fold. It starts by considering the most likely substructures of length 1, then iteratively uses the most likely substructures of length i to compute the most likely substructures of length $i + 1$. Nussinov's algorithm can model bulges, internal loops and multiloops. However, a critical limitation is that it *cannot model pseudoknots*! This makes it an oversimplification that is not used in practice.

   Nussinov's algorithm has a space complexity of $\mathcal{O}(n^2)$ for a sequence of length n and a time complexity of $\mathcal{O}(n^3)$. The additional time complexity comes from looking back for potential substructures which may merge (*i.e.* allowing substructures other tha a single stem with bulges and internal loops).

   In the example given, the algorithm would output that `C` binds with `G`, `A` with `U` and `C` with `G`.

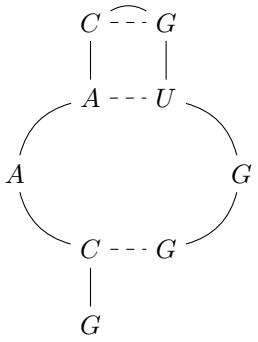


Figure 1: A diagram of a possible folding for the sequence `GCAACGUCG`

3 Phylogeny

1. Compare and contrast the *distance-based* with *parsimony-based* approaches to constructing phylogenetic trees, and comment on their relative merits.

   I consider the distance-based approaches to phylogeny to be UPGMA and Neighbour Joining; while the parsimony based approaches are the algorithms which solve the small/large parsimony problems.

   Distance-based approaches to phylogeny are more resistant than parsimony based approaches: parsimony based approaches do not allow comparisons between strings which do not have the same length. Furthermore, parsimony-based approaches can have strings which are very similar to each other rated very differently: they suffer from the problem with hamming distances (*i.e.* that they do not take account of insertion or deletion thus `TATATATAT` and `ATATATATA` would be rated very differently).

   The large parsimony is an NP-complete problem doesn't have an exact solution: this means that the algorithms that solve it are not exact and have no guarantees. No good polynomial-time approximation algorithm exists for problems like this. The heuristic algorithm proposed has a time complexity of $\Omega(n^2 \cdot m \cdot \Sigma^2)$ – guaranteed to be asymptotically worse than that for distance based metrics!

   However, the Parsimony-based algorithms solve more general problems than the distance-based solution. The large parsimony problem can create either a rooted or unrooteed tree and makes no assumptions about the structure of the tree. The UPGMA algorithm does not preserve distances and the neighbour joining algorithm can not be used to make a rooted tree. In summary, parsimony-based algorithms are more general but take more time and have more weaknesses than distance based algorithms.

2. For distance-based approaches, it is often desirable for the distance matrix to be *additive*. Provide a mathematical formulation of the additivity property. [Hint: Consider any four leaf nodes (i, j, k, l) in the tree.]

   Definition 6 (Additive Matrix)**.** A Distance Matrix is additive if there exists an unrooted tree fitting it, specifically if there is an unrooted tree such that $\forall i, j \in S.\, D_{i,j} = \text{path\_cost}(i, j)$.

3. What does it mean for a phylogenetic tree to be *ultrametric*, and what assumption does it make on the underlying evolutionary process.

   A phylogenetic tree is *ultrametric* if all leaves of the tree have the same path length to the root. This assumes that the rate at which species evolve is constant.

4. Compare the three covered approaches to distance-based phylogeny (Additive, UGPMA and Neighbour Joining), in terms of properties and computational complexities.

   UPGMA builds an ultrametric tree. It is given a set of nodes as input. For n the number of taxa, the time complexity is $\mathcal{O}(n^3)$ and the space complexity is $\mathcal{O}(n^2)$.

   Neighbour Joining solves the distance-based phylogeny problem. It is given a distance matrix as input. for n the number of taxa, the time complexity is $\mathcal{O}(n^3)$ (naïvely – there exist more optimal implementations which are not covered) and the space complexity is $\mathcal{O}(n^2)$.

   The tree produced by NJ is not rooted, while distances between two nodes in the tree produced by UPGMA does not necessarily represent the same distance between them in the distance matrix.

   Additive phylogeny was not covered: I shall not discuss it.

5. Apply UPGMA and Neighbour Joining to the following distance matrix:

	A	B	C	D
A	0	2	4	6
B		0	4	6
C			0	6
D				0

- **UPGMA**

  UPGMA recursively merges the two taxa which are the most similar (*i.e.* merges the $\min_{i,j} D_{i,j}$).

	A	B	C	D
A	0	2	4	6
B		0	4	6
C			0	6
D				0

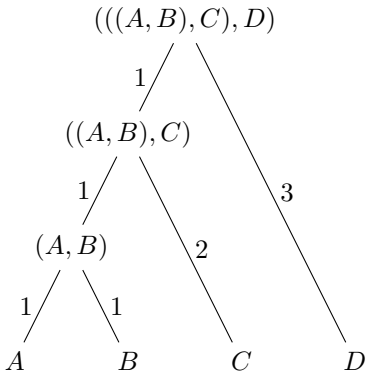
Table 1: Merge A and B to form (A, B) where the height of (A, B) is given by $\frac{2}{2} = 1$

	(A, B)	C	D
(A, B)	0	4	6
C		0	6
D			0

Table 2: Merge (A, B) and C to form $((A, B), C)$ where the height of $((A, B), C)$ is given by $\frac{4}{2} = 2$

	((A, B), C)	D
((A, B), C)	0	6
D		0

Table 3: Merge $((A, B), C)$ and D to form $(((A, B), C), D)$ where the height of $(((A, B), C), D)$ is given by $\frac{6}{2} = 3$



- **Neighbour Joining**

	A	B	C	D
A	0	2	4	6
B	2	0	4	6
C	4	4	0	6
D	6	6	6	0

Table 4: D

The first step of neighbour joining is to form the Neighbour Joining Matrix D^* defined on an $n \times n$ matrix as $D^*_{i,j} = (n-2) \cdot D_{i,j} - \sum_{k \in \mathbb{N}_{<n}} D_{i,k} - \sum_{k \in \mathbb{N}_{<n}} D_{k,j}$.

	A	B	C	D
A	0	-20	-18	-18
B	-20	0	-18	-18
C	-18	-18	0	-20
D	-18	-18	-20	0

Table 5: D^*

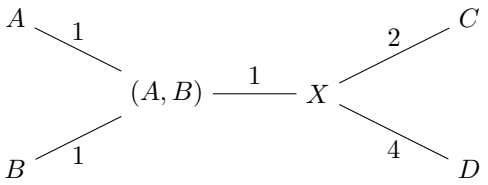
The next step of Neighbour Joining is to merge the two nodes which have the lowest score in the neighbour joining table. In this case, that's A and B. Create a new node (A, B). Define $\Delta_{A,B} = \frac{\sum_{k \in \mathbb{N}_{<n}} D_{A,k} - \sum_{k \in \mathbb{N}_{<n}} D_{k,B}}{n-2} = 0$. Now set the cost of edge $A \to (A, B)$ to $\frac{D_{A,B}+\Delta_{A,B}}{2} = 1$ and $B \to (A, B)$ to $\frac{D_{A,B}-\Delta_{A,B}}{2} = 1$. Now, remove A and B from the table, creating a node (A, B) where $D_{(A,B),k} = \frac{D_{A,k}+D_{B,k}-D_{A,B}}{2}$.

	(A, B)	C	D
(A, B)	0	3	5
C	3	0	6
D	5	6	0

Table 6: D

We have now reached the base case (size 3) and it is guaranteed that D^* will be of the form $k \cdot (\mathbf{1} - \mathbf{I})$. Thus we do not have to run the algorithm anymore and may apply the algebraic solution to form the correct tree.

Create a new node X. $\Delta_{C,D} = \frac{-2}{1} = -2$ thus we have $\text{Cost}(C \to X) = \frac{D_{C,D}+\Delta_{C,D}}{2} = \frac{6-2}{2} = 2$ and $\text{Cost}(D \to X) = \frac{D_{C,D}-\Delta_{C,D}}{2} = \frac{6+2}{2} = 4$. We also have $\text{Cost}((A, B) \to X) = 3 - 2 = 1$.



Question 2 (Base Case)**.** Both Pietro and original paper state that there is are $n - 3$ iterations a base case for $n = 3$. This is a base case that works. I derived my own working base case which is mathematically equivalent to the base case. What is the *official* way of stating the base case?

6. Implement either UPGMA or NJ in a language of your choice, and verify your findings from the previous question.

   I implemented UPGMA: the results validated the correctness of my algorithm

```
from itertools import combinations


def upgma(distances):
    # number of elements in the groups
    sizes = {k: 1 for k in distances.keys()}
```

```
    # ages of the nodes
    ages = {k: 0 for k in distances.keys()}
    while len(distances) > 1:
        # find the least D_{i,j}
        i, j = min(
            combinations(distances.keys(), 2), key=lambda x: distances[x[0]][x[1]]
        )
        dij = distances[i][j]

        # remove rows and columns i, j
        dist_i = distances.pop(i)
        dist_j = distances.pop(j)
        for dist in distances.values():
            dist.pop(i)
            dist.pop(j)

        # create the new node m
        m = f"({i}, {j})"
        sizes[m] = sizes[i] + sizes[j]
        ages[m] = dij / 2
        dist_m = {
            k: (sizes[i] * dist_i[k] + sizes[j] * dist_j[k]) / (sizes[i] + sizes[j])
            for k in distances.keys()
        }
        for k in distances.keys():
            distances[k][m] = dist_m[k]
        distances[m] = dist_m

    return next(iter(distances)), ages
```

7. What is the input, output and time complexity of the dynamic programming algorithm for *small parsimony*? Illustrate the algorithm on a small example of your choice.

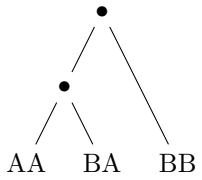
   The input to the dynamic programming algorithm for the *small parsimony* problem is: a rooted tree with n nodes where each leaf has an associated string of length m. Sankoff's algorithm also takes a mutation matrix which ranks how expensive it is for one symbol to turn into another. Variants which do not use a score matrix typically use the Kronecker Delta.

   The output of the algorithm is an assignment of strings of length m on all n nodes of the tree which minimizes the Parsimony. Parsimony is the sum of the hamming distances (or a weighted variant if we consider Sankoff's algorithm) across all edges.

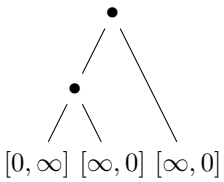
   The time complexity of the algorithm is $\mathcal{O}(n \cdot m \cdot \Sigma^2)$ for a tree with n nodes where each string has length m and symbols drawm from the alphabet Σ.

   For each character in the strings; the algorithm aims to compute the minimum cost (of all edges between descendants) for every node to take every possible value. Once it has computed the minimum cost for the root to take every value, it sets the value of the root to the minimum-cost value and backtracks down the tree filling in the values as it goes. Initialise the costs for any leaf as: the cost for them to take their actual value is 0 and the cost for them to take any other value is ∞.

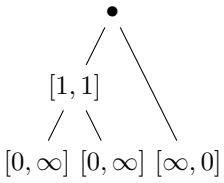
   Consider the following example over the language $\{A, B\}$



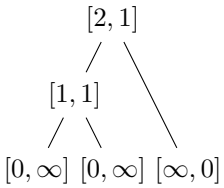
We iterate through every character in the strings. For the first character in the strings, we initialise the tree with the following costs:



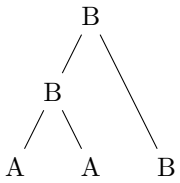
Next, compute the cost of the interior nodes whose children all have computed costs.



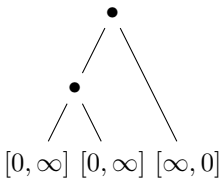
Finally, compute the cost of the root:



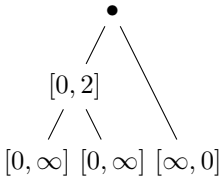
Now we have computed the minimum cost for the root. In this case, the value of the root which minimizes the total cost is B. The value of the intermediate node which achieved the minimum cost for the root was B. Thus the intermediate node also takes the value B, leaving us the following tree:



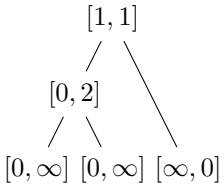
Now, for the second character in the string: initialise the cost for the leaves of the tree.



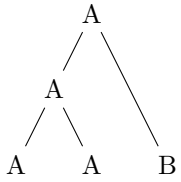
Next, compute the cost of the interior nodes whose children all have computed costs.



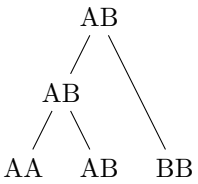
Compute the cost of the interior nodes whose children all have computed costs.



Now we have computed the minimum cost for the root. We backpropagate the values down the tree. In this case, I will break the tie by choosing A. The value of the intermediate node which achieved the minimum cost for the root was A. Thus the intermediate node also takes the value A, leaving us the following tree.



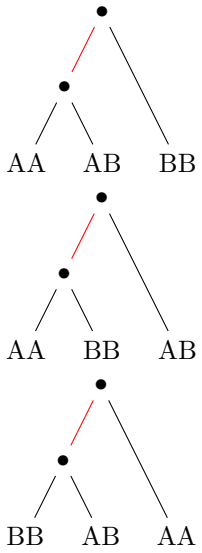
Combining the results for each of the characters gives the following tree:



8. Briefly explain, with the aid of a diagram where appropriate, the greedy heuristic for *large parsimony*. Illustrate the candidate trees generated by the algorithm for the small example you used in the previous question.

   The large greedy heuristic for the large parsimony problem is the *Nearest Neighbour Interchange Heuristic*. In this, we start by initialising a random binary tree and computing its parsimony. For each edge in the tree, we consider the subtrees of nodes on either end of it and consider the tree formed by permuting them. For each possible tree, we solve the small parsimony problem and compute the total parsimony of the tree. Consider the new tree with the least parsimony: we replace the original tree with this tree and update the parsimony.

   Imagine that we are solving the large parsimony problem for a rooted tree and randomly initialise to the tree we used as an example for the small parsimony problem in the last question. Consider the neighbours formed by permuting neighbours via the internal edge.



4 Multiple Alignment

1. Compare and contrast the *dynamic programming*, *greedy* and *progressive* approaches to aligning k sequences of length n, highlighting their respective time and memory complexities.

   The dynamic programming approach to aligning sequences of length n is the generalisation to pairwise alignment. The input to this algorithm is a score matrix $\delta : \Sigma^k \to \mathbb{R}$ and k strings. The output is an alignment between them. This algorithm proceeds by constructing k-dimensional grid where each dimension is of size $n + 1$. The equation for the $(x_1, x_2, \ldots, x_k)^{\text{th}}$ element is a function which computes the cost over all the possible places indels could be placed and takes the minimum of these. This is exponential in k. This algorithm is guaranteed to compute the exact global alignment: but it's time complexity is $\mathcal{O}(2^k \cdot n^k)$ and its space complexity is $\mathcal{O}(n^k)$ (and the cost of storing the score matrix of size Σ^k). Thus, the dynamic programming approach is computationally intractable on for all but the smallest k.

   The progressive alignment algorithm is a heuristic algorithm (which is not guaranteed to get the same alignments) and has a space complexity of $\mathcal{O}(n \cdot m^2 + n^2)$ and a time complexity of $\mathcal{O}(m^2 \cdot n^2)$. Thus it's computationally tractible for larger datasets.

2. Explain the inputs and steps performed by the CLUSTALW algorithm.

   The progressive alignment method for aligning m sequences of length n is a more computationally tractable heuristic approach. It starts by computing the pairwise global alignments between every pair of taxa in the dataset. This takes $\mathcal{O}(m^2 \cdot n^2)$. This is then input to a distance matrix which is used to build a guide tree *i.e.* using distance-based phylogeny approaches. This tree is then used to guide the order in which alignments should be made. Start at the root and progressively align the current string / alignment to the least similar taxa.

3. How would you assign a *"quality" score* to an obtained multiple alignment.

   The sum of the entropy of the columns. For each column, we compute the observational probability that the alignment takes each value and then sum over the columns. We chen compute the entropy of each of the columns and use this as a metric of the overall quality of the alignment. We can even plot the entropy by columns to see which parts of the taxa align particularly well.

   Consider the following example:

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

Table 7: a possible multiple alignment

$P(A)$	1	0.25	0.25	0
$P(C)$	0	0.25	0.25	0
$P(G)$	0	0.25	0.5	0.5
$P(T)$	0	0.25	0	0.5

Table 8: The observational probabilities corresponding to Table 7

$h(x)$	0	2	1.5	1

Table 9: The entropies for each column

Overall, the entropy of the alignment is 4.5 bits. We can see that the first column is a very good alignment, while the second column is a very bad one by plotting this in a graph.

Question 3 (Indels)**.** How does this method deal with indels? Consider a column where 10% of values are A and 90% are indels. This is an *incredibly* misaligned column! However, the entropy measure would appear to rank it as being very good since the entropy is low *i.e.* "almost certainly an indel!".

5 Approximate Search

BLAST is no longer in the syllabus.