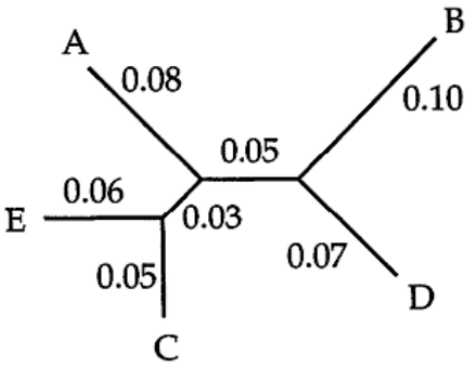
PROJECT 5

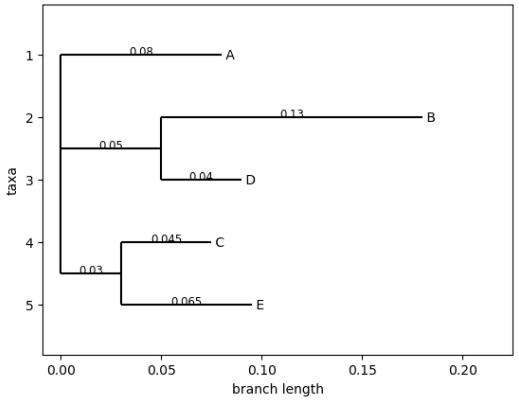
Status of your work, does your program work correctly on the test case in example_slide4.phy, i.e. from this matrix produces the tree on slide 4. If not, what do you think is wrong and what have you done to alleviate it.

Our Nj Program works as intended on the example from the slides, giving us the same tree and similar distances.

Tree on slide 4:



Tree produced by our algorithm:



Although the program works as intended on the example, the later experiments show that our implementation is slow, resulting in a quite time consuming process (>2h for the biggest file). Therefore we have not been able to compute the bigger files as our later table will show. Some modifications have been tried, including preprocessing and changing some parts of the algorithm to be ‘inplace’, but it was not sufficient to reduce the running time enough to make the larger experiments feasible.

A description of your implementation of nj explaining what you have done in order to make it as efficient as possible, including which programming language(s) you have used.

To calculate the distance parameter Dij-Ri+Rj and keep track of the pair/rowwise distances we compute a matrix N, the computation takes o(n**2).

```
def calculateN(d,nodes):
    s = len(d)
    N = np.zeros((s,s))

    for i in range(s):
        for j in range(s):
            if i != j:
                sumI = (1/ (s-2)) * sum(d[i,label] for label in nodes.values())
                sumJ = (1/ (s-2)) * sum(d[j,label] for label in nodes.values())
                N[i][j] = d[i,j] - (sumI + sumJ)
            else:
                N[i][j] = 0
    return N
```

And the lowest distance was found using .unravel_index, which locates the min in O(n):

```
def minimum_entry(N):
    return np.unravel_index(N.argmin(), N.shape)
```

Then we compute the new edge length for i-k and j-k, k being the new node:

```
def getNewEdges(i,j,d):

    sumI = sum(d[i,k] for k in range(len(d)))
    sumJ = sum(d[j,k] for k in range(len(d)))

    dki = round(0.5 * (d[i,j] + sumI  - sumJ),3)
    dkj = round(0.5 * (d[i,j] + sumJ  - sumI),3) #- dki
    return (dki,dkj)
```

Following the computation of the new edge length, we update the distance matrix: we have a dictionary of nodes that contain the index to which their distances are in the matrix. We remove these two indices of the dictionary and generate a new set of indices for the remaining nodes in the tree (O(n)).

```
def updateDistanceMatrix(a,b,m,keys):
    ## get the indices of the nodes to merge
    i = keys[a]
    j = keys[b]

    ## update the new keys
    updated_nodes = keys.copy()
    ### remove a and b
    updated_nodes.pop(a)
    updated_nodes.pop(b)
    ### update the indices
    updated_nodes = {k: i for i, k in enumerate(sorted(updated_nodes.keys(), key=lambda x: updated_nodes[x]))}
```

Now we update in the “new” distance matrix the distances between the nodes that do not take part in the merging operation (O(n**2)) and generate the new index for the merged node:

```
size_nd = len(m)-1
nd = np.zeros((size_nd,size_nd))
## update distance matrix for the nodes that are not involved in the merging operation
for node, upd_index in updated_nodes.items():
    index_in_m = keys.get(node)
    for node_ , upd_index_ in updated_nodes.items():
        index_in_m_ = keys.get(node_)
        nd[upd_index,upd_index_] = m[index_in_m,index_in_m_]

### generate a new index for the new node
updated_nodes[a+b] = len(updated_nodes)
```

And finally, compute the distances from all the nodes to the new merged node (O(n)):

```
    ## compute the distance from all the nodes to the merged node
    for node, upd_index in updated_nodes.items():
        # get the index in the original distance matrix
        index_in_m = keys.get(node)
        # if the node exists in the matrix (it is not the merged node), get the distance
        if index_in_m is not None:
            nd[-1][upd_index] = (m[i][index_in_m] + m[j][index_in_m] - m[i][j]) / 2.
            nd[upd_index][-1] =(m[i][index_in_m] + m[j][index_in_m] - m[i][j]) / 2.
```

Therefore, the computational cost of running the **updateDistanceMatrix** function is O(n**2). Now we wrap all our functions to complete our implementation of NJ, returning a newick tree.

```
def NeighbourJoining(d,nodes):

    S = len(d)
    T = {}
    # d -> np array + dictionary of nodes and indices
    while S > 3:
        ## 1.a Compute N
        N = calculateN(d,nodes)
        ## 1.b Find the min in N
        lowestPair = minimum_entry(N)

        ## 2. Add a new node k to the tree T
        ## 3. add edges with weights
        i = lowestPair[0]
        j = lowestPair[1]
        newEdges = getNewEdges(i,j,d)
        #nodes[(i,j)] = newEdges

        node_a = list(nodes.keys())[i]
        node_b = list(nodes.keys())[j]
        ## 4. Update the ds matrix by deleting rows and columns corresponding
        ## to i and j and adding a new row and column for the new taxon k
        d,nodes,new_node = updateDistanceMatrix(node_a,node_b,d,nodes)

        ## save Newick format
        ## print("Merging: ({}:{},{}:{})".format(node_a, newEdges[0], node_b, newEdges[1]))
        T[new_node] = f"({node_a}:{newEdges[0]},{node_b}:{newEdges[1]})"

        S-=1

    ## termination
    i, j, m = 0,1,2
    remaining_nodes = list(nodes.keys())
    v_i = round(0.5 * (d[i,j] + d[i,m] - d[j,m]),3)
    v_j = round(0.5 * (d[i,j] + d[j,m] - d[i,m]),3)
    v_m = round(0.5 * (d[i,m] + d[j,m] - d[i,j]),3)

    newick = f"({remaining_nodes[i]}:{v_i},{remaining_nodes[j]}:{v_j},{remaining_nodes[m]}:{v_m})"+';'

    for clade, newick_format in T.items():
        newick = newick.replace(clade,newick_format)

    return newick
```

The total asymptotic running time of the Neighbour Joining algorithm we implemented here is O(n**3), where n is the number of nodes (or taxa) in the input tree. This is because the main algorithm loop iterates n-3 times, and within each iteration, the function calculateN() performs two nested loops over all nodes, resulting in a time complexity of O(n**2). Additionally, the function updateDistanceMatrix() performs two nested loops over all remaining nodes, which also has a time complexity of O(n**2). Therefore, the total running time is O(n)*2*O(n**2) = O(n**3).

A description of the machine (cpu, ram, os, ...) that you have used to perform the experiment and how you have measured the running time.

The details of the machine used for the experiment:

```
Processor: Intel64 Family 6 Model 142 Stepping 10, GenuineIntel
System: Windows 10
Architecture: AMD64
RAM: 7.86 GB
Python version: 3.9.16
```

We have measured the running time of QuickTreeNJ and RapidNJ using this bash script and for the running time of our algorithm you can find the details in the TestingNJ Jupyter Notebook.

A table summarizing the results of your experiment. For each of 14 distance matrix in distance_matrices.zip, your table should contain a row with 8 entries that report:

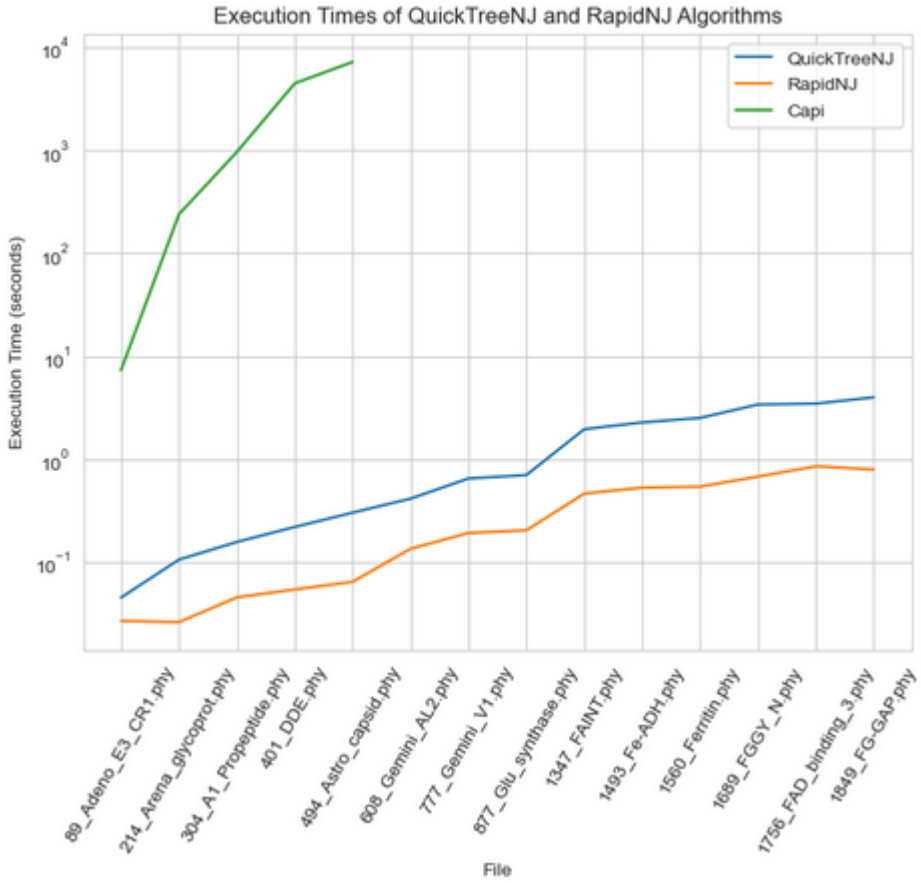


Figure 1. Running time of QuickTree, RapidNJ and Capi (our implementation), Running time is logscaled.

When we compare our algorithm with RapidNJ or QuickTree, it becomes quite clear that it doesn't scale well with large numbers of sequences. Additionally, we observe that rapidNJ compared to the quicktree implementation, congruent with the finding *M. Simonsen et al.*

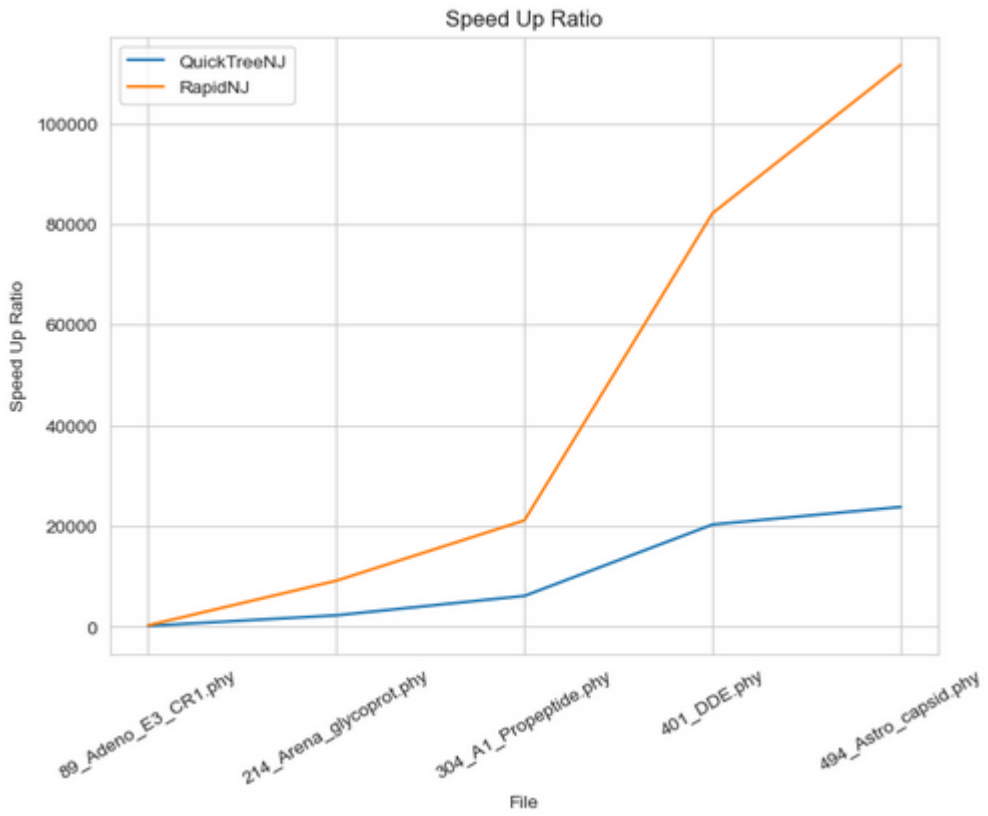


Figure 2. The relative speed up of quicktree/rapidNJ compared to the capi implementation (eg. Capi / QuickTree)

With a speed up ratio of >20000 on 304 sequences for rapidNJ and ~10000 for QuickTree, it is abundantly clear that both of these implementations out compete ours, And that RapidNJ out competes QuickTree.

Out[51]:

	214_Arena_glycoprot.nwk	304_A1_Propeptide.nwk	401_DDE.nwk	494_Astro_capsid.nwk	89_Adeno_E3_CR1.nwk
Distance our implementation vs QuickTree	216	306	403	496	91
Distance our implementation vs RapidNJ	216	306	403	496	91
Distance QuickTree vs RapidNJ	56	80	100	532	32

Figure 3. The RF-distance between trees computed by QuickTree, Capi and RapidNJ.

We have compared the trees computed by the three different algorithms and for some reason the distance is constant between our implementation and the other two implementations. We have tried to debug it and do it manually and debug it again, but we keep seeing the same pattern. An interpretation of this could be that there is a constant amount of difference between Capi and the other two, but since the distance between QuickTree and RapidNJ is nonzero, the differences between Capi/RapidNJ and Capi/QuickTree are not the same, but it seems highly improbable.