Nathan Newbury **Hw 2_Writeup**

i) Recursive Formula + Base Cases

After setting the first score and arrow as 0 and none respectively, we initialize each axis with the formula below:

```
S[i][0][0] = (S[i-1][0][0][0] + 2*gap, [1,0,0])
```

Because no other letters are added on the axis , we calculate the sum of the pairwise score of an additional letter as 2 gaps (The pairwise score of gap to gap is 0) and add it to the previous score. The axis only has one previous box to choose from and the arrow is set from this.

Then we initialize each face with the formula below:

```
Score =max( S[i-1][j-1][0][0] + B[s1[i-1]][s2[j-1]] + 2*gap, S[i][j-1][0][0] +
2*gap, S[i-1][j][0][0] + 2*gap)
```

This is almost identical to the 2 dimensional alignment problem. In this case, we use 2 gap penalties as we always add a gap in the third dimension.

We also record the arrow, depending on the option picked:

```
            if max_index == 0:
                arrow = [1, 1, 0]
            elif max_index == 1:
                arrow = [0, 1, 0]
            elif max_index == 2:
                arrow = [1,0, 0]
```

The arrow can only point in 3 directions, just like the 2 pair alignment. We have to run this algorithm for all 3 initial faces (i,j and i,k and j,k).

Finally the Recursive formula is below, with 7 previous boxes to choose from:

```
score = max([S[i-1][j-1][k-1][0] + B[s1[i-1]][s2[j-1]] + B[s1[i-1]][s3[k-1]] +
B[s2[j-1]][s3[k-1]],
                                 S[i][j-1][k-1][0] + 2*gap + B[s2[j-1]][s3[k-1]],
                                 S[i-1][j][k-1][0] + 2*gap + B[s1[i-1]][s3[k-1]],

                                 S[i-1][j-1][k][0] + 2*gap + B[s1[i-1]][s2[j-1]],

                                 S[i][j][k-1][0] + 2*gap,

                                 S[i][j-1][k][0] + 2*gap,

                                 S[i-1][j][k][0] + 2*gap])
```

We also store the arrow based on these options.

```
                if max_index == 0:

                    arrow = [1, 1, 1]

                elif max_index == 1:

                    arrow = [0,1, 1]

                elif max_index == 2:

                    arrow = [1,0,1]

                elif max_index == 3:

                    arrow = [1,1,0]

                elif max_index == 4:

                    arrow = [0,0,1]

                elif max_index == 5:

                    arrow = [0,1,0]

                elif max_index == 6:

                    arrow = [1,0,0]
```

We recorded the arrow for backtracking. The arrow is in the form: [Δi,Δj,Δk]. A 1 in any of these dimensions means we move back in that direction by 1. An arrow of [1,1,1] would backtrack in each direction. This gives us 7 cases [0,0,1], [0,1,0], [1,0,0], [0,1,1],[1,0,1], [1,1,0], [1,1,1]. We also keep track of our current position [i,j,k]. Our current position - arrow gives us our new position. The code is seen below:

```
currentposition = [a - b for a, b in zip(currentposition, arrow)]
```

We find the arrow of the new position as we loop through the path. The direction of the arrow determines whether we add a gap, or a full letter. If we back track in a direction, we need to add a letter to our aligned string. If we don't back track in a direction (an arrow of 0) we add a gap to that string (Each dimension of our cube has a string associated with it).

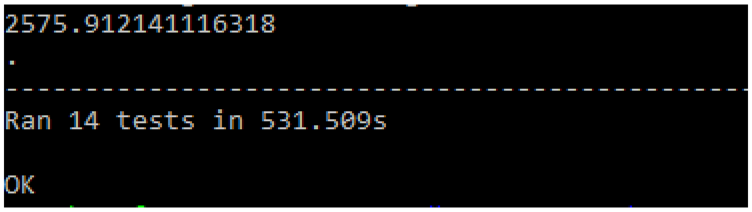
ii) Asymptotic Running Time

We are no longer using a dictionary, but I am still going to assume constant lookup time and constant subproblem time (compute maximum and add gap penalty etc). Additionally, I am going to assume that the backtracking time is negligible. Filling in the full cube, or the main recursive algorithm, scales the worst and takes the most time. Because our sub problem time is approximately constant (finding the maximum of a set of 7 and writing to our list), the time complexity can be approximated as O(|E| + |V|*d) where |E| is the number of edges, |V| is the number of subproblems and d is our sub-problem time . Normally |V|<|E|<|V|^2. In our case |E| = 7*|V|. With a string length of n, |V| = n^3. This means our time complexity is O(7n^3 + d*n^3) = O(n^3).

iii) Gap Penalty

The total gap penalty is calculated as $Log_2(\beta)$. If we can calculate β we can find our gap penalty. $\Sigma(\beta q_a)$ is the total probability of an insertion or a deletion of the letter q_a. An insertion or deletion can result in the substitution of letter A or a gap. So $Indelrate * P(substitution) = \Sigma(\beta q_a)$. $\Sigma(q_a)$ is the probability of a letter, which approximates to 1 (was like .9987). Thus the gap penalty is $Log_2(Indelrate * P(substitution) / \Sigma(q_a))$. The probability of substitution can be calculated by summing all of the non-diagonal elements of my P matrix.

The code for some reason only runs in verbose. Below is proof I passed all 14 tests:\



```
2575.912141116318
.
----------------------------------------------------
Ran 14 tests in 531.509s

OK
```