Question 1

DNA is made of chemical building blocks called nucleotides. The nitrogen bases found in these nucleotides are: adenine (A), thymine (T), guanine (G) and cytosine (C). A is a complementary base to T, and G is a complementary base to C. A DNA sequence is said to be a palindrome if it's equal to its reverse complement. For example, ACCTAGGT is palindromic since it's complement is TGGATCCA. Reversing this complement gives us back the original sequence. Let's call this a "genetic palindrome".

A palindrome partition arises when a given sequence is broken into sub-strings such that each sub-string is also a palindrome. For example, "ACA—G—CCTCC—G—T—ACA" is a palindrome partitioning of the sequence "ACAGCCTCCGTACA", with 5 cuts. If the sequence is a palindrome, 0 cuts are needed.

Your job as a bored lab assistant is as follows: given a DNA sequence, find the fewest cuts required to palindrome partition a sequence. You are very interested in 'genetic palindrome' partitions but finding them seems too difficult. You decide first you will work out how to calculate palindrome partitions efficiently.

1.1 You are given a list S of n nitrogen bases. We want to build an $O(n^3)$ Dynamic Programming algorithm that will find the minimum cuts 'genetic palindrome' partition (You are guaranteed that one exists). Define the sub-problems, the base cases and the recursion relation between the sub-problems and justify the time complexity of your solution.

We define the sub-problem P(i,j) as the minimum cuts to create a palindrome partition of the sub-list S[i:j]. The base case is P(i,j) = 0 if S[i:j] is a palindrome or 1 otherwise. We may define the recurrence relation as $P(i,j) = min\{P(i,k) + P(k+1,j) + 1, i \leq k \leq j-1\}$. We are populating an n by n lookup table which takes $O(n^2)$ and for each index we must check if the corresponding sub-string is a palindrome which takes $O(n)$ so the time complexity of this algorithm is $O(n^3)$ overall.

- **[2 marks]** Correct sub-problems identified
- **[2 marks]** Correct base cases identified
- **[2 marks]** Valid recurrence relation
- **[1 mark]** Justification for time complexity provided

1.2 Describe your Dynamic Programming algorithm in pseudocode and verify that it achieves the desired $O(n^3)$ time complexity based on your coded solution.

We can solve the problem by taking a bottom-up approach. We also know that every substring of length 1 is a palindrome.

```
// Returns the minimum number of cuts
// needed to partition a string
// such that every part is a palindrome
minPartitons(string seq)
{
    n = seq.length

    // bool pal[n][n] = 1 if substring seq[i..j] is
```

```
//          palindrome, else 0
// int cuts[n][n] = min number of cuts for
//            substring seq[i..j]

//initialization
for (int i = 0; i < n; i++) {
    pal[i][i] = true;
    cuts[i][i] = 0;
}

// l is length of the subsequence
for (int l = 2; l <= n; l++) {

    //try every starting index for each subsequence
    for (int i = 0; i < n - l + 1; i++) {
        int j = i + l - 1; // Set ending index

        //case for just 2 characters in the sequence
        if (l == 2)
            pal[i][j] = (seq[i] == seq[j]);
        else
            pal[i][j] = (seq[i] == seq[j]) && pal[i + 1][j - 1];

        // if seq[i..j] is palindrome, then C[i][j] is 0
        if (pal[i][j] == true)
            cuts[i][j] = 0;
        else {

            // Make a cut at every possible
            // location and get the minimum cost cut.
            cuts[i][j] = int_max; //some arbitrary large number
            for (int k = i; k <= j - 1; k++)
                cuts[i][j] = min(cuts[i][j],
                cuts[i][k] + cuts[k + 1][j] + 1);
        }
    }
}

// minimum cuts for given subsequence
return cuts[0][n - 1];
```

The running time for this algorithm is $O(n^3)$, since there is a loop with 2 inner loops present in the solution. Justification provided in 1.1.

- [**1 mark**] Takes a top-down or bottom up dynamic programming solution
- [**4 marks**] Provides a correct solution to the problem with justifications for the algorithm design
- [**1 mark**] Correct time complexity with valid reasoning
- Pseudocode or solutions with better running times are also accepted.

1.3 Describe how you could optimize your algorithm so that the time complexity is reduced to $O(n^2)$.

What is the time complexity of finding all sub-strings of S which are palindromes?

For this question it is important to notice that, for our algorithm in 1.1, it takes $O(n^3)$ because we have to check if each sub-string is a palindrome as we go, with each check taking $O(n)$. If we pre-compute which sub-strings of S are palindromes and store this information in our lookup table Pal[i][j], then we may subsequently perform our algorithm from 1.1 but it will only take $O(n^2)$ as we may check if a sub-string is a palindrome in $O(1)$ using our lookup table. Since we may pre-compute which sub-strings of S are palindromes in $O(n^2)$, and then perform our algorithm from 1.1 using these results in $O(n^2)$, the overall time complexity has been reduced to $O(n^2)$

- [2 mark] Recognises how we may perform pre-computation to reduce time complexity **- [1 mark]** Provides justification for why the precomputing reduces the time complexity