AAD Project

Aum Khatlawala, 2020113008

The contents of this pdf and the website are the same but the website offers an interface with clickables, while this pdf offers a more coherent representation of my project. To access the website, please click on the index.html file in the html folder.

1 Matrix Multiplication

1.1 My interpretation and analysis of the algorithm

If we have two 2×2 matrices X and Y and we wish to multiply them, we would do it as follows:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

This would be easy for a human brain to compute but would be considered a naive solution when it comes to algorithms because it would have a time complexity of $O(n^3)$. Why? Because there are n^2 entries to be computed and each takes $O(n)$ time. A more precise explanation for this is as follows:
we get 8 $n/2$ sized products: $AE, BG, AF, BH, CE, DG, CF$ and DH. So the total running time can be calculated as follows:
$T(n) = 8T(n/2) + O(n^2)$, which turns out to be $O(n^3)$.
However, using Strassen's Divide and Conquer Algorithm, the total time complexity has been proven to be reduced to $O(n^{\log 7})$ which is equal to $O(n^{2.81})$. Here, n is the dimension of the square matrices we are multiplying.
Strassen was able to reduce the eight computation products (for n=2) to seven using a trick which is so precise that it makes us wonder how he ever discovered it in the first place. He listed the decomposition as follows:

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

$$P_1 = A(F - H)$$
$$P_2 = (A + B)H$$
$$P_3 = (C + D)E$$
$$P_4 = D(G - E)$$
$$P_5 = (A + D)(E + H)$$
$$P_6 = (B - D)(G + H)$$
$$P_7 = (A - C)(E + F)$$

Thus, for 7 computations, running time is $T(n) = 7T(n/2) + O(n^2)$. This equals $O(n^{\log 7})$ which is approximately equal to $O(n^{2.81})$.
Recently, however, a Refined Laser Method was introduced by Alman and Williams to further reduce the complexity of Matrix Multiplication to $O(n^{2.3728596})$. This shows us that this algorithm has been evolving for quite some time and we still haven't found the perfect optimum solution for it.
The code:

```
int main()
{
    int X[2][2], Y[2][2], XY[2][2];
    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 2; j++)
        {
            scanf("%d" , &X[i][j]);
        }
    }
    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 2; j++)
        {
            scanf("%d" , &Y[i][j]);
        }
    }
    int A = X[0][0];
    int B = X[0][1];
    int C = X[1][0];
    int D = X[1][1];
```

```
    int E = Y[0][0];
    int F = Y[0][1];
    int G = Y[1][0];
    int H = Y[1][1];
    int P_1 = A * (F - H);
    int P_2 = (A + B) * H;
    int P_3 = (C + D) * E;
    int P_4 = D * (G - E);
    int P_5 = (A + D) * (E + H);
    int P_6 = (B - D) * (G + H);
    int P_7 = (A - C) * (E + F);
    XY[0][0] = P_5 + P_4 - P_2 + P_6;
    XY[0][1] = P_1 + P_2;
    XY[1][0] = P_3 + P_4;
    XY[1][1] = P_1 + P_5 - P_3 - P_7;
    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 2; j++)
        {
            printf("%d", XY[i][j]);
        }
        printf("\n");
    }
}
```

1.2 Applications in different fields

The most basic application is in the subject of Linear Algebra. Matrix Multiplication forms the basis of Linear Algebra calculations (Linear maps, Eigenvectors, Inner products and so on) and the fact that Linear Algebra is such a fundamental concept in multiple fields in Computer Science such as graphics, machine learning, computer vision, cryptography and quantum algorithms makes it even more important to search for the best possible algorithm to solve matrix multiplication.
Another important advantage of matrix multiplication is that it can model essentially any linear system of equations. Now, linear equations are used extensively in and are critical to multiple fields including Modern Physics and Chemistry. Let's take a few examples:

1. If we need to find the magnitude of magnetic field described by a matrix at some

point in space described by another matrix, we just need to multiply the magnetic field by the geographic matrix.

2. If we want to build a search engine using a sparse matrix (mostly consisting of 0s) of websites and how they connect together, we can use the power iteration algorithm (given a diagonalizable matrix A, the algorithm will produce a number λ, which is the greatest (in absolute value) eigenvalue of A) to accomplish the task. This is exactly what led to the foundation of Google.

3. The mechanism of self driving cars is based on the concept of neural networks in the computer vision system and this relies on real time matrix multiplication done by GPUs.

2 Tower of Hanoi

2.1 My interpretation and analysis of the algorithm

The Tower of Hanoi is a very popular problem in the realm of algorithms and it is the perfect example to see how computer algorithms can also be used to solve real life fun problems.
Problem Statement - The Tower of Hanoi consists of three vertical towers and N disks of different sizes, each shaped like a donut. The disks are originally stacked on top of each other in one of the towers in order of decreasing size (largest disk at the bottom). The goal of the problem is to move all disks to a different rod while keeping three rules in mind:

1. Only one disk can be moved at a time.

2. Only the disk at the top of a stack may be moved.

3. A disk may not be placed on top of a smaller disk.

Approach to solve the problem - We shall solve the problem by dividing it into sub problems. Let's say the original configuration of the disks is in Tower A and we need to move the disks to Tower C and Tower B is the auxiliary tower. The scenarios are as follows:

1. Single disk in Tower A - This is simple, just move the disk from Tower A to Tower C and we're done with the solution.

2. Two disks in Tower A - Move the upper disk from A to B, move the other, larger, disk from A to C and then move the disk in B to C. We're done.

3. Three disks in Tower A - Move the two upper disks from A to B using C. How? Same method as the one used for two disks, but with the destination being B. This will take 3 moves. Now, move the largest disk, which is still in A, to C. Now, move the two disks in B to C using A. How? Same intuition as the first 3 steps in this case. We're done. As you can see, we have used recursion of the two disks scenario twice to solve the three disks scenario. Now, we generalise it to N disks.

4. N disks in Tower A - Move $N-1$ disks from A to B using C. Then, move the largest disk in A to C and then, move $N-1$ disks from B to C using A.

The code:

```
void tower_of_hanoi(int n, int A, int B, int C)
{
    // n => number of disks
    // A => move from this tower
    // B => move using this auxiliary tower
    // C => move to this tower
    if (n > 0)
    {
        tower_of_hanoi(n - 1, A, C, B);
        printf("Move a disk from %d to %d", A, C);
        tower_of_hanoi(n - 1, B, A, C);
    }
};
```

From the code snippet, we can easily see that the total time taken would be $T(n) = T(n-1) + 1 + T(n-1) = 2T(n-1)$. Let's call this equation 1.
Putting $n = n-1$ in equation 1, we get $T(n-1) = 2T(n-2) + 1$. Let's call this equation 2.
Putting $n = n-2$ in equation 1, we get $T(n-2) = 2T(n-3) + 1$. Let's call ths equation 3.
Combining equation 2 and 3 by substituting value of $T(n-2)$ from equation 2 in equation 3, we get:
$T(n-1) = 2T(2T(n-3)+1)+1$. Let's call this equation 4.
Putting value of $T(n-1)$ in equation 1 from equation 4, we get:
$T(n) = 2(2(2T(n-3)+1)+1)+1 = 2^3T(n-3) + 2^2 + 2^1 + 1$.
We generalise this to the following:
$T(n) = 2^kT(n-k) + 2^{(k-1)} + 2^{(k-2)} + ... + 2^2 + 2^1 + 2^0$.
Our base condition is: $T(1) = 1$, so $k = n-1$.

Put $k = n - 1$ in the generalised equation and we get:
$T(n) = 2^{(n-1)}T(n - (n - 1)) + 2^{(k-1)} + 2^{(k-2)} + ... + 2^2 + 2^1 + 2^0$.
$T(n) = 2^k(1) + 2^{(k-1)} + 2^{(k-2)} + ... + 2^2 + 2^1 + 2^0$.
This is a Geometric Progression with $r = 2$ and $a = 1$.
Sum of GP $= a(1 - r^n)/1 - r$.
Thus, $T(n) = 1.(1 - 2^n)/(1 - 2) = 2^n - 1$.
This means $T(n) = O(2^n - 1)$ which gives us a time complexity of $O(2^n)$.

2.2 Applications in different fields

The Tower of Hanoi is a very common puzzle and is mainly used in psychological research on how people approach certain problems. Multiple variants of the Tower of Hanoi puzzle have been made solely for neuropsychological diagnosis (for example: frontal lobe deficits) and treatment of executive functions of the human body.
Jiajie Zhang and Donald Norman published a paper in Psychology which used different variants of the Tower of Hanoi for empirical investigations into the principles of distributed representations. They came to multiple conclusions about external representations of problem statements posed in the different isomorphs of the Tower of Hanoi puzzle. They also concluded the following: The distributed representations approach has shed light on questions like whether cognition is solely in the mind or distributed across the mind and the environment and whether people reason on formal structures or on content-specific representations. Practically, it can provide design principles for effective representations. This paper has helped in the development of a framework for human-computer interaction.
It is also used as a scheme for data back up rotation which is essential in minimising the number of media (like tapes) used to back up data. Using the Tower of Hanoi scheme, the first tape backs up data once every 2 days, the second tape backs up data once every 4 days, the third tape backs up data once every 8 days and so on. Thus, n tapes will allow backups for 2^{n-1} days.
The most interesting application of the Tower of Hanoi was observed recently (in 2010), where researchers found out that an ant species *Linepithema humile* was able to solve the 3 disk Tower of Hanoi problem through pheromone signals and non linear dynamics.

3 Huffman Encoding

3.1 My interpretation and analysis of the algorithm

Huffman Encoding answers the question - what is the most economical way to encode a given long string in binary.
Let's take an example: We want to encode a 120 million character string made up of the alphabets A, B, C, D, E, F, G *and* H. The traditional approach to encode this in binary would be to assign a two bit binary number to each alphabet ($A \rightarrow 000,\ B \rightarrow 001,\ C \rightarrow 010$ and $D \rightarrow 011,\ E \rightarrow 100,\ F \rightarrow 101,\ G \rightarrow 110$ and $H \rightarrow 111$) and encode it to give us a total length of 360 million bits. Now, there must be a way to encode it better. How do we do it? Huffman Encoding is the answer. We try to do variable length encoding based on how frequently each alphabet appears in the string.

Alphabet	Frequency
A	24 million
B	45 million
C	3 million
D	3 million
E	23 million
F	4 million
G	8 million
H	8 million

Now, the Huffman Encoding algorithm tells us to proceed as follows to achieve variable length encoding of the string.

1. Build a priority queue that contains 8 nodes where each node represents the root of a tree with single node.

2. Extract two minimum frequency nodes ($C \Rightarrow 3$ million and $F \Rightarrow 4$ million) from the queue and add a new internal node with frequency 7 million.

3. Extract two minimum frequency nodes ($D \Rightarrow 5$ million and $(C + F) \Rightarrow 7$ million) and add a new internal node with frequency 12 million.

4. Extract two minimum frequency nodes ($G \Rightarrow 8$ million and $H \Rightarrow 8$ million) from the queue and add a new internal node with frequency 16 million.

5. Extract two minimum frequency nodes ($(D + (C + F)) \Rightarrow 12$ million and $(G + H)$ $\Rightarrow 16$ million) and add a new internal node with frequency 28 million.

6. Extract two minimum frequency nodes ($E \Rightarrow 23$ million and $A \Rightarrow 24$ million) from the queue and add a new internal node with frequency 47 million.

7. Extract two minimum frequency nodes ($((D + (C + F)) + (G + H)) \Rightarrow 28$ million and $B \Rightarrow 45$ million) and add a new internal node with frequency 73 million.

8. Extract two minimum frequency nodes ($(E + A) \Rightarrow 47$ million and $(((D + (C + F)) + (G + H)) + B) \Rightarrow 73$ million) and add a new internal node with frequency 120 million.

Thus, we have done a variable length encoding of the 120 million character string using Huffman Encoding. Now we see whether this actually provides an improvement on the total amount of bits required to encode. With the normal encoding scheme, we were able to encode the string in 360 million bits. Now, for Huffman Encoding, we get the total cost of the tree as follows - the sum from $i = 1$ to $i = n$ of (frequency of i^{th} symbol $\times$ depth of i^{th} symbol in tree). Here, the calculation for that is as follows - $(2 \times 23)+(2 \times 24)+(2 \times 45)+(4 \times 5)+(5 \times 3)+(5 \times 4)+(4 \times 8)+(4 \times 8) = 303$ million bits. This is a 16 percent improvement on the encoding length.
Now, let's have a look at how the two Greedy Approach Properties are satisfied in this algorithm:

1. Greedy Choice Property - The two symbols with the smallest frequencies must be at the bottom of the optimal tree, as children of the lowest internal node. Why? It's simple. Because otherwise, swapping these two symbols with whatever is lowest in the tree would improve encoding.

2. Optimum Substructure Property - Define the frequency of any internal node to be the sum of the frequencies of its descendant leaves. The cost of a tree is the sum of the frequencies of all leaves and internal nodes except the root. Any tree in which f_1 and f_2 are sibling leaves has cost $(f_1 + f_2)$ plus the cost for a tree with $(n - 1)$ leaves of frequencies $(f_1 + f_2), f_3, f_4 ... f_n$. Using the greedy choice statement and applying it recursively, we affirm the optimum substructure property. Essentially, in every iteration, we remove the bottom two leaves and combine their frequencies to form the node which is their parent at $(n - 1)$ level.

The structure of the code:

```
void huffman_encoding(int f[])
{
    int n = sizeof(f);
    // f is an array of frequencies
    // Let H be a priority queue of integers ordered by f
```

```
    for (int i = 1; i <= n; i++)
    {
        // insert(H, i)
    }
    for (int k = n + 1; k <= 2 * n - 1; k++)
    {
        // i = deletemin(H), j = deletemin(H)
        // create a node numbered k with children i, j
        // f[k] = f[i] + f[j]
        // insert(H, k)
    }
    // return Huffman tree
};
```

The time complexity of the Huffman algorithm is $O(n \log n)$. Using a heap to store the weight of each tree, each iteration requires $O(\log n)$ time to determine the cheapest weight and insert the new weight. There are $O(n)$ iterations, one for each item.

3.2 Applications in different fields

It is easy to imagine the potential use cases of Huffman encoding. Let's try to list down some generic real life applications of encoding a piece of information.

1. Data compression: This is a very common use case of encoding which is used by the whole world in this day and age.
    - Audio and video streaming / uploading is only possible due to compression techniques like mp3, jpeg, mpeg4 and so on.
    - Large pieces of data can easily be stored and transferred in computer networks using processes like ZIPing files and RARing files. These are a result of encoding data.
2. The concept of barcodes and QR codes is possible only due to the process of encoding. It allows us to easily access data that we might need instead of performing manual operations to access that chunk of data.
3. Encoding standards like ASCII and UTF help encode computer understandable formats to human readable characters for making computer data human readable. Emojis are also a result of such encoding and decoding protocols.
4. Multiple different types of codes like morse code and gray code use digital encoding and were popular in the last century.

Huffman encoding, as proven during the course of this article, is more efficient than a lot of the standard encoding methods and for this reason, it is used often when it comes to any type of encoding. GZIP, JPEG and PNG formats heavily use Huffman encoding.
Multiple communications with and from the internet are Huffman encoded at some point. Multiple communication protocols use it.
Huffman encoding, combined with Dynamic Programming, also serves as the basis of lossless compression algorithms like Brotli Compression (released by Google).

4 Horn Formulas

4.1 My interpretation and analysis of the algorithm

The whole concept of Horn Formulas and prolog (programming by logic) came from the need for computers to display some level of logical reasoning similar to us humans. Horn Formulas are a way to derive logical conclusions based on certain facts and rules provided to it. Let's understand the basics of Horn Formulas by taking the example where the Horn Formula is a Boolean variable - it takes either true or false as a value. Imagine the murder of a high ranking military general took place at his house.
Let variables u, v, w, x, y *and* z denote the following statements:

1. $u \rightarrow$ the colonel is innocent
2. $v \rightarrow$ the wife is innocent
3. $w \rightarrow$ the murder took place at 8 pm
4. $x \rightarrow$ the murder took place in the kitchen
5. $y \rightarrow$ the butler is innocent
6. $z \rightarrow$ the colonel was asleep at 8 pm

We will call a variable x or its negation x' a literal. We have two kinds of clauses in Horn Formulas:

1. Implications - These clauses are of the form 'If the conditions on the LHS hold, the conditions on the RHS must also be true by implication'. Two examples from the example scenario: '$(z \wedge w) \Rightarrow u$' means 'if the colonel was asleep at 8 pm and the murder took place at 8 pm, the colonel is innocent.' '$\Rightarrow x$' is a degenerate type of implication which means 'the murder took place in the kitchen'. Implications lead us to setting some variables to true.

2. Pure negative clauses - These clauses contain an OR of any number of negative literals. For example, '$(u' \vee v' \vee y')$' means 'all three suspects can't be innocent'. Pure negative clauses lead us to setting some variables to false.

Now that we have understood the basic terminology with the help of an example, we move onto the technicalities. Our goal is to determine whether we can assign either true or false values to the variable that satisfies all the clauses. This is called the satisfying assignment.
Strategy to solve Horn Formulas:

1. Set all variables to false.

2. Set some variables to true very carefully keeping implications in mind. While there is an implication that is not satisfied, set the variable on the RHS to true.

3. Look at the negative clauses left and make sure the pure negative clauses are satisfied. If yes, we have a satisfying assignment. If no, the formula provided does not have a satisfying argument.

Now, we can show that this solution uses a Greedy approach to test for satisfying assignments because the two properties for Greedy Approach are satisfied in the second phase of the algorithm - the first and third phase of the strategy are only for setting and checking. The two properties are as follows:

1. Greedy Choice Property - We do know the first step towards the optimum solution. Set the first variable with the implication clause to true.

2. Optimum Substructure Property - Once the first-step is taken, can we restate the rest of the problem as a smaller version of the original problem and thus, solve using recursion? Yes. The global final answer has to be built from the local answers for each implication.

Now, why is the strategy correct? Because, if the strategy returns an assignment, this assignment satisfies both the implications and the negative clauses and is thus a valid satisfying assignment of the Horn Formula. Now, we need to convince ourselves that if the strategy doesn't return a satisfying assignment, there isn't one anyways. This is due to the fact that if a certain set of variables is set to true, then they must be true in any satisfying assignment. Thus, if the truth assignment found after the second phase doesn't satisfy the negative clauses, there is no satisfying assignment.
The structure of the code:

```
void horn_formulas(char* formula)
{
    // apply regex to the formula and set all variables to false

    // while there is an implication that is not satisfied:
    //      set the RHS variable of the implication to true

    // if all pure negative clauses are satisfied:
    //    return the assignment
    // else:
    //      return ("Formula is not satisfiable");
};
```

Let's take an example dry run to understand this better.
Input: $(w \wedge y \wedge z) \Rightarrow x, (x \wedge z) \Rightarrow w, x \Rightarrow y, \Rightarrow x, (x \wedge y) \Rightarrow w, (w' \vee x' \vee y'), (z')$.
Process:
First, set all variables w, x, y, z to false. Then,

1. set x to true because ' $\Rightarrow x$ '.
2. set y to true because '$x \Rightarrow y$ '.
3. set w to true because '$(x \wedge y) \Rightarrow w$ '.

Now, looking at the pure negative clauses, we see that the first one is not satisfied.
Output: Formula is not satisfiable.

4.2 Applications in different fields

It is easy to see why Horn Formulas might be used in multiple fields. We provide a set of rules and it outputs a statement that satisfies the rules. It is an important part of Logic Programming which is used extensively in Computer Science because first order logic (set of axioms / rules) is well understood and can represent all computational problems.
Horn Formulas play a major role in various branches of Computer Science like Data Science, Machine Learning and so on. Why so? Universal Horn theories (set of Horn clauses) are exactly the framework in which the notion of any generic example can be applied. Horn Formulas are an improvement on the initial models used in logic programming and this is why Horn Theory is so popular.
Since Horn Formulas are a special kind of Syntactic Unification (solving equations between symbolic expressions), they can be used in multiple areas of Computer Science

that use Syntactic Unification. Some of these include: Theorem Proving, Natural Language Processing, Pattern matching, Scientific calculators, Expert Systems, Parsing and Query Languages.

5 Edit Distance

5.1 My interpretation and analysis of the algorithm

This algorithm deals with the problem of the closeness of two strings. Now, a good measure of closeness of two strings is the extent to which they can be aligned. So, the edit distance problem deals with the question - how many minimum edits (insertions, deletions and substitutions / conversions) are required to convert one word to another. Let's take an example. To convert the word 'Saturday' to 'Sunday', we need 3 minimum edits. This can be visualised as follows:

$$SATURDAY$$

$$S--UNDAY$$

So, we delete A and T and convert R to N to get 'Sunday' from 'Saturday'.
We could easily identify the optimal answer in this case, however, there could be multiple non optimal ways to reach the answer. How do we write an algorithm for the minimum answer? Based on the example, we can say that the problem boils down to finding the best alignment for the two words given and then finding how many places the letters don't match in. We can also observe that the two properties required for the Dynamic Programming Approach are satisfied for this kind of a problem:

1. Overlapping Subproblems Property: solutions of the same subproblems are needed again and again. This can observed if we try to write a recursive algorithm to solve this. This is because we can reach the same subproblem from two different tree paths while writing the recursive algorithm to solve this.

2. Optimal Substructure Property: optimal solution of the given problem can be obtained by using optimal solutions of its subproblems. How? This is what we shall understand now.

Let's provide the intuition to solve the $E(i, j)$ subproblem. There are three possible cases:

1. The i^{th} letter of the first string is aligned with a dash.

2. A dash is aligned with the j^{th} letter of the second string.

3. The i^{th} letter of the first string is aligned with the j^{th} letter of the second string.

We can express $E(i, j)$ in terms of smaller subproblems.
$E(i, j) = \min\{1 + E(i-1, j), 1 + E(i, j-1), diff(i, j) + E(i-1, j-1)\}$.
This might look like a gigantic complicated formula, but it is rather simple to understand.
The minimum edit distance for the subproblem is equal to the minimum out of the three where $1 + E(i-1, j)$ is for the first case (we need to find minimum distance for the smaller subproblem $E(i-1, j)$ and add 1 to it because the i^{th} letter of the first string is aligned with a dash indicating a mismatch), $1 + E(i, j-1)$ is for the second case (we need to find minimum distance for the smaller subproblem $E(i, j-1)$ and add 1 to it because the j^{th} letter of the second string is aligned with a dash indicating a mismatch) and $diff(i, j) + E(i-1, j-1)$ is for the third case (we need to find minimum distance for the smaller subproblem $E(i-1, j-1)$ and add 1 if the i^{th} and j^{th} letter of the respective strings aren't the same and add 0 if the i^{th} and j^{th} letter of the respective strings are the same). Now, we only need to decide the base cases. $E(0, j)$ is the edit distance between the 0-length prefix of x, namely the empty string, and the first j letters of y: clearly, j. And similarly, $E(i, 0) = i$.
The structure of the code:

```
int editdistance(const char *word1, int m, const char *word2, int n)
{
    int E[m + 1][n + 1];
    for (int i = 0; i <= m; i++)
    {
        E[i][0] = i;
    }
    for (int j = 0; j <= n; j++)
    {
        E[0][j] = j;
    }
    for (int i = 0; i <= m; i++)
    {
        for (int j = 0; j <= n; j++)
        {
            // E(i, j) = min{E(i-1, j) + 1, \
            // E(i, j-1) + 1, E(i-1, j-1) + diff(i, j)}
        }
    };
    return E[m][n];
```

}

5.2 Applications in different fields

Computers have become such an important tool for studying human and other animals' genes that computational genomics has become a new field in itself. Here are some examples of common questions that arise in this realm:

1. When a new gene is discovered, one way to learn more about its function is to look for known genes that are similar to it. This is especially useful for transferring knowledge from well-studied species, such as mice, to humans. The basic idea of the search problem is to see how closely two gene strings match. A generalisation of edit distance and Dynamic Programming can be used to effectively compute it. There is also another problem of searching through gene databases like GenBank. This is done using methods like BLAST, which is a combination of algorithmic tricks like the ones used in edit distance and biological concepts. This has led to it being the most widely used computational biology software.

2. DNA sequencing methods include DP algorithms as the fragment sizes are around 500-700 characters long (with the property of Overlapping Subproblem) and they need to be assembled into a coherent DNA sequence of billions of these.

3. The concept of phylogenetic tree and its computation methods (after gene sequencing) is also something that can be solved using computational biology softwares.

Edit Distance is also used in finding out how similar a set of DNA / protein sequences are and this forms the basis of methods like UPGMA which find the phylogenetic relationship between different species.
Autocorrection models can also be improved using the algorithm for Edit Distance.
Approximate string matching, where the objective is to find matches for short strings in many longer texts, also uses edit distance.
Hirschberg's algorithm uses the concept of minimising edit distance to compute the optimum alignment of two strings.
Levenshtein automata are finite state machines that recognize a set of strings within bounded edit distance of a fixed reference string.

6 Egg Dropping Puzzle

6.1 My interpretation and analysis of the algorithm

The puzzle is as follows:
Imagine there are 2 eggs and a building with 100 floors and we want to know which stories of the building are safe to drop eggs from. What is the least number of egg droppings that will help us find the answer to the question?
We make a few assumptions:

1. If an egg survives the fall, it can be used again.
2. A broken egg is discarded.
3. All eggs are identical in terms of effect of a fall.
4. If an egg breaks on the i^{th} floor, it'll break from all floors above the i^{th} floor.
5. If an egg doesn't break on the i^{th} floor, it won't break on a floor lesser than the i^{th} floor.
6. No assumptions are made about either the 1^{st} floor or the 100^{th} floor.

Before we move onto understanding the solution, let's look at a property of this puzzle that will help us understand the approach easily.
This is the Optimal Substructure Property:

1. If the egg breaks after dropping from the x^{th} floor, then we only need to check the lower floors with the remaining eggs. Therefore, the problem reduces to $x-1$ floors and $n-1$ eggs.
2. If the egg doesn't break after dropping from the x^{th} floor, then we only need to check for higher floors. Therefore, the problem reduces to $k-x$ floors and n eggs.

We proceed keeping these two properties in mind and also keeping in mind that we have to deal with the worst case scenario at every stage and then find the minimum possible number of trials. So the problem can be remodeled as follows:

$$egg\_drop(n,k) = 1 + \min\{\max(egg\_drop(n-1,x-1), egg\_drop(n,k-x))\}$$

If we try solving the problem using recursion, the approach would be as follows:
Let's say we have two eggs and two floors. Let's try throwing the egg from the first floor.
We have two cases:

1. If the egg breaks from the first floor, then the first floor is the threshold floor. So, the egg needs to be dropped from 0 floors to minimise the total number of trials.

2. If the egg doesn't break from the first floor, then we only need to test the second floor to get the answer. So, the answer becomes 1.

Thus, for the first floor, $egg\_drop(2, 1) = 1 + \max(0, 1) = 2$.
Now, if we try throwing from the second floor, we have two cases:

1. If the egg breaks from the second floor, then we will have 1 egg and 1 floor (second floor) to find the threshold.

2. If the egg doesn't break from the second floor, then the egg needs to be dropped from 0 floors to minimise the number of trials (because there are only 2 floors).

Thus, for the second floor, $egg\_drop(2, 2) = 1 + \max(1, 0) = 2$.
Thus, the total answer is 2 as it is the minimum value from 2 and 2.
This was the recursive method of solving the problem. But if we try to visualise this approach for higher number of floors, we realise that there are many values in the recursion tree being computed multiple times. This means there is an Overlapping Subproblem Property to this problem too. This increases the time complexity of the recursive algorithm to exponential time. We can solve this hurdle by using memoisation in a Dynamic Programming approach.
The approach will be to maintain a matrix / table with the results of all subproblems as and when we calculate the answer to a subproblem. For example, to fill the $(i, j)^{th}$ entry of the matrix (where i is the number of eggs and j is the number of floors for that subproblem), we traverse for each floor x from 1 to j and find the minimum of the following expression: $(1 + \max(matrix[i-1][j-1], matrix[i][j-x]))$.
It is best to understand this by visualising the matrix.
Let's say we have 2 eggs and 6 floors. The matrix will be as follows (with each row corresponding to an egg and each column corresponding to a floor):

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 2 & 3 & 3 & 3 \end{bmatrix}$$

How do we get this?
The first row is straight forward to fill as for one egg, the minimum number of attempts to find out from which floor the egg will break (in the worst case scenario) is the number of floors itself.
It gets trickier for 2 floors. Let's go through it step by step.

- For 2 eggs and 1 floor, the number of eggs is more than number of floors, it doesn't help and we just copy the value given in 1 egg and 1 floor entry.

- For 2 eggs and 2 floors, I have explained the reasoning earlier. The answer is 2.

- For 2 eggs and 3 floors;
  if we drop from first floor and it breaks, we have 0 as the answer. If it doesn't break, we have 2 floors and 1 egg to work with. To find this, we refer to the $(1,2)^{th}$ entry of the matrix and see the answer is 2. So, $1 + \max(0,2) = 3$.
  if we drop from the second floor and it breaks, we have 1 floor and 1 egg to work with, so the answer is 1 ($(1,1)^{th}$ entry). If it doesn't break, we have 1 floor and 2 eggs to work with, so the answer from the matrix is 1. So, $1 + \max(1,1) = 2$.
  if we drop from the third floor and it breaks, we have 2 floors and 1 egg to work with, so the answer is 2. If it doesn't break, we have 0 as the answer. So, $1 + \max(2,0) = 3$.
  So the $(2,3)^{th}$ entry is $\min(3,2,3) = 2$.

- For 2 eggs and 4 floors;
  Dropping from the first floor $\rightarrow$ Egg breaks $\rightarrow$ 0 and if egg doesn't break $\rightarrow$ 3 floors and 2 eggs to work with $\rightarrow$ 2. So, $1 + \max(0,2) = 3$.
  Dropping from the second floor $\rightarrow$ Egg breaks $\rightarrow$ 1 floor and 1 egg to work with $\rightarrow$ 1 and if egg doesn't break $\rightarrow$ 2 floors and 2 eggs to work with $\rightarrow$ 2. So, $1 + \max(1,2) = 3$.
  Dropping from the third floor $\rightarrow$ Egg breaks $\rightarrow$ 2 floors and 1 egg to work with $\rightarrow$ 2 and if egg doesn't break $\rightarrow$ 1 floor and 2 eggs to work with $\rightarrow$ 1. So, $1 + \max(2,1) = 3$.
  Dropping from the fourth floor $\rightarrow$ Egg breaks $\rightarrow$ 3 floors and 1 egg to work with $\rightarrow$ 3 and if egg doesn't break $\rightarrow$ 0. So, $1 + \max(3,0) = 4$.
  So the $(2,4)^{th}$ entry is $\min(3,3,3,4) = 3$.

- For 2 eggs and 5 floors;
  $1 \rightarrow$ breaks $\rightarrow$ 0 and doesn't break $\rightarrow$ 4 floors and 2 eggs $\rightarrow$ 3. So, $1+\max(0,3) = 4$.
  $2 \rightarrow$ breaks $\rightarrow$ 1 floor and 1 egg $\rightarrow$ 1 and doesn't break $\rightarrow$ 3 floors and 2 eggs $\rightarrow$ 2. So, $1 + \max(1,2) = 3$.
  $3 \rightarrow$ breaks $\rightarrow$ 2 floors and 1 egg $\rightarrow$ 2 and doesn't break $\rightarrow$ 2 floors and 2 eggs $\rightarrow$ 2. So, $1 + \max(2,2) = 3$.
  $4 \rightarrow$ breaks $\rightarrow$ 3 floors and 1 egg $\rightarrow$ 3 and doesn't break $\rightarrow$ 1 floor and 2 eggs $\rightarrow$ 1. So, $1 + \max(3,1) = 4$.
  $5 \rightarrow$ breaks $\rightarrow$ 4 floors and 1 egg $\rightarrow$ 4 and doesn't break $\rightarrow$ 0. So, $1+\max(4,0) = 5$.

So the $(2,5)^{th}$ entry is $\min(4,3,3,4,5) = 3$.

- For 2 eggs and 6 floors;
  $1 \rightarrow$ breaks $\rightarrow 0$ and doesn't break $\rightarrow 5$ floors and 2 eggs $\rightarrow 3$. So, $1+\max(0,3) = 4$.
  $2 \rightarrow$ breaks $\rightarrow 1$ floor and 1 egg $\rightarrow 1$ and doesn't break $\rightarrow 4$ floors and 2 eggs $\rightarrow 3$. So, $1 + \max(1,3) = 4$.
  $3 \rightarrow$ breaks $\rightarrow 2$ floors and 1 egg $\rightarrow 2$ and doesn't break $\rightarrow 3$ floors and 2 eggs $\rightarrow 2$. So, $1 + \max(2,2) = 3$.
  $4 \rightarrow$ breaks $\rightarrow 3$ floors and 1 egg $\rightarrow 3$ and doesn't break $\rightarrow 2$ floors and 2 eggs $\rightarrow 2$. So, $1 + \max(3,2) = 4$.
  $5 \rightarrow$ breaks $\rightarrow 4$ floors and 1 egg $\rightarrow 4$ and doesn't break $\rightarrow 1$ floor and 2 eggs $\rightarrow 1$. So, $1 + \max(4,1) = 5$.
  $6 \rightarrow$ breaks $\rightarrow 5$ floors and 1 egg $\rightarrow 5$ and doesn't break $\rightarrow 0$. So, $1+\max(5,0) = 6$.
  So the $(2,5)^{th}$ entry is $\min(4,4,3,4,5,6) = 3$.

The code:

```
int egg_drop(int n, int k)
{
    // n = eggs
    // k = floors
    int matrix[n + 1][k + 1];
    for (int j = 1; j <= k; j++)
    {
        matrix[1][j] = j;
    }
    for (int i = 1; i <= n; i++)
    {
        matrix[i][0] = 0;
        matrix[i][1] = 1;
    }
    int i_j_entry_candidate;
    for (int i = 2; i <= n; i++)
    {
        for (int j = 2; j <= k; j++)
        {
            matrix[i][j] = 9999;
            for (int x = 1; x <= j; x++)
```

```
            {
                int maximum = max(matrix[i - 1][x - 1], matrix[i][j - x]);
                i_j_entry_candidate = 1 + maximum;
                if (i_j_entry_candidate < matrix[i][j])
                {
                    matrix[i][j] = i_j_entry_candidate;
                }
            }
        }
    }
    return matrix[n][k];
}
```

6.2 Applications in different fields

There aren't too many applications of the Egg Dropping Puzzle except in Psychological studies and a few other domains. It is similar to the Tower of Hanoi problem in terms of both being puzzles but it isn't as famous as the Tower of Hanoi problem.
The Egg Dropping puzzle, however, is a very interesting puzzle and is thus used for many cognitive ability tests and it is also an interview question for Computer Science students.

7 Basics of Cryptography

We shall talk about the basics of encryption. Encryption is a method of converting data into an undecipherable format so that only the authorized parties can access the information. Cryptographic keys are used along with encryption algorithms to make the process of encryption possible.
There are two types of encryption methods based on the way the cryptographic keys are applied - Symmetric encryption (aka private key encryption) and Asymmetric encryption (aka public key encryption).
First, let's talk about Symmetric Encryption. As the name implies, it uses a single cryptographic key to encrypt and decrypt data. The most primitive use of Symmetric Encryption was proposed by Julius Caesar (Caesar's cipher). Let's imagine a situation where Alice wants to send an encrypted message to Bob using this protocol but wants to make sure that an eavesdropper, Eve, can't decipher the message. To do this, Alice and Bob meet up beforehand and choose a secret codebook with rules for deciphering the encrypted message. Let's say that the codebook is very straightforward and has the

only rule as 'each letter in the encrypted message is replaced by a letter 10 positions down the alphabet'. Now, using this, if Alice sends a message to Bob, Eve will only be able to decipher the message by partially using some encoded messages to figure out the codebook. Although this was a simple example, it helps in giving us the basic understanding.

We shall talk about one kind of Symmetric Encryption scheme called the one-time pad. In this scheme, Alice and Bob meet up beforehand and select a string r of the same length as the message Alice plans to send later on.

Now, to encrypt the message, Alice takes a bitwise XOR of each bit in the message with the corresponding bit in the string r. Let's say the message she wants to send is 11110000 and the string r is 01110010.

The encryption function will work as follows: $e(x) = x \oplus r = 11110000 \oplus 01110010 = 10000010$. Now, the reason this works is because if we apply the encryption function again, we get the original message.

How? $e(e(x)) = (x \oplus r) \oplus r = x \oplus (r \oplus r) = x \oplus 0' = x$ (where 0' is a string of all zeros with the same length as r and thus, the same length as x). Therefore, to decrypt, Bob just needs to apply the encryption function to the encoded message.

Now, how will this stop Eve from intercepting the message? Simple. Bob and Alice should pick r at random in such a way that the resulting string is equally likely to be an element of $\{0,1\}^n$. This will ensure that if Eve intercepts the message, she gets no information about x.

Let's understand with an example. If Eve finds out that the encrypted message is 10, she won't be able to deduce x because r could be designed for any 2 bit binary x string (00, 01, 10 and 11) because $00 \oplus 10{=}10$, $01 \oplus 11{=}10$, $10 \oplus 00{=}10$ and $11 \oplus 01{=}10$. Thus, there are multiple possible options for x if one only knows its encoded message.

The downside of the one-time pad is that it has to be discarded after one use - this is how it gets its name. This is because if Eve intercepts two encoded messages - $x \oplus r$ and $z \oplus r$ - for the two messages x and z, she could take the bitwise XOR of the two encoded messages to get $x \oplus z$, which might give her important information like - do the two messages begin or end the same (due to XOR properties) and if one message contains a long sequence of 0s, the corresponding part of the other message could be exposed ($b \oplus 0 = b$ where b is either 0 or 1).

Now, if Alice and Bob agree on a shared 128 bit random string r and it specifies a bijection from 128 bit strings to 128 bit strings along with the property that the encryption function can be used multiple times, then this kind of Symmetric Encryption scheme is called the Advanced Encryption Standard (AES). The only way to break the code of encryption at present is through brute force approach and that makes it highly unlikely to break the code if the shared random string r is selected correctly.

Now, we move onto Asymmetric Encryption. Consider the example we talked about at

the start. If Bob wants to communicate with hundreds of people securely using the private key encryption protocol, it wouldn't be too practical as he would have to maintain a detailed list of which key corresponds to whom. To resolve this issue, the concept of public key encryption was introduced. In this scheme, Bob gives a public key to everyone who sends him messages and keeps a private key to himself. The encryption is done through the public key and the decryption is done through the private key.
For public key encryption, we want a representation which can be encrypted fast but which has slow / almost impossible public key decryption and fast private key decryption. To enable this kind of public key encryption, an ingenious protocol called the Rivest, Shamir and Adleman (RSA) algorithm was formulated.
The RSA algorithm relies heavily on number theory. Think of the message that you want to send as numbers modulo N where messages larger than N can be broken into smaller pieces. The encryption function will be a one to one and onto mapping on $\{0, ...N-1\}$ and the decryption function will be the inverse mapping. Now, we move onto the two essential properties for the RSA algorithm. Pick two primes p and q such that $pq = N$. For any e relatively prime to $(p-1)(q-1)$:

1. The mapping $x \to x^e \mod N$ is a bijection on $\{0, ...N-1\}$.
   This property tells us that the mapping is a reasonable mapping for the encoding process of the message x, since no information is lost. Using this property, Bob can publish (N, e) as his public key.

2. Let d be the inverse of $e \mod (p-1)(q-1)$. Then for all $x \in \{0, ...N-1\}$, $(x^e)^d = x \mod N$.
   This property tells us how decryption can be achieved. The secret key, d, can be retained by Bob and he can simply use the formula given to decode his message.

Before moving onto the proof of the properties, let's look at an example of the RSA scheme: Let $N = 39 = 3 \cdot 13$. Choose $e = 5$ since it is relatively prime to 24 (= (3-1) $\cdot$ (13-1)). The decryption exponent is then the inverse of $(5 \mod 24) = 29$ (because $(5 \cdot 29) \bmod 24 = 1$). This can be computed using the Extended Euclid's Algorithm. Thus, for any message $x \mod 39$, the encryption of x is $y = x^5 \mod 39$ and the decryption of y is $x = y^{29} \mod 39$. For example, if $x = 11$, $y = 11^5 \mod 39 = 20$ and $x = 11 = 20^{29} \mod 39$.
Proof of the two properties:
If the mapping $x \to x^e \mod N$ is invertible, it must be a bijection. Thus, the second property implies the first property. We prove the second property. E is invertible modulo $(p-1)(q-1)$ because it is relatively prime to that number. Now, we examine the equation $(x^e)^d = x \mod N$. We observe that since $ed = 1 \mod (p-1)(q-1)$

(property of modulo inverse), $ed = 1 + k(p-1)(q-1)$ for some value of k. Now we need to show that the difference $(x^e)^d - x = x^{(1+k(p-1)(q-1))} - x$ is always $0 \mod N$. Now, we use Fermat's Little Theorem. It states that if p is a prime, then for every $1 \leq a \leq p$, $a^{(p-1)} = 1(\mod p)$. The second form of the equation is divisible by p (since $x^{p-1} = 1(\mod p)$) and likewise by q. Since p and q are primes, it must also be divisible by N (their product). Thus, $x^{(1+k(p-1)(q-1))} - x = 0(\mod N)$. This completes our proof.
Now, we will show how secure this scheme is:
The claim is as follows: Given N, e and $y = x^e \mod N$, it is computationally intractable to determine x. Why? Two reasons:

1. Finding x would require the brute force method of checking whether $x^e = y \mod N$ and this would take exponential time.

2. Factoring N to retrieve p and q and then figuring out d using the Extended Euclid's Algorithm would be a very hard process because factoring is assumed and proven to be hard.

Thus, intractability is what the RSA scheme takes advantage of to remain secure.

8 Machine Learning

One of the most exciting prospects in the realm of Computer Science is Machine Learning and Artificial Intelligence. How does a computer become intelligent? It starts with huge amounts of data being fed into computers. This data is somehow, over a considerable period of time, understood and learnt by the computer and this is followed by the computer developing some kind of intelligence. Although this is a more than simplified view of the whole process, it gives us a very basic understanding of the intuition behind the whole process.
Personally, the intermediate step of Machine Learning is the most intriguing. This, along with the high involvement of algorithms in Machine Learning, gave me the perfect opportunity to write about the basics of Machine Learning in this project.
I want to provide a basic overview of ML algorithms in this part of my project.
Before we begin, there are three types of ML algorithms: supervised learning algorithms, unsupervised learning algorithms and reinforcement learning algorithms.
Supervised learning algorithms usually consist of independent variables as input and dependent variables as output which are predicted from the input. The learning process continues until the model reaches a particular level of accuracy.
Unsupervised learning algorithms are usually used to cluster populations of variables / data in different groups for multiple plausible reasons.

Reinforcement learning algorithms are where the machine is exposed to a training environment where it continually makes decisions using trial and error after learning from past decisions. This improves the accuracy of its decision making.
Now, let's move onto an overview of some of the algorithms:

1. Linear Regression: This is an example of Supervised learning algorithms. Given a set of data points of the dependent and independent variable, we try forming an equation of the form $Y = mX + c$ where Y is the dependent variable, m is the slope of the line, X is the independent variable and c is the intercept. We derive the equation constants m and c by minimising the sum of the squared difference of distance between the data points and the regression line. This can be done as follows after reading the n data points: calculate four kinds of sums - $\sum X$, $\sum x^2$, $\sum Y$ and $\sum XY$. Then, calculate m by using the formula $m = (n \times \sum XY - \sum X \times \sum Y)/(n \times \sum X^2 - \sum X \times \sum X)$ and c by using the formula $(\sum Y - m \times \sum X)/n$.

2. Decision Tree: This is also an example of Supervised learning algorithms. In this algorithm, we look at some independent data points and classify them into two or more homogeneous sets. The main task is to make these sets as distinct as possible. There are two types of Decision Trees: categorical variable tree (definitive yes or no answers make up the tree. Ex: Will a person eat non vegetarian food?) and continuous variable tree (range of answers make up different sets in the tree. Ex: Human height). Let's look at one algorithm used for making Decision Trees called Gini algorithm: Let's say we have two input variables, gender and class, and we want to see which split would produce more homogeneous sub nodes for people that play cricket. Gini tells us to find the sum of squares of probability of success (p) and failure ($q = 1 - p$) for the first proposed partition and calculate the weighted Gini score for it. Do the same for the second proposed partition and the partition with higher Gini score is the better way to partition the data points. For example, if the data given to you is the following:

   (a) There are 30 students. 10 females and 20 males. 14 9^{th} graders and 16 10^{th} graders.

   (b) 2 females play cricket (0.2) and 13 males play cricket (0.65). 6 9^{th} graders play cricket (0.43) and 9 10^{th} graders play cricket (0.56).

   We need to find whether a split on gender would be better or a split on class would be better.
   For split on gender;

   (a) Gini for sub node females = (0.2)*(0.2) + (0.8)*(0.8) = 0.68

(b) Gini for sub node males = (0.65)*(0.65) + (0.35)*(0.35) = 0.55

(c) Calculated weighted Gini score = (10/30)*0.68 + (20/30)*0.55 = 0.59

For split on class;

(a) Gini for sub node 9^{th} graders = (0.43)*(0.43) + (0.57)*(0.57) = 0.51

(b) Gini for sub node 10^{th} graders = (0.56)*(0.56) + (0.44)*(0.44) = 0.51

(c) Calculated weighted Gini score = (14/30)*0.51 + (16/30)*0.51 = 0.51

We can see that the Gini score for split on gender is more favourable, so we make the node split based on gender.

3. K-means Clustering: This is one of the most widely used unsupervised ML algorithms. The objective of this algorithm is to group similar data points together and discover underlying patterns. To solve this, the algorithm searches for a fixed number of clusters (k) (cluster - collection of data points based on certain similarities) in a dataset. The algorithm gets its name because it finds k centroids in its approach. Essentially, what this algorithm does is given a set of data points, it divides the graph into multiple clusters by finding centroids for each cluster. A python library that can perform the task of finding the centroid is sklearn.cluster. However, if the data cannot be clustered efficiently, the data points need to be converted into points of a higher dimensions. Another approach of the algorithm is to generate k random centroids and assign each data point to its nearest centroid (by finding minimum Euclidean distance). After clustering all data points using the last step, we gather all points of a particular cluster and identify the actual centroid by taking the average of all points assigned to that cluster. We repeat this for all clusters. Now, we keep repeating the two previous steps until convergence is reached. One amazing application of K-Means clustering is to help an ecommerce giant like Samsung decide where it should open new stores in such a way that not too many stores are opened close to each other and the stores are not too far apart (for sales coverage reasons). Using the database which contains information about all the clients, Samsung can easily solve this problem using K-means clustering.