Algorithm Lab(CS2271)- Assignment 2

Made By- Ramesh Chandra Soren (2022CSB086)
Deep Saha(2022CSB078)
Santosh Methre(2022CSB084)

1. You need to implement the polygon triangulation problem.

The polygon triangulation problem involves dividing a given polygon into triangles with non-overlapping interior regions.

A polygon is a closed shape with straight sides, and triangulation involves partitioning this polygon into triangles.

Non-Overlapping Triangles: Triangles formed should not overlap with each other.

Interior Triangles: All triangles should have their vertices as the vertices of the original polygon and should lie completely inside the polygon.

Challenges in Polygon Triangulation:

1. Holes: Handling polygons with holes adds complexity to the triangulation process as the algorithm needs to account for the presence of holes within the polygon.

2. Degenerate Cases: Some polygons may have degenerate cases, such as collinear vertices or self-intersections, which require special handling to ensure a valid triangulation.

3. Efficiency: Achieving an efficient triangulation algorithm is crucial, especially for large or complex polygons, to minimize computation time and memory usage.

Applications of Polygon Triangulation:

Computer Graphics: Triangulation is essential for rendering complex shapes in computer graphics, as triangles are often the basic building blocks for 3D models.

Mesh Generation: Triangulation is used in mesh generation algorithms to create finite element meshes for simulations in engineering and physics.

Geographic Information Systems (GIS): Triangulation is employed in GIS for tasks like terrain modeling, map overlay analysis, and spatial data processing.

(a) Prepare a dataset for the convex polygon with (increasing) number of arbitrary vertices.

1. Theory of Convex Polygons:

- **Definition:** A polygon is convex if, for every pair of points within the polygon, the line segment connecting them lies entirely inside or on the boundary of the polygon.
- **Convexity Property:** In a convex polygon, the interior angles are all less than 180 degrees, and any line segment connecting two points inside the polygon lies entirely within the polygon.

2. Generating the Dataset:

- **Random Vertex Generation**: Start by randomly generating a set of vertices. Ensure that the vertices are unique and do not lie on the same line, as this would create a non-convex polygon.
- **Convex Hull**: Construct the convex hull of the generated vertices. The convex hull is the smallest convex polygon that contains all the given points.
- **Increasing Number of Vertices**: Generate convex polygons with an increasing number of vertices. Start with a minimum number of vertices and gradually increase it to create a dataset with varying polygon sizes.
- **Ensure Convexity**: After adding each vertex, check if the newly formed polygon remains convex. If not, adjust the vertices to maintain convexity.

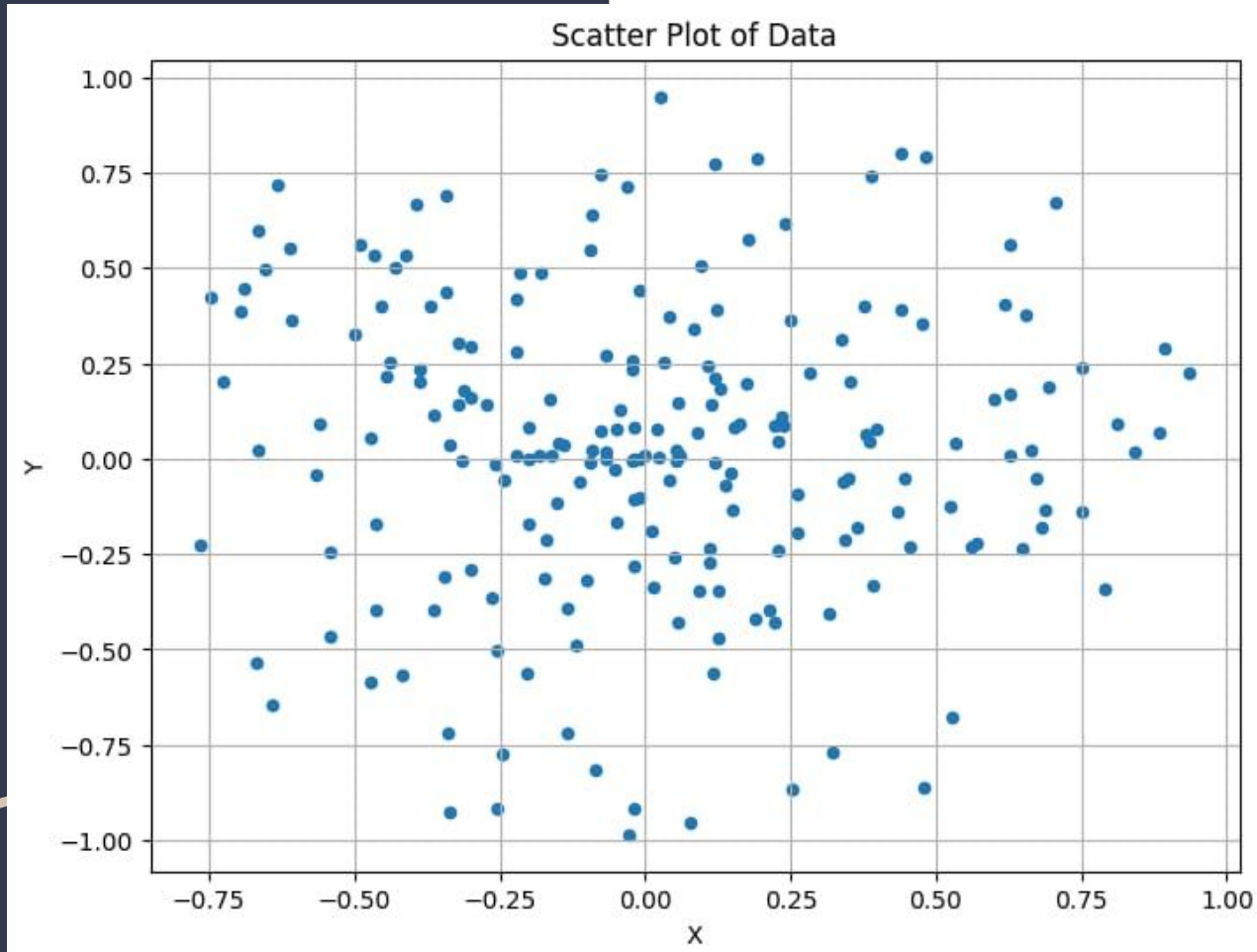
3. Dataset Format:

- Each entry in the dataset consists of:
  - The number of vertices of the convex polygon.
  - Coordinates of each vertex (x, y).
  - Additional attributes or labels if required for your analysis or application.

4. Example Dataset:

- **Number of Vertices**: 4
  - Vertices: (1, 1), (2, 4), (5, 5), (4, 2)
- **Number of Vertices**: 5
  - Vertices: (2, 2), (3, 5), (6, 4), (5, 1), (4, 3)
- **Number of Vertices**: 6
  - Vertices: (1, 2), (3, 1), (5, 2), (6, 4), (4, 5), (2, 4)

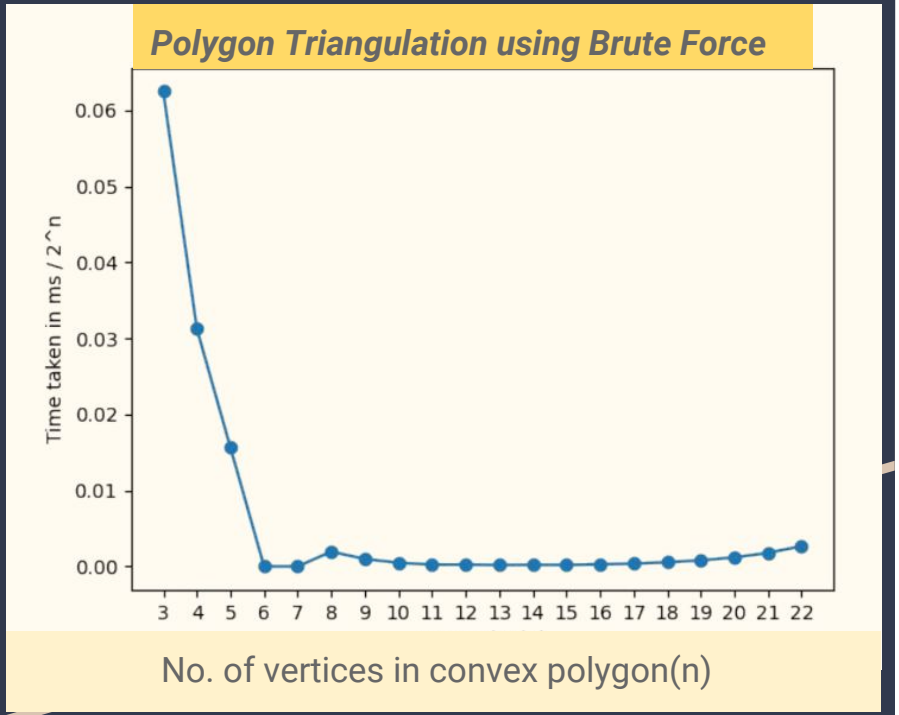
```c
#include <stdio.h>
#include <stdbool.h>
typedef struct {
    double x;
    double y;
} Point;

// Function to compute the cross product of two vectors
int cross_product(Point p1, Point p2, Point p3) {
    int dx1 = p2.x - p1.x;
    int dy1 = p2.y - p1.y;
    int dx2 = p3.x - p2.x;
    int dy2 = p3.y - p2.y;

    return dx1 * dy2 - dx2 * dy1;
}
// Function to check if a polygon is convex
bool isConvex(Point polygon[], int num_vertices) {
    bool has_positive = false;
    bool has_negative = false;

    for (int i = 0; i < num_vertices; i++) {
        Point p1 = polygon[i];
        Point p2 = polygon[(i + 1) % num_vertices];
        Point p3 = polygon[(i + 2) % num_vertices];

        int cross = cross_product(p1, p2, p3);

        if (cross > 0) {
            has_positive = true;
        } else if (cross < 0) {
            has_negative = true;
        }
        // If both positive and negative cross products are found, polygon is not convex
        if (has_positive && has_negative) {
            return false;
        }
    }
    // If either positive or negative cross product is missing, polygon is convex
    return true;
```


Scatter Plot of Data
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Y
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
X

(b) Consider a brute force method where you do an exhaustive search for finding the optimal result.

The complexity of polygon triangulation using the brute force method can be analyzed based on the number of possible triangulations for a given polygon.

1. **Number of Triangulations:**
   - For a polygon with n vertices, the number of possible triangulations can be quite large.
   - The Catalan number, Cn, represents the number of possible triangulations for a convex polygon with n vertices. It is given by the formula Cn = (2n)! / [(n + 1)! * n!].
   - For example, for a polygon with 4 vertices, there is only 1 possible triangulation, while for a polygon with 6 vertices, there are 5 possible triangulations.
2. **Brute Force Approach:**
   - In the brute force method, we would enumerate all possible triangulations for the given polygon.
   - This involves considering every possible combination of non-overlapping triangles that partition the polygon.
   - The algorithm would generate all possible sets of non-overlapping triangles and evaluate each set to determine if it forms a valid triangulation.



3.<u>Complexity Analysis:</u>

- The time complexity of the brute force method for polygon triangulation is exponential, as it involves generating and evaluating all possible triangulations.
- Specifically, the time complexity is O(Cn), where Cn is the nth Catalan number representing the number of possible triangulations for a polygon with n vertices.
- Since the Catalan numbers grow exponentially with the number of vertices, the brute force method becomes increasingly impractical for polygons with a large number of vertices.

4.<u>Space Complexity:</u>

- The space complexity primarily depends on the storage of vertices and diagonals, requiring O(n) space for the polygon representation.

```
hm/question1$ ./a.out
Cost for points: 3.073993
Cost for points: 6.163367
Cost for points: 3.922428
Cost for points: 4.750324
Cost for points: 6.041372
Cost for points: 12.537194
Cost for points: 8.232381
Cost for points: 10.441747
Cost for points: 10.802883
```

This is brute force implementation of polygon triangulation. We can see that after 9 set of vertices it got stuck.

Thus we can conclude that Brute force approach is very costly and nonoptimal.

(c) Next apply dynamic programming, in line with the matrix chain multiplication problem.

Applying dynamic programming to solve the polygon triangulation problem involves breaking down the problem into smaller subproblems and efficiently solving each subproblem to find the optimal solution.

Problem Statement: Given a polygon with n vertices, the goal is to find the minimum number of triangles required to triangulate the polygon while ensuring that no two triangles intersect or overlap.

Identifying Subproblems: Subproblem Definition: Let T(i, j) represent the minimum number of triangles required to triangulate the sub polygon formed by vertices Vi, Vi+1, ..., Vj.

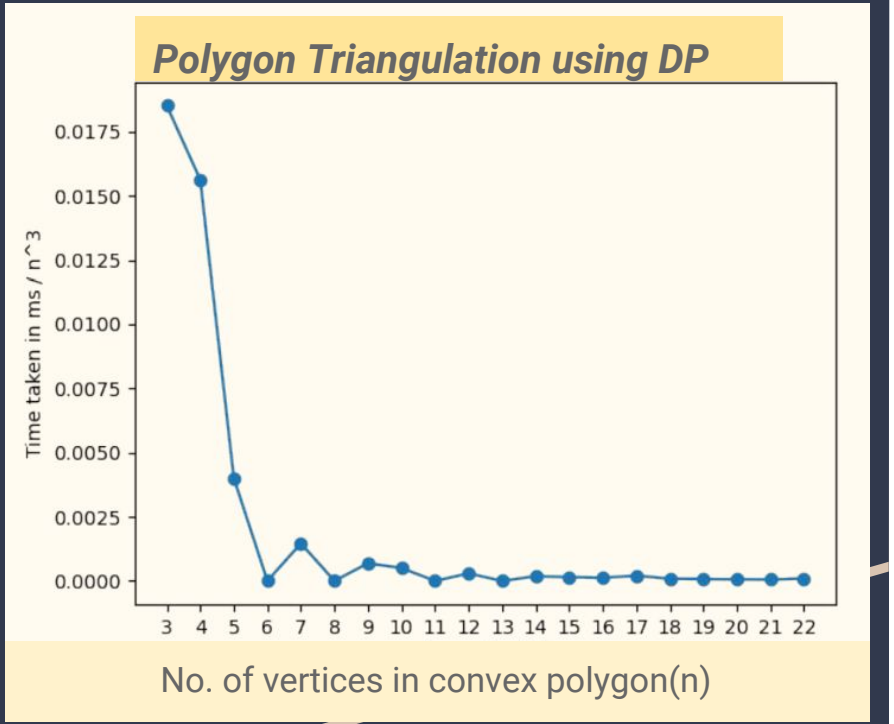
Recurrence Relation: To solve the problem recursively, we establish a recurrence relation based on the subproblems:

$T(i,j)=\min_{i\le k<j}\{T(i,k)+T(k+1,j)+1\}T(i,j)=\min_{i\le k<j}\{T(i,k)+T(k+1,j)+1\}$

This recurrence relation states that to triangulate the sub polygon from vertex i to vertex j, we split it into two smaller sub polygons at vertex k (where $i \le k < j$), triangulate each sub polygon recursively, and add 1 for the new triangle formed.

Base Cases: Base cases are needed to initialize the dynamic programming table:

- When i = j, T(i, j) = 0 since there are no triangles to triangulate a single vertex.
- When j = i + 1, T(i, j) = 0 since there are no triangles to triangulate a line segment.



We fill the DP table bottom-up, starting from smaller subproblems and gradually building up to the solution for the original problem.

The optimal solution to the original problem is found at T(1, n), which represents the minimum number of triangles required to triangulate the entire polygon.

The time complexity of the dynamic programming approach for polygon triangulation is O(n^3), where n is the number of vertices in the polygon. This is because there are O(n^2) subproblems to compute, and each subproblem involves examining O(n) choices for the split point k.

(d) Then implement a greedy strategy to solve the same problem by choosing non-intersecting diagonals in sorted order.

The greedy strategy for polygon triangulation involves iteratively selecting non-intersecting diagonals of the polygon in a sorted order until the entire polygon is triangulated. Here's the theory behind this approach:

Definition of Greedy Strategy: Greedy algorithms make locally optimal choices at each step with the hope of finding a global optimum. In the context of polygon triangulation, the greedy strategy selects diagonals one by one, making the locally optimal choice of the next diagonal to add.

Selection Criteria: The key aspect of the greedy strategy is determining the criteria for selecting the next diagonal. In this case, we choose diagonals that do not intersect with any existing diagonals, ensuring that the triangulation remains valid.

Sorting Diagonals: Before applying the greedy strategy, diagonals are sorted based on some criteria, such as the length of the diagonal or the angle it forms with a reference axis. Sorting ensures that we consider diagonals in a consistent order, facilitating the selection process.

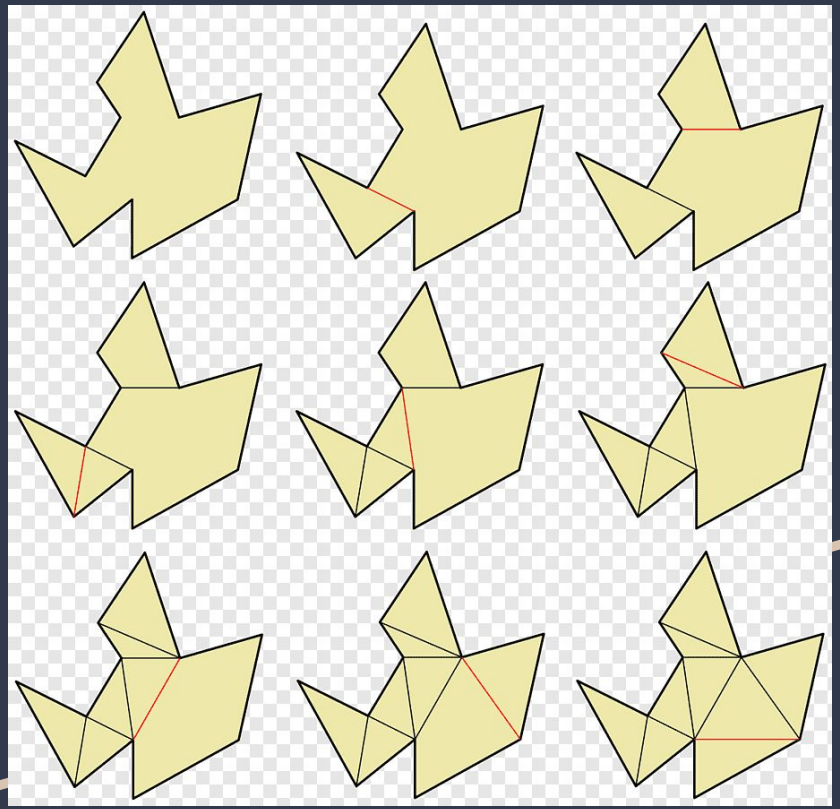
Greedy Triangulation Process:

- Start with an empty set of diagonals.
- Sort all possible diagonals of the polygon.
- Iterate through the sorted list of diagonals.
- For each diagonal, check if it intersects with any diagonals already selected. If not, add it to the set of selected diagonals.
- Continue this process until the entire polygon is triangulated.

At each step, it's essential to check whether adding a diagonal would result in intersecting diagonals. If so, the diagonal should not be added to ensure that the resulting triangulation remains valid.

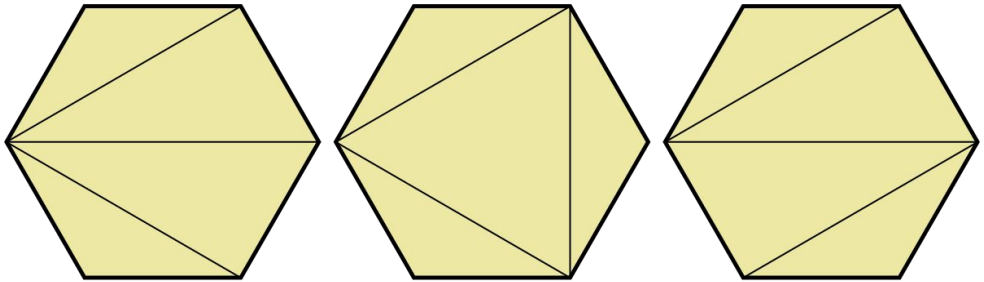
Optimality and Correctness:
While the greedy strategy does not guarantee an optimal solution in all cases, it often provides reasonably good solutions efficiently. However, the correctness of the triangulation depends on the validity check performed at each step to ensure non-intersecting diagonals.

Time Complexity: The time complexity of the greedy strategy for polygon triangulation depends on the sorting algorithm used and the complexity of the intersection check. In the worst case, where all diagonals need to be checked for intersections, the time complexity is O(n^2), where n is the number of vertices in the polygon.



Delaunay triangulation is generally considered an incremental algorithm rather than dynamic.

Fan triangulation is more related to computational geometry and geometric algorithms rather than dynamic programming.

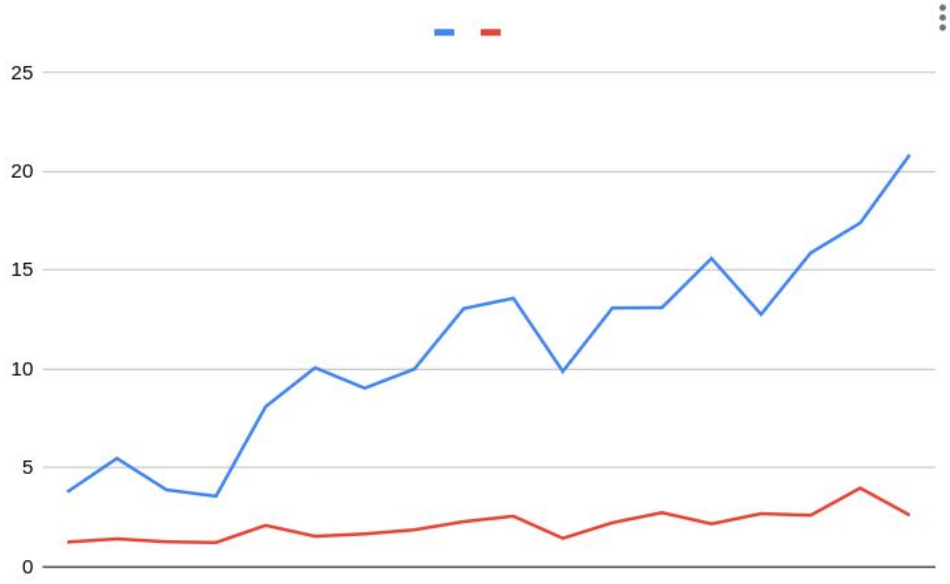


Fan triangulation

Greedy triangulation

Delaunay triangulation

(e) Finally compare the results and suggest whether the DP and Greedy approach results show mismatch.



Blue line show the triangulation length of polygon using Dynamic programming.

Red line shows the polygon triangulation length using Greedy approach.

We can clearly see that there is a mismatch b/w both these results. Greedy is not giving the correct answer.

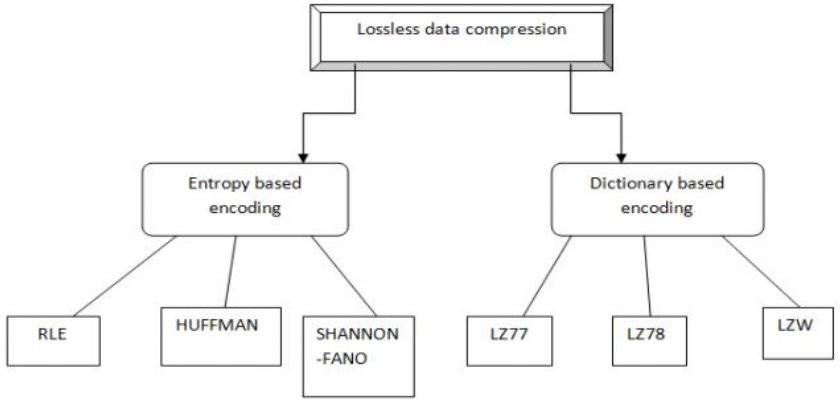
2. Implement data compression strategies:

- Lossless compression
- Lossy compression

Lossless Compression:

Lossless compression algorithms ensure that the original data can be perfectly reconstructed from the compressed data. This type of compression is suitable for scenarios where maintaining every detail of the data is crucial. Some common lossless compression strategies include:

- Run-Length Encoding (RLE)
- Huffman Coding
- Lempel-Ziv (LZ) Compression



Lossy compression :------>

Lossy compression algorithms sacrifice some amount of data fidelity in exchange for higher compression ratios. This type of compression is commonly used for multimedia data, such as images, audio, and video, where minor losses in quality are acceptable. Some prominent lossy compression strategies include

- Discrete Cosine Transform (DCT)
- Quantization
- Wavelet Transform

(a) Write encoding algorithms for the Shannon-Fano coding scheme.

Algorithm for Constructing Shannon-Fano Code:

- Sort the symbols based on their probabilities (or frequencies) in non-increasing order.
- Divide the symbols into two sets such that the total probability of one set is approximately equal to the total probability of the other set. This division can be done recursively until each set contains only one symbol.
- Assign binary codes to the symbols, with one set assigned '0' as the prefix and the other set assigned '1' as the prefix. This ensures that no code is a prefix of another, making it a prefix-free code.
- Recursively apply the above steps to each subset until each symbol is assigned a binary code.

Shannon-Fano coding scheme

Encoding Algorithm:

- Using the Shannon-Fano code generated for each symbol, encode the input data symbol by symbol.
- Replace each symbol with its corresponding Shannon-Fano code.
- Concatenate the encoded codes to generate the compressed data.

```c
// Function to find Shannon code
void shannon(int l, int h, node p[]) {
    float pack1 = 0, pack2 = 0, diff1 = 0, diff2 = 0;
    int i, d, k, j;
    if ((l + 1) == h || l == h || l > h) {
        if (l == h || l > h)
            return;
        p[h].arr[++(p[h].top)] = 0;
        p[l].arr[++(p[l].top)] = 1;
        return;
    } else {
        for (i = l; i <= h - 1; i++)
            pack1 = pack1 + p[i].pro;
        pack2 = pack2 + p[h].pro;
        diff1 = pack1 - pack2;
        if (diff1 < 0)
            diff1 = diff1 * -1;
        j = 2;
        while (j != h - l + 1) {
            k = h - j;
            pack1 = pack2 = 0;
            for (i = l; i <= k; i++)
                pack1 = pack1 + p[i].pro;
            for (i = h; i > k; i--)
                pack2 = pack2 + p[i].pro;
            diff2 = pack1 - pack2;
            if (diff2 < 0)
                diff2 = diff2 * -1;
            if (diff2 >= diff1)
                break;
            diff1 = diff2;
            j++;
        }
        k++;
        for (i = l; i <= k; i++)
            p[i].arr[++(p[i].top)] = 1;
        for (i = k + 1; i <= h; i++)
            p[i].arr[++(p[i].top)] = 0;

        // Invoke shannon function
        shannon(l, k, p);
        shannon(k + 1, h, p);
    }
}
```

```c
// Function to sort the symbols
// based on their probability or frequency
void sortByProbability(int n, node p[]) {
    int i, j;
    node temp;
    for (j = 1; j <= n - 1; j++) {
        for (i = 0; i < n - 1; i++) {
            if ((p[i].pro) > (p[i + 1].pro)) {
                temp.pro = p[i].pro;
                temp.sym = p[i].sym;

                p[i].pro = p[i + 1].pro;
                p[i].sym = p[i + 1].sym;

                p[i + 1].pro = temp.pro;
                p[i + 1].sym = temp.sym;
            }
        }
    }
}

// Function to display Shannon codes
void display(int n, node p[]) {
    int i, j;
    printf("\n\n\n\tSymbol\tProbability\tCode");
    for (i = n - 1; i >= 0; i--) {
        printf("\n\t%c\t\t%.2f\t", p[i].sym, p[i].pro);
        for (j = 0; j <= p[i].top; j++)
            printf("%d", p[i].arr[j]);
    }
}
```

Calculation of Shannon-Fano code:

Symbol	p_i	l_i	Sum from p_i to i-1	Sum to p_i binary	Code
S_1	0.36	2	0.0	0.00000…	00
S_2	0.18	3	0.36	0.01011…	010
S_3	0.18	3	0.54	0.10001…	100
S_4	0.12	3	0.72	0.10111…	101
S_5	0.09	4	0.84	0.11010…	1101
S_6	0.07	4	0.93	0.11101…	1110

Output of my code:-

```
Enter number of symbols : 5
Enter symbol 1 : A
Enter symbol 2 : B
Enter symbol 3 : C
Enter symbol 4 : D
Enter symbol 5 : E

Enter probability of A : 0.22

Enter probability of B : 0.28

Enter probability of C : 0.15

Enter probability of D : 0.30

Enter probability of E : 0.05



        Symbol  Probability     Code
        D               0.30    00
        B               0.28    01
        A               0.22    10
```

(b) Implement Huffman encoding scheme using heap as data structure.

Benefits of Huffman Encoding:

1. Provides efficient compression by assigning shorter codes to frequently occurring characters.

2. Simple and widely used in various applications like file compression, data transmission, etc.

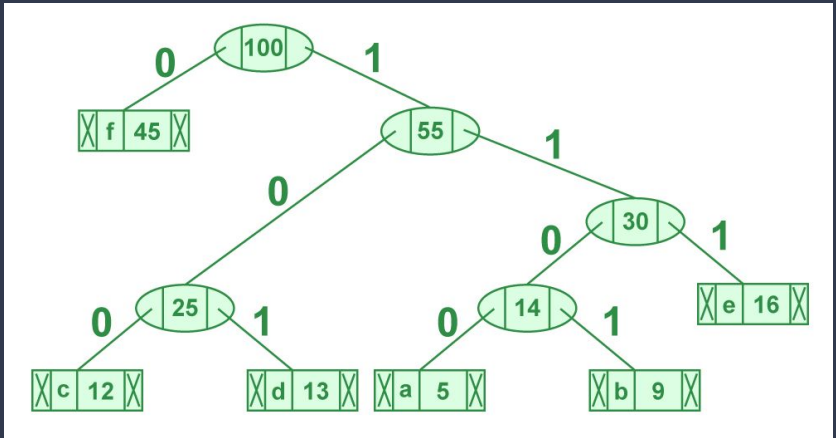
3. Lossless compression ensures no data loss during compression and decompression.

Min Heap Implementation : The min heap is used to efficiently select the nodes with the lowest frequencies during the construction

Huffman Tree Construction :It repeatedly extracts two nodes with the lowest frequencies from the heap, creates a new internal node with a frequency equal to the sum of the frequencies of the extracted nodes, and then inserts the new node back into the heap. This process continues until only one node remains in the heap, which becomes the root of the Huffman tree

Printing Huffman Codes :The `printCodes` function recursively traverses the Huffman tree to generate and print Huffman codes for each character. It traverses left branches with '0' and right branches with '1', appending the appropriate bits to the code array until it reaches a leaf node (representing a character), at which point it prints the character and its corresponding code.

Huffman tree diagram



Huffman Encoding Overview: Huffman encoding is a widely used technique for lossless data compression.

It assigns variable-length codes to input characters, with shorter codes assigned to more frequent characters, optimizing the overall compression.

- **struct Node**: Represents a node in the Huffman tree, containing a symbol (character), frequency count, and pointers to left and right child nodes.
- **struct MinHeap**: Represents a min-heap data structure used to build the Huffman tree, where the nodes are ordered based on their frequencies.
- **createNode**: Creates a new node with a given symbol and frequency.
- **createMinHeap**: Creates an empty min-heap.
- **minHeapify**: Maintains the min-heap property by recursively fixing the heap from a given index.
- **buildMinHeap**: Builds a min-heap from an array of nodes.
- **insertMinHeap**: Inserts a new node into the min-heap while maintaining the heap property.
- **extractMin**: Extracts the node with the minimum frequency from the min-heap.
- **buildHuffmanTree**: Builds the Huffman tree using a min-heap of nodes based on character frequencies.
- **printCodes**: Prints the Huffman codes for each character in the tree.
- **HuffmanCodes**: Calculates the Huffman codes for a given input text.
- **calculateFrequencies**: Calculates the frequency of each character in a file.
- **getCode**: Retrieves the Huffman code for a given character in the Huffman tree.
- **writeEncodedText**: Writes the encoded text to an output file using the Huffman codes.
- **calculateCompressionRatio**: Calculates the compression ratio of the original and compressed files.

```c
struct Node {
    char symbol;
    int frequency;
    struct Node *left, *right;
};

struct MinHeap {
    int size;
    struct Node *array[MAX_SYMBOLS];
};

struct Node *createNode(char symbol, int frequency) {
    struct Node *node = (struct Node *)malloc(sizeof(struct Node));
    node->symbol = symbol;
    node->frequency = frequency;
    node->left = node->right = NULL;
    return node;
}

struct MinHeap *createMinHeap() {
    struct MinHeap *minHeap = (struct MinHeap *)malloc(sizeof(struct MinHeap));
    minHeap->size = 0;
    return minHeap;
}

void swapNodes(struct Node **a, struct Node **b) {
    struct Node *temp = *a;
    *a = *b;
    *b = temp;
}
```

Prefix Rule:

Huffman Coding implements a rule known as a prefix rule.

This is to prevent the ambiguities while decoding.

It ensures that the code assigned to any character is not a prefix of the code assigned to any other character.

```c
void HuffmanCodes(char *text) {
    int frequencies[MAX_SYMBOLS] = {0};
    struct MinHeap *minHeap = buildHuffmanTree(text, frequencies);
    struct Node *root = minHeap->array[0];

    int codes[MAX_SYMBOLS];
    int top = 0;
    printf("Huffman Codes:\n");
    printCodes(root, codes, top);
}

int main() {
    char text[] = "hello world";
    HuffmanCodes(text);
    return 0;
}
```

```c
void minHeapify(struct MinHeap *minHeap, int index) {
    int smallest = index;
    int left = 2 * index + 1;
    int right = 2 * index + 2;

    if (left < minHeap->size && minHeap->array[left]->frequency < minHeap->array[smallest]->frequency)
        smallest = left;

    if (right < minHeap->size && minHeap->array[right]->frequency < minHeap->array[smallest]->frequency)
        smallest = right;

    if (smallest != index) {
        swapNodes(&minHeap->array[smallest], &minHeap->array[index]);
        minHeapify(minHeap, smallest);
    }
}

void buildMinHeap(struct MinHeap *minHeap) {
    int n = minHeap->size - 1;
    for (int i = (n - 1) / 2; i >= 0; i--) {
        minHeapify(minHeap, i);
    }
}

void insertMinHeap(struct MinHeap *minHeap, struct Node *node) {
    minHeap->size++;
    int i = minHeap->size - 1;
    while (i && node->frequency < minHeap->array[(i - 1) / 2]->frequency) {
        minHeap->array[i] = minHeap->array[(i - 1) / 2];
        i = (i - 1) / 2;
    }
    minHeap->array[i] = node;
}

struct Node *extractMin(struct MinHeap *minHeap) {
    struct Node *temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeapify(minHeap, 0);
    return temp;
}
```

Output of Huffman code:

```
Huffman Codes:
o: 00
h: 010
e: 011
l: 10
 : 1100
w: 1101
r: 1110
```

```c
struct MinHeap *buildHuffmanTree(char *text, int *frequencies) {
    for (int i = 0; i < strlen(text); i++) {
        frequencies[(int)text[i]]++;
    }

    struct MinHeap *minHeap = createMinHeap();
    for (int i = 0; i < MAX_SYMBOLS; i++) {
        if (frequencies[i] > 0) {
            insertMinHeap(minHeap, createNode((char)i, frequencies[i]));
        }
    }

    while (minHeap->size != 1) {
        struct Node *left = extractMin(minHeap);
        struct Node *right = extractMin(minHeap);
        struct Node *top = createNode('$', left->frequency + right->frequency);
        top->left = left;
        top->right = right;
        insertMinHeap(minHeap, top);
    }

    return minHeap;
}

void printCodes(struct Node *root, int *codes, int top) {
    if (root->left) {
        codes[top] = 0;
        printCodes(root->left, codes, top + 1);
    }
    if (root->right) {
        codes[top] = 1;
        printCodes(root->right, codes, top + 1);
    }
    if (!root->left && !root->right) {
        printf("%c: ", root->symbol);
        for (int i = 0; i < top; i++) {
            printf("%d", codes[i]);
        }
        printf("\n");
    }
}
```

(c) Implement an instantaneous decoding algorithm

```c
typedef struct {
  char code[MAX_CODE_LEN];
  char character;
} CodeTableEntry;

// Function to build the code table from the input
int buildCodeTable(CodeTableEntry table[], char characters[], char codes[][MAX_CODE_LEN], int num_codes) {
  for (int i = 0; i < num_codes; i++) {
    strcpy(table[i].code, codes[i]);
    table[i].character = characters[i];
  }
  return 1;
}

// Function to decode the encoded string
char* decodeString(CodeTableEntry table[], char encoded_string[], int num_codes) {
  int curr_pos = 0;
  int decoded_index = 0;
  char* decoded_string = (char*)malloc(strlen(encoded_string) + 1); // Allocate space for decoded string

  while (encoded_string[curr_pos] != '\0') {
    int found = 0;
    for (int i = 0; i < num_codes; i++) {
      // Check if the current encoded substring matches a code in the table
      if (strncmp(encoded_string + curr_pos, table[i].code, strlen(table[i].code)) == 0) {
        decoded_string[decoded_index++] = table[i].character;
        curr_pos += strlen(table[i].code);  // Move to next character
        found = 1;
        break;
      }
    }
    if (!found) {
      // If no match found, handle error (invalid code)
      printf("Error: Invalid code encountered during decoding.\n");
      return NULL;
    }
  }

  decoded_string[decoded_index] = '\0'; // Add null terminator
  return decoded_string;
}
```

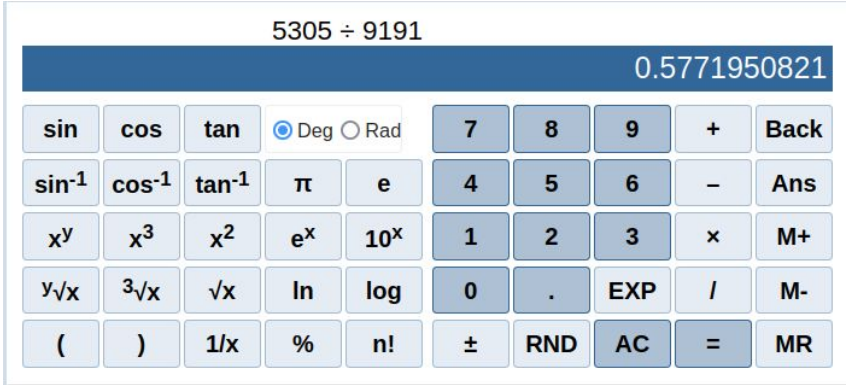
```c
int main() {
  char characters[] = {'a', 'b', 'c', 'd'}; // Replace with actual characters
  char codes[][MAX_CODE_LEN] = {"00", "10", "01", "11"}; // Replace with actual codes (ensure prefix property)
  int num_codes = sizeof(characters) / sizeof(characters[0]);

  CodeTableEntry table[MAX_CHAR];
  buildCodeTable(table, characters, codes, num_codes);

  char encoded_string[] = "00100111"; // Replace with encoded string

  char* decoded_string = decodeString(table, encoded_string, num_codes);
  if (decoded_string != NULL) {
    printf("Decoded String: %s\n", decoded_string);
    free(decoded_string); // Free memory allocated for decoded string
  }

  return 0;
}
```

Output of my code:

Decoded String: abcd

(d) Choose a large text document and get results of Huffman coding with this document.

Output of my code: Compression ratio: 0.58




```
total 68
-rw-rw-r-- 1 ramesh ramesh  3562 Mar 10 19:17 2a.c
-rw-rw-r-- 1 ramesh ramesh  3539 Mar  9 14:19 2b.c
-rw-rw-r-- 1 ramesh ramesh  2184 Mar 10 19:23 2c.c
-rwxrwxr-x 1 ramesh ramesh 16656 Mar 17 15:11 a.out
-rw-rw-r-- 1 ramesh ramesh  5305 Mar 17 15:11 compressed.bin
-rw-rw-r-- 1 ramesh ramesh  9191 Mar 17 15:09 document.txt
-rw-rw-r-- 1 ramesh ramesh  6792 Mar 17 15:11 huffmaan.c
-rw-rw-r-- 1 ramesh ramesh  5314 Mar 17 14:56 shannon.c
```

Calculation of ratio:

```
float calculateCompressionRatio(FILE *originalFile, FILE *compressedFile) {
    fseek(originalFile, 0L, SEEK_END);
    long originalSize = ftell(originalFile);
    fseek(compressedFile, 0L, SEEK_END);
    long compressedSize = ftell(compressedFile);
    return (float)compressedSize / originalSize;
}
```

(e) Examine the optimality of Huffman codes as data compression strategy by comparing with another document.

Shannon-Fano and Huffman encodings are both efficient methods for encoding text files, but they have distinct characteristics.

Shannon-Fano encoding involves fixed-length inputs and variable-length outputs, while Huffman encoding works with variable-length inputs and outputs.

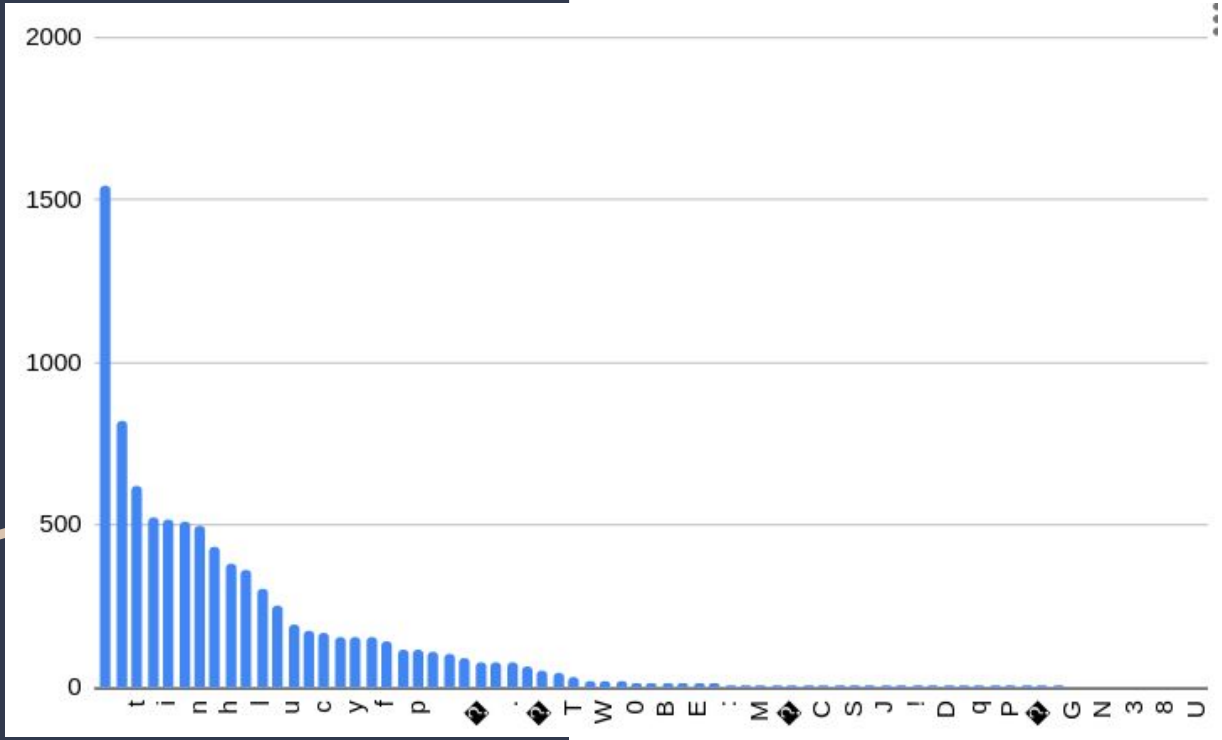
In terms of efficiency, Huffman coding is generally preferred over Shannon-Fano coding.

Huffman coding achieves a more optimal average code length that is asymptotically equal to the entropy of the source, making it highly efficient in terms of compression.

On the other hand, Shannon-Fano coding does not always achieve the same level of efficiency as Huffman coding.

Therefore, when considering which encoding method is more efficient for text file compression, Huffman coding is often the preferred choice due to its ability to provide better compression ratios and performance.

Frequency of characters in my text document



Thank You