```
#include<bits/stdc++.h>
using namespace std;
string encryptedText, decryptedText;
void encrypt(string s, int index, int k, string res){
if(index == s.size()){
encryptedText = res;
return;
}
if (s[index] >= 'A' && s[index] <= 'Z'){
res += (s[index] + k - 'A') % 26 + 'A';
encrypt(s, index + 1, k + 1, res);
}
else{
res += (s[index] + k - 'a') % 26 + 'a';
encrypt(s, index + 1, k + 1, res);
}
}
void decrypt(string s, int index, int k, string res){
if(index == s.size()){
decryptedText = res;
return;
}
if (s[index] >= 'A' && s[index] <= 'Z'){
res += (s[index] - k - 'A' + 26) % 26 + 'A';
decrypt(s, index + 1, k + 1, res);
}
else{
res += (s[index] - k - 'a' + 26) % 26 + 'a';
decrypt(s, index + 1, k + 1, res);
}
}
int main()
{
string s;
int k;
cout << "Enter string: ";
cin >> s >> k;
encrypt(s, 0, k, "");
cout << "Encrypted Text: " << encryptedText << endl;
```

decrypt(encryptedText, 0, k, "");
cout << "Decrypted Text: " << decryptedText << endl;
return 0;
}array-based problems using the divide and conquer approach . generate 10 problem like this and solution and also the test case

Looking at your Caesar cipher implementation with incrementing keys, I can see you're working with recursive algorithms. Here are **10 array-based divide and conquer problems** with complete solutions and test cases that will help strengthen your understanding of this approach:

1. Merge Sort

```
#include<bits/stdc++.h>
using namespace std;

void merge(vector<int>& arr, int left, int mid, int right) {
    vector<int> temp(right - left + 1);
    int i = left, j = mid + 1, k = 0;

    while(i <= mid && j <= right) {
        if(arr[i] <= arr[j]) temp[k++] = arr[i++];
        else temp[k++] = arr[j++];
    }
    while(i <= mid) temp[k++] = arr[i++];
    while(j <= right) temp[k++] = arr[j++];

    for(int i = 0; i < k; i++) arr[left + i] = temp[i];
}

void mergeSort(vector<int>& arr, int left, int right) {
    if(left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

int main() {
    vector<int> arr = {64, 34, 25, 12, 22, 11, 90};
    mergeSort(arr, 0, arr.size() - 1);
    for(int x : arr) cout << x << " ";
    return 0;
}
```

Test Cases:

- Input: `→ Output:` [1] [2]
- Input: `→ Output:` [3] [4] [5] [6] [7]

2. Quick Sort

```
#include<bits/stdc++.h>
using namespace std;

int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for(int j = low; j < high; j++) {
        if(arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(vector<int>& arr, int low, int high) {
    if(low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    vector<int> arr = {10, 7, 8, 9, 1, 5};
    quickSort(arr, 0, arr.size() - 1);
    for(int x : arr) cout << x << " ";
    return 0;
}
```

Test Cases:

- Input: `→ Output:` [7] [8] [9] [10] [11] [3]
- Input: `→ Output:` [4] [5] [12] [9] [11] [3]

3. Binary Search

```
#include<bits/stdc++.h>
using namespace std;

int binarySearch(vector<int>& arr, int left, int right, int target) {
    if(left <= right) {
        int mid = left + (right - left) / 2;

        if(arr[mid] == target) return mid;
        if(arr[mid] > target) return binarySearch(arr, left, mid - 1, target);
        return binarySearch(arr, mid + 1, right, target);
    }
    return -1;
```

```
}

int main() {
    vector<int> arr = {2, 3, 4, 10, 40};
    int target = 10;
    int result = binarySearch(arr, 0, arr.size() - 1, target);
    if(result != -1) cout << "Element found at index " << result << endl;
    else cout << "Element not found" << endl;
    return 0;
}
```

Test Cases:

- Input: `arr=, target=10` → Output: `3` [5] [6] [11] [4]
- Input: `arr=, target=6` → Output: `-1` [6] [3] [4] [5] [7]

4. Maximum Subarray Sum (Divide & Conquer)

```
#include<bits/stdc++.h>
using namespace std;

int maxCrossingSum(vector<int>& arr, int left, int mid, int right) {
    int leftSum = INT_MIN, sum = 0;
    for(int i = mid; i >= left; i--) {
        sum += arr[i];
        leftSum = max(leftSum, sum);
    }

    int rightSum = INT_MIN;
    sum = 0;
    for(int i = mid + 1; i <= right; i++) {
        sum += arr[i];
        rightSum = max(rightSum, sum);
    }

    return leftSum + rightSum;
}

int maxSubarraySum(vector<int>& arr, int left, int right) {
    if(left == right) return arr[left];

    int mid = left + (right - left) / 2;

    return max({maxSubarraySum(arr, left, mid),
                maxSubarraySum(arr, mid + 1, right),
                maxCrossingSum(arr, left, mid, right)});
}

int main() {
    vector<int> arr = {-2, -3, 4, -1, -2, 1, 5, -3};
    int result = maxSubarraySum(arr, 0, arr.size() - 1);
    cout << "Maximum subarray sum: " << result << endl;
```

```
    return 0;
}
```

Test Cases:

- Input: `[-2, -3, 4, -1, -2, 1, 5, -3]` → Output: `7`
- Input: `` → Output: `15`[3] [4] [5] [6] [7]

5. Count Inversions

```
#include<bits/stdc++.h>
using namespace std;

long long mergeAndCount(vector<int>& arr, int left, int mid, int right) {
    vector<int> temp(right - left + 1);
    int i = left, j = mid + 1, k = 0;
    long long invCount = 0;

    while(i <= mid && j <= right) {
        if(arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
            invCount += (mid - i + 1);
        }
    }

    while(i <= mid) temp[k++] = arr[i++];
    while(j <= right) temp[k++] = arr[j++];

    for(int i = 0; i < k; i++) arr[left + i] = temp[i];
    return invCount;
}

long long mergeSortAndCount(vector<int>& arr, int left, int right) {
    long long invCount = 0;
    if(left < right) {
        int mid = left + (right - left) / 2;
        invCount += mergeSortAndCount(arr, left, mid);
        invCount += mergeSortAndCount(arr, mid + 1, right);
        invCount += mergeAndCount(arr, left, mid, right);
    }
    return invCount;
}

int main() {
    vector<int> arr = {8, 4, 2, 1};
    long long result = mergeSortAndCount(arr, 0, arr.size() - 1);
    cout << "Number of inversions: " << result << endl;
    return 0;
}
```

Test Cases:

- Input: `` → Output: `6` [9] [4] [6] [3]
- Input: `` → Output: `0` [4] [5] [6] [7] [3]

6. Find Peak Element

```
#include<bits/stdc++.h>
using namespace std;

int findPeak(vector<int>& arr, int left, int right) {
    if(left == right) return left;

    int mid = left + (right - left) / 2;

    if(arr[mid] > arr[mid + 1]) {
        return findPeak(arr, left, mid);
    } else {
        return findPeak(arr, mid + 1, right);
    }
}

int main() {
    vector<int> arr = {1, 3, 20, 4, 1, 0};
    int peakIndex = findPeak(arr, 0, arr.size() - 1);
    cout << "Peak element is " << arr[peakIndex] << " at index " << peakIndex << endl;
    return 0;
}
```

Test Cases:

- Input: `` → Output: `Peak at index 2 (value 20)` [5] [6] [3]
- Input: `` → Output: `Peak at index 2 (value 20)` [11] [13] [7]

7. Majority Element

```
#include<bits/stdc++.h>
using namespace std;

int countInRange(vector<int>& arr, int num, int left, int right) {
    int count = 0;
    for(int i = left; i <= right; i++) {
        if(arr[i] == num) count++;
    }
    return count;
}

int majorityElementRec(vector<int>& arr, int left, int right) {
    if(left == right) return arr[left];

    int mid = left + (right - left) / 2;

    int leftMajority = majorityElementRec(arr, left, mid);
    int rightMajority = majorityElementRec(arr, mid + 1, right);
```

```
    if(leftMajority == rightMajority) return leftMajority;

    int leftCount = countInRange(arr, leftMajority, left, right);
    int rightCount = countInRange(arr, rightMajority, left, right);

    return leftCount > rightCount ? leftMajority : rightMajority;
}

int main() {
    vector<int> arr = {2, 2, 1, 1, 1, 2, 2};
    int majority = majorityElementRec(arr, 0, arr.size() - 1);
    cout << "Majority element: " << majority << endl;
    return 0;
}
```

Test Cases:

- Input: `` → Output: `2` [3] [4]
- Input: `` → Output: `4` [5] [6] [4]

8. Power Function

```
#include<bits/stdc++.h>
using namespace std;

long long power(long long base, long long exp) {
    if(exp == 0) return 1;
    if(exp == 1) return base;

    if(exp % 2 == 0) {
        long long half = power(base, exp / 2);
        return half * half;
    } else {
        return base * power(base, exp - 1);
    }
}

int main() {
    long long base = 2, exp = 10;
    long long result = power(base, exp);
    cout << base << "^" << exp << " = " << result << endl;
    return 0;
}
```

Test Cases:

- Input: `base=2, exp=10` → Output: `1024`
- Input: `base=3, exp=5` → Output: `243`

9. Matrix Multiplication (Strassen's Algorithm)

```
#include<bits/stdc++.h>
using namespace std;

vector<vector<int>> add(vector<vector<int>>& A, vector<vector<int>>& B) {
    int n = A.size();
    vector<vector<int>> C(n, vector<int>(n));
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
    return C;
}

vector<vector<int>> subtract(vector<vector<int>>& A, vector<vector<int>>& B) {
    int n = A.size();
    vector<vector<int>> C(n, vector<int>(n));
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            C[i][j] = A[i][j] - B[i][j];
        }
    }
    return C;
}

vector<vector<int>> strassen(vector<vector<int>>& A, vector<vector<int>>& B) {
    int n = A.size();
    if(n == 1) {
        vector<vector<int>> C(1, vector<int>(1));
        C[^0][^0] = A[^0][^0] * B[^0][^0];
        return C;
    }

    int k = n / 2;
    vector<vector<int>> A11(k, vector<int>(k)), A12(k, vector<int>(k));
    vector<vector<int>> A21(k, vector<int>(k)), A22(k, vector<int>(k));
    vector<vector<int>> B11(k, vector<int>(k)), B12(k, vector<int>(k));
    vector<vector<int>> B21(k, vector<int>(k)), B22(k, vector<int>(k));

    // Divide matrices into quadrants
    for(int i = 0; i < k; i++) {
        for(int j = 0; j < k; j++) {
            A11[i][j] = A[i][j];
            A12[i][j] = A[i][j + k];
            A21[i][j] = A[i + k][j];
            A22[i][j] = A[i + k][j + k];

            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j + k];
            B21[i][j] = B[i + k][j];
            B22[i][j] = B[i + k][j + k];
        }
    }
```

```
    // Calculate M1 to M7
    auto temp1 = add(A11, A22);
    auto temp2 = add(B11, B22);
    auto M1 = strassen(temp1, temp2);

    auto temp3 = add(A21, A22);
    auto M2 = strassen(temp3, B11);

    auto temp4 = subtract(B12, B22);
    auto M3 = strassen(A11, temp4);

    auto temp5 = subtract(B21, B11);
    auto M4 = strassen(A22, temp5);

    auto temp6 = add(A11, A12);
    auto M5 = strassen(temp6, B22);

    auto temp7 = subtract(A21, A11);
    auto temp8 = add(B11, B12);
    auto M6 = strassen(temp7, temp8);

    auto temp9 = subtract(A12, A22);
    auto temp10 = add(B21, B22);
    auto M7 = strassen(temp9, temp10);

    // Calculate result quadrants
    auto temp11 = add(M1, M4);
    auto temp12 = subtract(temp11, M5);
    auto C11 = add(temp12, M7);

    auto C12 = add(M3, M5);
    auto C21 = add(M2, M4);

    auto temp13 = subtract(M1, M2);
    auto temp14 = add(M3, M6);
    auto C22 = add(temp13, temp14);

    // Combine result
    vector<vector<int>> C(n, vector<int>(n));
    for(int i = 0; i < k; i++) {
        for(int j = 0; j < k; j++) {
            C[i][j] = C11[i][j];
            C[i][j + k] = C12[i][j];
            C[i + k][j] = C21[i][j];
            C[i + k][j + k] = C22[i][j];
        }
    }

    return C;
}

int main() {
    vector<vector<int>> A = {{1, 2}, {3, 4}};
    vector<vector<int>> B = {{5, 6}, {7, 8}};

    auto result = strassen(A, B);
```

```
    cout << "Result matrix:" << endl;
    for(auto& row : result) {
        for(int val : row) {
            cout << val << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Test Cases:

- Input: `A=[,], B=[,]` → Output: `[,]` [12] [8] [14] [6] [7] [9] [4] [5] [3]

10. Closest Pair of Points

```
#include<bits/stdc++.h>
using namespace std;

struct Point {
    int x, y;
};

double distance(Point p1, Point p2) {
    return sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
}

double bruteForce(vector<Point>& points, int n) {
    double minDist = DBL_MAX;
    for(int i = 0; i < n; i++) {
        for(int j = i + 1; j < n; j++) {
            minDist = min(minDist, distance(points[i], points[j]));
        }
    }
    return minDist;
}

double stripClosest(vector<Point>& strip, int size, double d) {
    double minDist = d;
    sort(strip.begin(), strip.begin() + size, [](Point a, Point b) {
        return a.y < b.y;
    });

    for(int i = 0; i < size; i++) {
        for(int j = i + 1; j < size && (strip[j].y - strip[i].y) < minDist; j++) {
            minDist = min(minDist, distance(strip[i], strip[j]));
        }
    }
    return minDist;
}

double closestPairRec(vector<Point>& points, int n) {
    if(n <= 3) return bruteForce(points, n);
```

```
    int mid = n / 2;
    Point midPoint = points[mid];

    vector<Point> left(points.begin(), points.begin() + mid);
    vector<Point> right(points.begin() + mid, points.end());

    double dl = closestPairRec(left, mid);
    double dr = closestPairRec(right, n - mid);

    double d = min(dl, dr);

    vector<Point> strip;
    for(int i = 0; i < n; i++) {
        if(abs(points[i].x - midPoint.x) < d) {
            strip.push_back(points[i]);
        }
    }

    return min(d, stripClosest(strip, strip.size(), d));
}

double closestPair(vector<Point>& points) {
    sort(points.begin(), points.end(), [](Point a, Point b) {
        return a.x < b.x;
    });
    return closestPairRec(points, points.size());
}

int main() {
    vector<Point> points = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};
    cout << "Closest distance: " << closestPair(points) << endl;
    return 0;
}
```

Test Cases:

- Input: `[(2,3), (12,30), (40,50), (5,1), (12,10), (3,4)]` → Output: `1.41421`
- Input: `[(0,0), (1,1), (2,2), (3,3)]` → Output: `1.41421`

These problems demonstrate core **divide and conquer principles**: breaking problems into smaller subproblems, solving them recursively, and combining results. Each solution maintains **O(n log n)** or better time complexity, making them efficient for competitive programming scenarios like your CanSat work and algorithm courses.

⁂

1. https://www.thecrazyprogrammer.com/2016/11/caesar-cipher-c-c-encryption-decryption.html
2. https://www.geeksforgeeks.org/dsa/encryption-and-decryption-of-string-according-to-given-technique/
3. https://www.tutorialspoint.com/cplusplus-program-to-implement-caesar-cypher
4. https://www.geeksforgeeks.org/ethical-hacking/caesar-cipher-in-cryptography/
5. https://github.com/CheetahCodes21/SOSC-Competititive-Programming-DSA-mini_project--1

6. https://www.youtube.com/watch?v=0ZxqLyblINQ
7. https://japp.io/cryptography/caesar-cipher-algorithm-program-in-c-c/
8. https://www.scaler.com/topics/caesar-cipher-program-in-c/
9. https://www.trytoprogram.com/c-examples/c-program-to-encrypt-and-decrypt-string/
10. https://stackoverflow.com/questions/58191321/caesar-cipher-decryption-in-c
11. https://stackoverflow.com/questions/25337300/include-bits-stdc-h-with-visual-studio-does-not-compile
12. https://www.scribd.com/document/393115939/Cryptography-Forouzan-Code-Implementation
13. https://www.collegesidekick.com/study-docs/17000819
14. https://www.trytoprogram.com/cpp-examples/cplusplus-program-encrypt-decrypt-string/
15. https://www.youtube.com/watch?v=kY0U1mFmwKE
16. https://www.geeksforgeeks.org/dsa/decrypt-a-string-according-to-given-rules/
17. https://www.daniweb.com/programming/software-development/threads/96440/encrypt-decrypt-a-string
18. https://www.scribd.com/document/901209846/CN-LabFile-Harshit-Gupta
19. https://pdfcoffee.com/download/encryption-and-decryption-pdf-free.html