1. Số lớn thứ nhất và lớn thứ 2 (11.10)

Giải thuật để tìm số lớn nhất và số lớn thứ hai trong mảng A[] như sau:

- Khởi tạo biến max1 và max2 lần lượt là phần tử đầu tiên và phần tử thứ hai trong mảng A[].
- Duyệt qua từng phần tử trong mảng A[] từ phần tử thứ 2 trở đi.
- Nếu phần tử hiện tại lớn hơn max1, gán max2 = max1 và gán max1 bằng phần tử hiện tại.
- Nếu phần tử hiện tại lớn hơn max2 nhưng nhỏ hơn hoặc bằng max1, gán max2 bằng phần tử hiện tại.
- Kết thúc vòng lặp, in ra max1 và max2.

Độ phức tạp thời gian của giải thuật này là O(N), với N là số lượng phần tử trong mảng A[].

```c
#include <stdio.h>

int main(){
    int n;
    scanf("%d",&n);
    int a[n];
    for(int i=0;i<n;i++) scanf("%d",&a[i]);

    int max1=a[0],max2=a[1];
    for(int i = 1; i < n; ++i) {
        if(a[i] > max1){
            max2=max1;
            max1=a[i];
        }else if(a[i] > max2){
            max2=a[i];
        }
    }
    printf("%d %d",max1,max2);
}
```

2. Sàng số nguyên tố (12.3)

Để liệt kê tất cả các số nguyên tố không vượt quá N, chúng ta có thể sử dụng thuật toán sàng Eratosthenes. Thuật toán này được thực hiện như sau:

- Khởi tạo một mảng bool primes[N+1] tất cả các phần tử đều là true. Mảng này sẽ được sử dụng để đánh dấu các số nguyên tố.
- Khởi tạo biến i = 2.
- Lặp lại cho đến khi i * i > N:
    + Nếu primes[i] = true, đánh dấu tất cả các bội số của i (trừ i) trong mảng primes là false.
    + Tăng i lên 1.
- In ra các số nguyên tố không vượt quá N, bằng cách duyệt qua mảng primes và in ra các phần tử có giá trị true tương ứng với các số nguyên tố.

Độ phức tạp thời gian của giải thuật này là O(N log log N).

Lưu ý: trong bài code này: j = i * i vì từ 2i đến (i-1)*I đã được xác định và loại bỏ trước đó.

```c
#include <stdbool.h>
void printPrimeNumbers(int n) {
    //Tạo một mảng đánh dấu, ban đầu tất cả đều là số nguy
    bool isPrime[n+1];
    for (int i = 2; i <= n; i++) {
        isPrime[i] = true;
    }

    // Đánh dấu các bội của các số nguyên tố
    for (int i = 2; i*i <= n; i++) {
        if (isPrime[i]) {
            for (int j = i*i; j <= n; j += i) {
                isPrime[j] = false;
            }
        }
    }
```


```c
    // in ra các số nguyên tố
    for (int i = 2; i <= n; i++) {
        if (isPrime[i]) {
            printf("%d ", i);
        }
    }
}

int main() {
    int n;
    scanf("%d", &n);
    printPrimeNumbers(n);
    return 0;
}
```

3. Tìm kiêm tuyên tính

Tìm kiếm tuyến tính là một thuật toán tìm kiếm một giá trị cụ thể trong một danh sách bằng cách duyệt qua từng phần tử của danh sách đó. Thuật toán này hoạt động theo cách thực hiện các bước sau:

- Duyệt từng phần tử của danh sách cho đến khi tìm thấy giá trị cần tìm hoặc duyệt hết danh sách.
- Nếu giá trị cần tìm bằng với phần tử hiện tại, trả về vị trí của phần tử đó trong danh sách.
- Nếu giá trị cần tìm không bằng với phần tử hiện tại, tiếp tục duyệt qua phần tử tiếp theo của danh sách.

Ví dụ, giả sử chúng ta có danh sách sau: [1, 5, 7, 9, 11, 13] và muốn tìm kiếm giá trị 9. Thuật toán tìm kiếm tuyến tính sẽ duyệt qua từng phần tử của danh sách theo thứ tự từ trái sang phải và trả về vị trí của phần tử có giá trị bằng 9, trong trường hợp này là 3.

Tuy nhiên, nếu giá trị cần tìm không có trong danh sách, thuật toán tìm kiếm tuyến tính sẽ phải duyệt qua toàn bộ danh sách để xác định điều này, do đó độ phức tạp của thuật toán là O(n), với n là số lượng phần tử trong danh sách.

```c
#include <stdio.h>

int linearSearch(int arr[], int n, int x) {
    int i;
    for (i = 0; i < n; i++) {
        if (arr[i] == x) {
            return i; // Trả về vị trí của phần tử cần tìm
        }
    }
    return -1; // Trả về -1 nếu không tìm thấy phần tử cần tìm
}

int main() {
    int arr[] = {2, 4, 6, 8, 10, 12};
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 8;
    int result = linearSearch(arr, n, x);
    if (result == -1) {
        printf("Element not found\n");
    } else {
        printf("Element found at index %d\n", result);
    }
```

4. Tìm kiếm nhị phân (12.6)

Tìm kiếm nhị phân là một thuật toán tìm kiếm hoạt động trên các mảng hoặc danh sách đã được sắp xếp. Thuật toán bắt đầu bằng cách chia mảng hoặc danh sách thành hai nửa, sau đó kiểm tra xem giá trị mục tiêu có nằm trong nửa đầu tiên hay nửa thứ hai không. Nếu không có, thuật toán sẽ loại bỏ một nửa của mảng hoặc danh sách đó và tiếp tục tìm kiếm trên nửa còn lại, cho đến khi tìm thấy giá trị mục tiêu hoặc hiểu rằng giá trị đó không có trong mảng hoặc danh sách đó.

Các bước của thuật toán tìm kiếm nhị phân như sau:

- Sắp xếp mảng hoặc danh sách theo thứ tự tăng dần hoặc giảm dần.
- Thiết lập hai biến "trái" và "phải" để đại diện cho đoạn mảng hoặc danh sách cần tìm kiếm. Ban đầu, giá trị "trái" sẽ là 0 và "phải" là độ dài của mảng hoặc danh sách trừ 1.
- Tính toán phần tử ở giữa đoạn mảng hoặc danh sách bằng cách lấy tổng của "trái" và "phải" rồi chia 2 (có thể sử dụng phép toán dịch bit để tối ưu hơn).
- So sánh giá trị tại vị trí giữa với giá trị cần tìm kiếm. Nếu giá trị này bằng giá trị cần tìm kiếm, trả về vị trí đó. Nếu giá trị này lớn hơn giá trị cần tìm kiếm, thực hiện tìm kiếm trên đoạn bên trái của mảng hoặc danh sách, bằng cách đặt "phải" bằng vị trí giữa - 1. Ngược lại, nếu giá trị này nhỏ hơn giá trị cần tìm kiếm, thực hiện tìm kiếm trên đoạn bên phải của mảng hoặc danh sách, bằng cách đặt "trái" bằng vị trí giữa + 1.
- Lặp lại 2 bước trên cho đến khi tìm thấy giá trị cần tìm kiếm hoặc không còn đoạn nào để tìm kiếm.

Khi tìm thấy giá trị cần tìm kiếm, thuật toán sẽ trả về vị trí của phần tử đó trong mảng hoặc danh sách. Nếu không tìm thấy giá trị cần tìm kiếm, thuật toán sẽ trả về giá trị -1 hoặc báo hiệu không tìm thấy.

```c
int binarySearch(int A[], int left, int right, int X) {
    if (left > right) {
        return 0; // Không tìm thấy X trong mảng A
    }
    int mid = left + (right - left) / 2;
    if (A[mid] == X) {
        return 1; // Tìm thấy X trong mảng A
    } else if (A[mid] > X) {
        return binarySearch(A, left, mid - 1, X); // Tìm kiếm trong mảng con bên trái của vị trí giữa
    } else {
        return binarySearch(A, mid + 1, right, X); // Tìm kiếm trong mảng con bên phải của vị trí giữa
    }
}

int main() {
    int N, T;
    scanf("%d", &N);
    int A[N];
    for (int i = 0; i < N; i++) {
        scanf("%d", &A[i]);
    }
    scanf("%d", &T);
    for (int i = 0; i < T; i++) {
        int X;
        scanf("%d", &X);
```

```c
        int mid = left + (right - left) / 2;
        if (A[mid] == X) {
            return 1; // Tìm thấy X trong mảng A
        } else if (A[mid] > X) {
            right = mid - 1; // Tìm kiếm trong mảng con bên trái của vị trí giữa
        } else {
            left = mid + 1; // Tìm kiếm trong mảng con bên phải của vị trí giữa
        }
    }
    return 0; // Không tìm thấy X trong mảng A
}

int main() {
    int N, T;
    scanf("%d", &N);
    int A[N];
    for (int i = 0; i < N; i++) {
        scanf("%d", &A[i]);
    }
    scanf("%d", &T);
    for (int i = 0; i < T; i++) {
        int X;
        scanf("%d", &X);
        if (binarySearch(A, 0, N-1, X)) {
```

5. Đếm tần suất (12.5)

Để giải bài toán này, ta sử dụng một mảng tần suất để lưu số lần xuất hiện của từng phần tử trong mảng.

- Đầu tiên ta khởi tạo mảng tần suất với giá trị 0 cho mỗi phần tử.
- Sau đó, ta duyệt qua từng phần tử trong mảng ban đầu, tăng giá trị tương ứng trong mảng tần suất lên 1.
- Cuối cùng, ta in ra mảng tần suất theo thứ tự từ nhỏ đến lớn, và sau đó in ra mảng tần suất theo thứ tự xuất hiện trong mảng ban đầu.

```c
#define MAX_VAL 10000000

void countFrequency(int* arr, int n) {
    int freq[MAX_VAL+1] = {0};
    bool printed[MAX_VAL+1] = {false};

    // t?o m?ng t?n su?t và d?m s? l?n xu?t hi?n c?a t?ng ph?n t? trong m?ng
    for (int i = 0; i < n; i++) {
        freq[arr[i]]++;
    }

    // in ra t?n su?t xu?t hi?n c?a các ph?n t? theo th? t? t? nh? d?n l?n
    for (int i = 0; i <= MAX_VAL; i++) {
        if (freq[i] > 0) {
            printf("%d %d\n", i, freq[i]);
        }
    }
    printf("\n");
    // in ra t?n su?t xu?t hi?n c?a các ph?n t? theo th? t? xu?t hi?n trong m?ng
    for (int i = 0; i < n; i++) {
        if (!printed[arr[i]]) {
            printf("%d %d\n", arr[i], freq[arr[i]]);
            printed[arr[i]] = true;
        }
    }
```

```c
    printf("\n");
    // in ra t?n su?t xu?t hi?n c?a các ph?n t? theo th? t? xu?t hi?n
    for (int i = 0; i < n; i++) {
        if (!printed[arr[i]]) {
            printf("%d %d\n", arr[i], freq[arr[i]]);
            printed[arr[i]] = true;
        }
    }
}

int main() {
    int n;
    scanf("%d", &n);
    int arr[n];
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    countFrequency(arr, n);
    return 0;
```