```
void interleaveQueue(queue<int>& q){
    int n = q.size();
    int half = n / 2;

    queue<int> firstHalf;
    queue<int> secondHalf;

    // Split the elements into two halves
    for (int i = 0; i < half; i++) {
        firstHalf.push(q.front());
        q.pop();
    }

    while (!q.empty()) {
        secondHalf.push(q.front());
        q.pop();
    }

    // Interleave the elements
    while (!firstHalf.empty() && !secondHalf.empty()) {
        q.push(firstHalf.front());
        firstHalf.pop();
        q.push(secondHalf.front());
        secondHalf.pop();
    }
}
```

```
bool isBipartite(vector<vector<int>> graph) {
    int n = graph.size();
    int color[1001];
    for (int i = 0; i < n; i++){
        color[i] = -1;
    }
    queue <int> q;
    bool check;
    for (int u = 0; u < n; u++){
        if (color[u] == -1 ){
        check = true;
        q.push(u); // them dinh u vao dau hang doi
        color[u] = 0; // 0 : red , 1 : blue
        while (!q.empty()){
            int v = q.front();  // lay phan tu o dau hang doi
            q.pop();  // xoa phan tu dau hang doi
            for (int x : graph[v]){
            // neu nhu dinh chua duoc to mau thi cho dinh do co mau doi voi dinh u
            if (color[x] == -1){
                color[x] =  1 - color[v];
                q.push(x);
            }
            else if (color[x] == color[v]) return false;
            }
        }

        }
    }
    return true;
}
```

```
void bfs(vector<vector<int>> graph, int start) {
    int n = graph.size();
    bool visited[1001];
    for (int i = 0; i < n; i++){
            visited[i] = false;
    }
    queue <int> q;
    q.push(start);
    visited[start] = true;
    while (!q.empty()){
        int v = q.front(); // lay dinh ow dau hang doi
        q.pop(); //xoa dinh o dau hang doi
        cout << v << " " ;
        for (int x : graph[v]){
            if (!visited[x]){
                q.push(x);
                visited[x] = true;
            }
        }

    }
}
```

```
void push(T item) {
    // TODO: Push new element into the end of the queue
    list.add(item);
}

T pop() {
    // TODO: Remove an element in the head of the queue
    return list.removeAt(0);
}

T top() {
    // TODO: Get value of the element in the head of the queue
    return list.get(0);
}

bool empty() {
    // TODO: Determine if the queue is empty
    return list.empty();
}

int size() {
    // TODO: Get the size of the queue
    return list.size();

}

void clear() {
    // TODO: Clear all elements of the queue
    list.clear();
}
```

```cpp
// iostream, vector and queue are included
// You can write helper methods

long long nthNiceNumber(int n) {
    if (n == 1) {
        return 2LL;
    }

    queue<long long> q;
    q.push(2LL);
    q.push(5LL);

    long long niceNumber = 0;
    while (n > 0) {
        niceNumber = q.front();
        q.pop();

        q.push(niceNumber * 10 + 2);
        q.push(niceNumber * 10 + 5);

        n--;
    }

    return niceNumber;
}
```

==========quá dài nên làm 2 ảnh==========

```
int secondsToBeRotten(vector<vector<int>>& grid) {
    int n = grid.size();
    int m = grid[0].size();
    vector<pair<int, int>> directions = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
    queue<pair<int, int>> rottenApples;
    int freshApples = 0;
    int minutes = 0;

    // Find all the rotten apples and count fresh apples
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (grid[i][j] == 2) {
                rottenApples.push({i, j});
            } else if (grid[i][j] == 1) {
                freshApples++;
            }
        }
    }

    while (!rottenApples.empty()) {
        int size = rottenApples.size();

        while (size--) {
            int x = rottenApples.front().first;
            int y = rottenApples.front().second;
            rottenApples.pop();
```

```
    while (!rottenApples.empty()) {
        int size = rottenApples.size();

        while (size--) {
            int x = rottenApples.front().first;
            int y = rottenApples.front().second;
            rottenApples.pop();

            for (const auto& dir : directions) {
                int newX = x + dir.first;
                int newY = y + dir.second;

                if (newX >= 0 && newX < n && newY >= 0 && newY < m && grid[newX][newY] == 1) {
                    grid[newX][newY] = 2;
                    freshApples--;
                    rottenApples.push({newX, newY});
                }
            }
        }

        if (!rottenApples.empty()) {
            minutes++;
        }
    }

    return freshApples == 0 ? minutes : -1;
}
```

```
int sumOfMaxSubarray(vector<int>& nums, int K) {
    int N = nums.size(), res = 0;
    std::deque<int> Qi(K);
    int i;
    for (i = 0; i < K; ++i) {
        while ((!Qi.empty()) && nums[i] >= nums[Qi.back()])
            Qi.pop_back();
        Qi.push_back(i);
    }
    for (; i < N; ++i) {
        res += nums[Qi.front()];
        while ((!Qi.empty()) && Qi.front() <= i - K)
            Qi.pop_front();
        while ((!Qi.empty()) && nums[i] >= nums[Qi.back()])
            Qi.pop_back();
        Qi.push_back(i);
    }
    res += nums[Qi.front()];
    return res;
}
```