```c
#include <stdio.h>
#include <stdlib.h>

// Un singolo elemento di una lista monodirezionale.
typedef struct _node {
    int key;
    struct _node* next;
} node;

// Struttura di supporto con puntatori a testa e coda di una lista
// e alla dimensione della lista, in modo da ottenere inserimento in coda
// e dimensione in tempo costante e non lineare.
typedef struct {
    node* head;
    node* last;
    size_t size;
} list;

// Struttura di una tabella hash, contiene i parametri della funzione
// di hashing (a, b, p), il numero di liste nella tabella, un puntatore alle
// liste e variabili di supporto aggiuntive per la soluzione di uno specifico
// esercizio, in questo caso la dimensione della lista più grande e il numero
// di conflitti.
typedef struct {
    size_t a, b, p;
    size_t size;
    list* buckets;
    size_t max_bucket_size;
    size_t num_conflicts;
} hashtable;

// Inserimento in coda ad una lista monodirezionale.
void list_append(list *l, int key) {
    node* new = malloc(sizeof(node));
    new->key = key;
    new->next = NULL;

    // Se la lista è vuota bisogna aggiornare il puntatore alla testa
    // altrimenti dobbiamo aggiornare il successore del puntatore alla coda.
    if (l->size == 0) {
        l->head = new;
    } else {
        l->last->next = new;
    }

    l->last = new;
    l->size++;
}

// Calcolo della funzione di hash per una specifica tabella hash.
size_t hash(const hashtable* table, int key) {
    return ((table->a * key + table->b) % table->p) % table->size;
}
```

```c
// Inizializza una tabella hash di dimensione size e parametri a, b per
// la funzione di hashing.
void hashtable_init(hashtable* table, size_t size, size_t a, size_t b) {
    table->a = a;
    table->b = b;
    table->p = 999149;
    table->size = size;
    table->max_bucket_size = 0;
    table->num_conflicts = 0;

    table->buckets = malloc(size * sizeof(list));
    // Inizializza ogni lista come vuota.
    for (size_t i = 0; i < size; i++) {
        table->buckets[i].head = NULL;
        table->buckets[i].last = NULL;
        table->buckets[i].size = 0;
    }
}

// Restituisce il massimo fra due numeri.
#define max(x, y) ((x) > (y) ? (x) : (y))

// Inserimento nella tabella hash.
void hashtable_insert(hashtable* table, int key) {
    size_t h = hash(table, key);

    list_append(&table->buckets[h], key);

    // Aggiorniamo, se necessario, la variabile che contiene la dimensione della
    // lista con più conflitti.
    table->max_bucket_size = max(table->max_bucket_size, table->buckets[h].size);

    // Se abbiamo inserito in una lista non vuota, dobbiamo incrementare
    // il contatore dei conflitti.
    if (table->buckets[h].size != 1) {
        table->num_conflicts++;
    }
}

int main(int argc, char* argv[]) {
    size_t n, a, b;
    int k;

    scanf("%lu %lu %lu", &n, &a, &b);

    hashtable table;
    hashtable_init(&table, 2 * n, a, b);

    for (size_t i = 0; i < n; i++) {
        scanf("%d", &k);
        hashtable_insert(&table, k);
    }

    printf("%lu\n", table.max_bucket_size);
    printf("%lu\n", table.num_conflicts);
```

```
    return 0;
}
```