

Bài 2. Stack và Queue

Stack và queue là hai kiểu dữ liệu trừu tượng rất quan trọng và được sử dụng nhiều trong thiết kế thuật toán. Về bản chất, stack và queue là danh sách tức là một tập hợp các phần tử cùng kiểu có tính thứ tự.

Trong phần này chúng ta sẽ tìm hiểu hoạt động của stack và queue và cách cài đặt chúng bằng các cấu trúc dữ liệu. Tương tự như danh sách, ta gọi kiểu dữ liệu của các phần tử sẽ chứa trong stack và queue là T . Khi cài đặt chương trình cụ thể, kiểu T có thể là kiểu số nguyên, số thực, ký tự, hay bất kỳ kiểu dữ liệu nào được chương trình dịch chấp nhận.

2.1. Stack

Stack (ngăn xếp) là một kiểu danh sách mà việc bổ sung một phần tử và loại bỏ một phần tử được thực hiện ở cuối danh sách.

Có thể hình dung stack như một chồng đĩa, đĩa nào được đặt vào chồng sau cùng sẽ nằm trên tất cả các đĩa khác và sẽ được lấy ra đầu tiên. Vì nguyên tắc “vào sau ra trước”, stack còn có tên gọi là *danh sách kiểu LIFO (Last In First Out)*. Vị trí cuối danh sách được gọi là *đỉnh (top)* của stack.

Đối với stack có các thao tác cơ bản:

- ✿ init: Khởi tạo một stack rỗng
- ✿ empty: Cho biết stack có rỗng không?
- ✿ top: Truy cập phần tử ở đỉnh stack
- ✿ push: Đẩy một phần tử vào stack
- ✿ pop: Lấy ra một phần tử từ stack

2.1.1. Biểu diễn stack bằng mảng

Cách biểu diễn stack bằng mảng cần có một mảng *Items* để lưu các phần tử trong stack và một biến nguyên n để lưu chỉ số của phần tử tại đỉnh stack. Ví dụ:

```
const int maxN = ...;
T Items[maxN];
int n;
```

Các thao tác cơ bản của stack có thể cài đặt như sau:

```
void init() //Khởi tạo stack rỗng
{
    n = -1;
}

bool empty() //Kiểm tra stack có rỗng không
{
    return n < 0;
}
```

```

T& top() //Truy cập phần tử ở đỉnh stack
{
    return Items[n];
}

void push(const T &x) //Đẩy x vào stack
{
    Items[++n] = x;
}

void pop() //Loại bỏ phần tử ở đỉnh stack
{
    --n;
}

```

2.1.2. Biểu diễn stack bằng danh sách móc nối đơn kiểu LIFO

Một cách cài đặt khác là sử dụng danh sách móc nối đơn kiểu LIFO bằng các biến động và con trỏ:

```

struct TNode //Cấu trúc nút
{
    T info; //Thông tin trong nút
    TNode* link; //Liên kết tới nút liền trước
};

TNode* head; //Chốt của danh sách móc nối

void init() //Khởi tạo stack rỗng
{
    head = nullptr; //Chốt == nullptr ⇔ danh sách rỗng
}

bool empty() //Kiểm tra stack có rỗng không
{
    return head == nullptr;
}

T& top() //Truy cập phần tử ở đỉnh stack
{
    return head->info;
}

void push(const T &x) //Đẩy x vào stack
{
    TNode* p = new TNode;
    p->info = x;
    p->link = head;
    head = p;
}

```

```
void pop() //Loại bỏ phần tử ở đỉnh stack
{
    TNode* p = head;
    head = head->link;
    delete p;
}
```

2.1.3. Stack trong C++

C++ cung cấp lớp mẫu `stack<T>` để biểu diễn stack chứa các phần tử kiểu `T`, ngoài các hàm `empty()`, `top()`, `push()`, `pop()` với chức năng trình bày như ở trên. Cần lưu ý rằng đối tượng này khi tạo ra đã là một stack rỗng nhưng trong quá trình sử dụng, không có cách nào để làm rỗng nó ngoại trừ cách lấy ra (`pop`) lần lượt từng phần tử. Ngoài ra cũng không có cách nào truy cập một phần tử khác ngoài phần tử `top()`*

Nếu cần thêm những thao tác khác ngoài những phương thức mà lớp mẫu cung cấp, ta có thể tự cài đặt lại cấu trúc dữ liệu stack như các phần trên hoặc sử dụng những lớp mẫu khác tùy biến hơn. Chẳng hạn sử dụng `vector<T>`, khi đó có thể:

Dùng phương thức `empty()` của vector thay cho phương thức `empty()` của stack

- ✿ Dùng `push_back()` thay cho `push()`
- ✿ Dùng `back()` thay cho `top()`
- ✿ Dùng `pop_back()` thay cho `pop()`
- ✿ Dùng `resize()` để đặt lại số phần tử (`resize(0)` để làm rỗng stack)
- ✿ Dùng iterator hoặc phép toán `[]` để truy cập phần tử

2.2. Queue

Queue (hàng đợi) là một kiểu danh sách mà việc bổ sung một phần tử được thực hiện ở cuối danh sách và việc loại bỏ một phần tử được thực hiện ở đầu danh sách.

Khi cài đặt queue, có hai vị trí quan trọng là vị trí đầu danh sách (*front*), nơi các phần tử được lấy ra, và vị trí cuối danh sách (*back*), nơi phần tử cuối cùng được đưa vào.

Có thể hình dung queue như một đoàn người xếp hàng mua vé: Người nào xếp hàng trước sẽ được mua vé trước. Vì nguyên tắc “vào trước ra trước”, queue còn có tên gọi là *danh sách kiểu FIFO (First In First Out)*.

Các thao tác cơ bản trên queue:

- ✿ `init`: Khởi tạo một queue rỗng
- ✿ `empty`: Cho biết queue có rỗng không?
- ✿ `front`: Đọc giá trị phần tử ở đầu queue
- ✿ `push`: Đẩy một phần tử vào (cuối) queue

* Những phương pháp cố tình giải mã cấu trúc stack trong thư viện chuẩn được coi là hack và không được khuyến khích. Các chương trình không tuân thủ chuẩn ISO Standard không đảm bảo chạy được trong các version khác.

✿ pop: Lấy ra một phần tử từ (đầu) queue

2.2.1. Biểu diễn queue bằng mảng

Ta có thể biểu diễn queue bằng một mảng *Items* để lưu các phần tử trong queue, một biến nguyên *i* để lưu chỉ số phần tử đầu queue và một biến nguyên *j* để lưu chỉ số phần tử cuối queue. Chỉ một phần của mảng *Items* từ vị trí *i* tới *j* được sử dụng lưu trữ các phần tử trong queue. Ví dụ:

```
const int maxN = ...;
T Items[maxN];
int i, j;
```

Các thao tác cơ bản trên queue có thể cài đặt như sau

```
void init() //Khởi tạo queue rỗng
{
    i = 0; j = -1;
}

bool empty() //Kiểm tra queue có rỗng không
{
    return i > j;
}

T& front() //Đọc phần tử đầu queue
{
    return Items[i];
}

void push(const T &x) //Đẩy x vào queue
{
    Items[++j] = x;
}

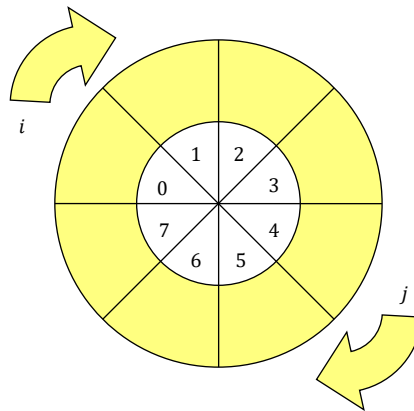
void pop() //Loại bỏ phần tử ở đầu queue
{
    ++i;
}
```

2.2.2. Biểu diễn queue bằng danh sách vòng

Xét việc biểu diễn stack và queue bằng mảng, giả sử mảng có tối đa 10 phần tử (đánh số từ 0 tới 9), ta thấy rằng nếu như bắt đầu từ stack rỗng ($top == -1$), làm 6 lần thao tác *Push*, rồi 4 lần thao tác *Pop*, rồi tiếp tục 8 lần thao tác *Push* nữa thì không có vấn đề gì xảy ra cả. Lý do là vì chỉ số *top* lưu đỉnh của stack sẽ được tăng lên thành 5, rồi giảm về 1, sau đó lại tăng lên thành 9 (chưa vượt quá chỉ số mảng). Nhưng nếu ta thực hiện các thao tác đó đối với cách cài đặt queue như trên thì sẽ gặp lỗi tràn mảng, bởi mỗi lần đẩy phần tử vào queue (*push*), chỉ số cuối queue *j* luôn tăng lên và không bao giờ bị giảm đi cả. Đó chính là nhược điểm mà ta nói tới khi cài đặt: Chỉ có các phần tử từ vị trí *i* tới *j* là thuộc queue, các phần tử ở vị trí khác là vô nghĩa.

Để khắc phục điều này, ta có thể biểu diễn queue bằng một danh sách vòng. Khi dùng mảng *Items*[0 ... *maxN* - 1] thì phần tử *Items*[0] được coi là đứng sau phần tử *Items*[*maxN* - 1].

Nói cách khác, ta coi như các phần tử của queue được xếp quanh vòng tròn theo một chiều nào đó (chẳng hạn chiều kim đồng hồ). Các phần tử nằm trên phần cung tròn từ vị trí i tới vị trí j là các phần tử của queue. Có thêm một biến n lưu số phần tử trong queue. Việc đẩy thêm một phần tử vào queue tương đương với việc ta dịch chỉ số j theo chiều vòng một vị trí rồi đặt giá trị mới vào đó. Việc lấy ra một phần tử trong queue tương đương với việc lấy ra phần tử tại vị trí i rồi dịch chỉ số i theo chiều vòng. (Hình 2-1)



Hình 2-1. Dùng danh sách vòng mô tả queue

Với cách đánh chỉ số từ 0, vị trí kế tiếp vị trí i trên vòng tròn có thể tính bằng công thức $(i + 1) \% \text{maxN}$

```
const int maxN = ...; //Dung lượng cực đại
T Items[maxN];
int i, j, n; //Chỉ số đầu, chỉ số cuối và số phần tử
```

Các thao tác cơ bản trên queue cài đặt trên danh sách vòng có thể cài đặt như sau

```
void init() //Khởi tạo queue rỗng
{
    i = 0; j = maxN - 1;
    n = 0;
}

bool empty() //Kiểm tra queue có rỗng không
{
    return n == 0;
}

T& front() //Đọc phần tử đầu queue
{
    return Items[i];
}

void push(const T &x) //Đẩy x vào queue
{
    Items[j = (j + 1) % maxN] = x;
    ++n;
}
```

```
void pop() //Loại bỏ phần tử ở đầu queue
{
    i = (i + 1) % maxN;
    --n;
}
```

2.2.3. Biểu diễn queue bằng danh sách móc nối đơn kiểu FIFO

Queue cũng có thể cài đặt bằng danh sách móc nối đơn kiểu FIFO với các thao tác được viết như sau:

```
struct TNode //Cấu trúc nút
{
    T info; //Thông tin trong nút
    TNode* link; //Liên kết tới nút liền trước
};
TNode *head, *last; //Chốt của danh sách móc nối và phần tử cuối

void init() //Khởi tạo queue rỗng
{
    head = nullptr; //Chốt == nullptr ⇔ danh sách rỗng
}

bool empty() //Kiểm tra queue có rỗng không
{
    return head == nullptr;
}

T& front() //Truy cập phần tử ở đầu queue
{
    return head->info;
}

void push(const T &x) //Đẩy x vào queue
{
    TNode* p = new TNode;
    p->info = x;
    p->link = nullptr;
    if (head == nullptr) head = p;
    else last->link = p;
    last = p;
}

void pop() //Loại bỏ phần tử ở đầu queue
{
    TNode* p = head;
    head = head->link;
    delete p;
}
```

2.3. DQueue

Tổng quát hơn so với stack và queue là kiểu dữ liệu trừu tượng *Hàng đợi hai đầu (Double-Ended Queue- DQueue)*. Đây là một kiểu danh sách mà việc bổ sung hay loại bỏ một phần tử có thể được thực hiện cả hai đầu danh sách.

DQueue cung cấp các thao tác căn bản:

- ✿ Khởi tạo một DQueue rỗng
- ✿ Cho biến DQueue có rỗng không?
- ✿ Đọc giá trị phần tử ở đầu/cuối DQueue
- ✿ Đẩy một phần tử vào đầu/cuối DQueue
- ✿ Lấy ra một phần tử từ đầu/cuối DQueue

DQueue có thể cài đặt dễ dàng bằng mảng hoặc danh sách móc nối với kỹ thuật sửa đổi một chút từ cách cài đặt queue

2.4. Một số chú ý về kỹ thuật cài đặt

Stack và Queue là hai kiểu dữ liệu trừu tượng tương đối dễ cài đặt, các thủ tục và hàm mô phỏng các thao tác có thể viết rất ngắn. Tuy vậy trong các chương trình dài, các thao tác vẫn nên được tách biệt ra thành hàm để dễ dàng gỡ rối hoặc thay đổi cách cài đặt (ví dụ đổi từ cài đặt bằng mảng sang cài đặt bằng danh sách móc nối). Điều này còn giúp ích cho lập trình viên trong trường hợp muốn biểu diễn các kiểu dữ liệu trừu tượng bằng các lớp và đối tượng. Nếu có băn khoăn rằng việc gọi thực hiện chương trình con sẽ làm chương trình chạy chậm hơn việc viết trực tiếp, bạn có thể đặt các thao tác đó dưới dạng inline functions.

C++ và hệ thống thư viện chuẩn đi kèm cũng có sẵn các lớp biểu diễn các cấu trúc dữ liệu như stack, queue và DQueue.

2.4.1. Stack trong C++ STL

Lớp mẫu `stack<T>` là stack chứa các phần tử kiểu T, nó cung cấp các hàm cơ bản:

- ✿ `bool empty()`: Kiểm tra stack có rỗng không
- ✿ `size_type size()`: Trả về số phần tử đang có trong stack
- ✿ `T& top()`: Trả về (tham chiếu tới) phần tử ở đỉnh stack
- ✿ `void push(x)`: Đẩy x (kiểu T) vào đỉnh stack
- ✿ `void pop()`: Lấy ra phần tử ở đỉnh stack
- ✿ `void emplace(...)`: (C++ 11): Nếu T là class hay struct, hàm đẩy đối tượng mới vào stack và khởi tạo đối tượng mới theo các tham số truyền vào.

2.4.2. Queue trong C++ STL

Lớp mẫu `queue<T>` là queue chứa các phần tử kiểu T, nó cung cấp các hàm cơ bản:

- ✿ `bool empty()`: Kiểm tra queue có rỗng không
- ✿ `size_type size()`: Trả về số phần tử đang có trong queue
- ✿ `T& front()`: Trả về (tham chiếu tới) phần tử đầu queue
- ✿ `T& back()`: Trả về (tham chiếu tới) phần tử cuối queue
- ✿ `void push(x)`: Đẩy x (kiểu T) vào cuối queue
- ✿ `void pop()`: Lấy ra phần tử ở đầu queue
- ✿ `void emplace(...)`: (C++ 11): Nếu T là class hay struct, hàm đẩy đối tượng mới vào cuối queue và khởi tạo đối tượng mới theo các tham số truyền vào.

2.4.3. Dequeue trong C++ STL

Lớp mẫu `deque<T>` là `Dequeue` chứa các phần tử kiểu `T`. Lớp mẫu này cung cấp các hàm cơ bản:

- ✿ **bool empty()**: Kiểm tra `Dequeue` có rỗng không
- ✿ **size_type size()**: Trả về số phần tử đang có trong `Dequeue`
- ✿ **void resize(s)**: Thay đổi số phần tử trong `Dequeue` thành `s`. Nếu `Dequeue` đang chứa nhiều hơn `s` phần tử, những phần tử cuối `Dequeue` sẽ bị cắt bỏ. Nếu `Dequeue` đang chứa ít hơn `s` phần tử, những phần tử mới sẽ thêm vào cho đủ số phần tử `s`, nếu dùng `resize(s, v)`, những phần tử mới sẽ nhận giá trị bằng `v`
- ✿ **T& front()**: Trả về (tham chiếu tới) phần tử đầu `Dequeue`
- ✿ **T& back()**: Trả về (tham chiếu tới) phần tử cuối `Dequeue`
- ✿ **void push_front(x)**: Đẩy `x` (kiểu `T`) vào đầu `Dequeue`
- ✿ **void push_back(x)**: Đẩy `x` (kiểu `T`) vào cuối `Dequeue`
- ✿ **void pop_front()**: Lấy ra phần tử ở đầu `Dequeue`
- ✿ **void pop_back()**: Lấy ra phần tử ở cuối `Dequeue`
- ✿ **void emplace_front(...)**: (C++ 11): Nếu `T` là class hay struct, hàm đẩy đối tượng mới vào đầu `Dequeue` và khởi tạo đối tượng mới theo các tham số truyền vào.
- ✿ **void emplace_back(...)**: (C++ 11): Nếu `T` là class hay struct, hàm đẩy đối tượng mới vào cuối `Dequeue` và khởi tạo đối tượng mới theo các tham số truyền vào.
- ✿ **at(i)** và toán tử `[i]`: Trả về (tham chiếu tới) phần tử đứng thứ `i` trong queue

Lưu ý rằng `deque<T>` còn cung cấp các iterator để duyệt phần tử thuận tiện hơn. Iterator của `deque` là loại cho phép truy cập ngẫu nhiên (*random access*)

- ✿ **iterator begin()**: Iterator tới phần tử đầu `Dequeue`
- ✿ **iterator end()**: Iterator tới sau phần tử cuối `Dequeue`
- ✿ **reverse_iterator rbegin()**: Reverse Iterator (Iterator chạy ngược) tới phần tử cuối `Dequeue`
- ✿ **reverse_iterator rend()**: Reverse Iterator tới phần tử trước phần tử đầu `Dequeue`

2.5. Một số bài toán ứng dụng

2.5.1. Cặp dấu ngoặc

🌀 Bài toán

Một dãy ngoặc đúng là một xâu ký tự định nghĩa như sau:

- ✿ Xâu rỗng (không có ký tự nào là một dãy ngoặc đúng)
- ✿ Nếu `S` là một dãy ngoặc đúng thì `(S)` là một dãy ngoặc đúng, dấu mở ngoặc thêm vào đầu xâu `S` và dấu đóng ngoặc thêm vào cuối xâu `S` được gọi là cặp với nhau
- ✿ Nếu `S` và `T` là hai dãy ngoặc đúng thì `S + T` (nối xâu `T` vào sau xâu `S`) là một dãy ngoặc đúng

Cho chuỗi ký tự $S = s_0 s_1 \dots s_{n-1}$ là một dãy ngoặc đúng. Với mỗi dấu ')', cho biết vị trí dấu '(' cặp với nó

Input

Một dòng chứa chuỗi S độ dài $n \leq 10^6$ tương ứng với một dãy ngoặc đúng, các ký tự được đánh số từ 0 tới n

Output

Ghi ra $\frac{n}{2}$ số, với mỗi dấu ')' tính từ đầu dãy, in ra vị trí dấu '(' cặp với nó

Sample Input	Sample Output
((()))((()))	1 0 6 5 4

Thuật toán

Trước hết ta xét một thuật toán đơn giản: Tìm dấu ngoặc đóng ')' đầu tiên, đứng trước nó chắc chắn là dấu ngoặc mở '(' ứng với nó. Ghi nhận lại vị trí, xóa luôn cặp dấu ngoặc này và lặp lại cho tới khi chuỗi trở thành rỗng.

Thuật toán trên có nhược điểm:

- ✿ Khó quản lý vị trí do mỗi phép xóa sẽ làm các ký tự bị đánh chỉ số lại (tuy điều này có thể khắc phục bằng cách duyệt ngược chuỗi, với mỗi dấu ngoặc mở ghi nhận vị trí dấu ngoặc đóng cặp với nó... nhưng sẽ phải có thêm thao tác sắp xếp lại kết quả trước khi xuất ra output)
- ✿ Tốc độ chậm do phép xóa chuỗi (Thuật toán có thời gian thực hiện $O(n^2)$)

Ta có thể sử dụng một thuật toán khác hiệu quả hơn sử dụng stack chứa các số nguyên: Duyệt chuỗi từ đầu đến cuối:

- ✿ Gặp dấu '(': Đẩy vị trí của nó vào stack
- ✿ Gặp dấu ')': In ra vị trí lưu ở đỉnh stack là vị trí dấu '(' tương ứng và loại bỏ phần tử ở đỉnh stack

Không khó khăn để hình dung ra cơ chế thực hiện và dễ dàng chứng minh được thuật toán có độ phức tạp $O(n)$

Cài đặt

🔗 PARENTHESES.CPP ✓ Cặp dấu ngoặc 🔗

```
#include <iostream>
#include <string>
#include <stack>
using namespace std;

int main()
{
    stack<int> s;
    string a;
    getline(cin, a);
    for (int i = 0; i < a.length(); ++i)
        if (a[i] == '(') s.push(i);
```

```

else
{
    cout << s.top() << ' ';
    s.pop();
}
}

```

2.5.2. Kỹ thuật xếp hàng

Một số bài toán xử lý dãy có thể giải quyết bằng cách duyệt lần lượt từng phần tử, khi duyệt tới một phần tử, những phần tử trước đó không còn giá trị sử dụng sẽ bị loại bỏ. Nếu quan sát thấy các phần tử cần loại bỏ là một dãy liên tiếp đứng liền trước phần tử mới, đó là cơ sở để chúng ta sử dụng stack.

Như bài toán cặp dấu ngoặc, khi gặp một dấu ngoặc đóng ')', dấu ngoặc mở '(' cuối cùng chính là dấu ngoặc cặp với nó và sau khi ghi nhận cặp dấu ngoặc này, dấu '(' cuối cùng không còn giá trị sử dụng nó, ta có thể loại bỏ.

Ta xét một bài toán khác:

🌸 Hình chữ nhật lớn nhất

Trên mặt phẳng người ta vẽ n cột hình chữ nhật dựng sát nhau, đáy nằm trên một đường thẳng nằm ngang. Mỗi cột có độ rộng 1 đơn vị và chiều cao biết trước. Nếu đánh số các cột từ 0 tới $n - 1$ từ trái qua phải thì cột thứ i có chiều cao h_i

Yêu cầu: Tìm hình chữ nhật có diện tích lớn nhất nằm trong phần các cột đã vẽ

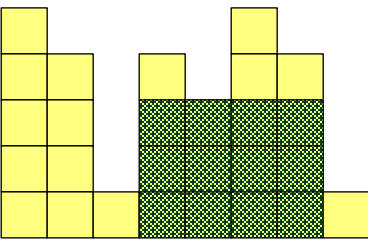
Input

- 🌟 Dòng 1 chứa số nguyên dương $n \leq 10^6$
- 🌟 Dòng 2 chứa n số nguyên dương h_0, h_1, \dots, h_{n-1} ($\forall i: h_i \leq 10^9$) cách nhau bởi dấu cách

Output

Diện tích hình chữ nhật theo phương án tìm được

Sample Input	Sample Output
8 5 4 1 4 3 5 4 1	12



🌸 Thuật toán

Giải pháp tầm thường là ta có thể xét mọi phạm vi từ cột i tới cột j ($\forall i \leq j$) và tìm hình chữ nhật lớn nhất có chiều rộng phủ hết phạm vi đó. Dễ thấy rằng hình chữ nhật lớn nhất này có chiều rộng đúng bằng $j - i + 1$ và chiều cao bằng chiều cao cột thấp nhất trong phạm vi từ cột i tới cột j .

Không khó để chỉ ra rằng thuật toán trên có thời gian thực hiện $\Theta(n^3)$. Ta có thể tối ưu hơn bằng cách với mỗi chỉ số i , ta xét các chỉ số j từ i tới $n - 1$, j chạy tới đâu cập nhật lại giá trị chiều cao cột thấp nhất trong phạm vi. Cải tiến này cho phép cài đặt thuật toán với độ phức tạp $\Theta(n^2)$, tuy nhiên vẫn chưa đủ tốt.

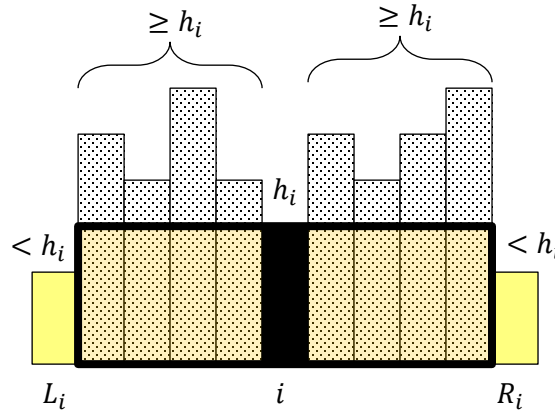
Nhận xét rằng hình chữ nhật có diện tích lớn nhất chắc chắn sẽ chứa trọn vẹn một cột nào đó (nếu không ta có thể giữ nguyên chiều rộng và nâng chiều cao lên thêm). Nếu hình chữ nhật cần tìm chứa trọn vẹn cột i thì chiều cao của nó chắc chắn bằng h_i . Chiều rộng của nó phủ qua cột i và vấn đề còn lại là của chúng ta chỉ là cố gắng “nới rộng” hình chữ nhật này về cả bên trái và bên phải cột i .

Như vậy nếu ta biết được hai giá trị sau với mỗi cột i :

- ✿ L_i : Cột đứng trước cột i , gần cột i nhất mà chiều cao thấp hơn cột i
- ✿ R_i : Cột đứng sau cột i , gần cột i nhất mà chiều cao thấp hơn cột i

Hình chữ nhật lớn nhất chứa trọn cột i có chiều cao h_i còn chiều rộng của nó sẽ trải từ cột $L_i + 1$ tới cột $R_i - 1$ (Xem Hình 2-2). Diện tích hình chữ nhật này là $h_i \times (R_i - L_i - 1)$. Từ đó suy ra hình chữ nhật lớn nhất dựng trong phạm vi các cột đã vẽ là:

$$\min_{0 \leq i < n} \{h_i \times (R_i - L_i - 1)\}$$



Hình 2-2. Ý nghĩa của các chốt $L[.]$, $R[.]$

Bây giờ ta sẽ trình bày thuật toán tính các chốt $L[0 \dots n - 1]$. Việc tính toán các chốt $R[0 \dots n - 1]$ sẽ được thực hiện tương tự.

Mô hình xếp hàng: Tưởng tượng như ta cho các cột lần lượt xếp hàng với một hàng cột ban đầu được khởi tạo rỗng. Xếp lần lượt các cột vào hàng theo thứ tự $0, 1, 2, \dots, n - 1$. Trước khi xếp một cột i vào, ta hủy ngay cột đứng cuối hàng nếu thấy cột đứng cuối hàng có chiều cao lớn hơn hay bằng cột i và cứ lặp lại như vậy... Khi quá trình kết thúc, nếu trong hàng không còn cột nào, tức là phía trước cột i không có cột nào thấp hơn nó, ta ghi nhận $L_i = -1$, nếu

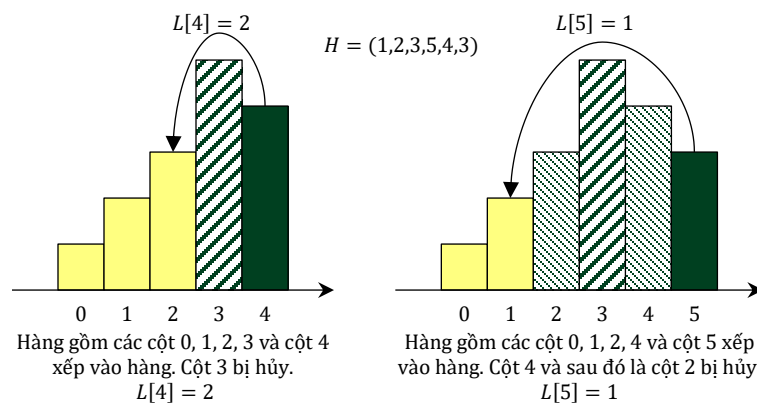
không thì cột không bị hủy đứng cuối hàng chính là cột đứng trước cột i , gần cột i nhất và có chiều cao thấp hơn h_i . Việc cuối cùng là cho cột i xếp vào cuối hàng.

Hình 2-3 là ví dụ về cơ chế thực hiện khi xếp 6 cột vào hàng với độ cao lần lượt là (1,2,3,5,4,3). 4 cột đầu tiên vào hàng và không có cột nào bị hủy do cột đứng cuối hàng tại mỗi bước đều thấp hơn cột đang xét. Ta có $L_0 = -1$; $L_1 = 0$; $L_2 = 1$ và $L_3 = 2$.

Khi cột số 4 (độ cao 4) xếp vào hàng cột đứng cuối hàng (cột 3) bị hủy do có chiều cao $5 > 4$. Ta ghi nhận $L_4 = 2$ (hình bên trái) và hàng gồm 4 cột (0,1,2,4)

Khi cột số 5 (độ cao 3) xếp vào hàng, lần lượt cột 4 bị hủy do chiều cao $4 > 3$ rồi cột 2 bị hủy do chiều cao $3 \geq 3$ (hình bên phải). Ta ghi nhận $L_5 = 1$ và khi cột 5 xếp vào, hàng gồm có 3 cột (0,1,5)

11/2021



Hình 2-3. Mô hình xếp hàng.

11/2021

Tính đúng đắn của giải thuật có thể diễn giải qua lý do ta loại bỏ các cột: Khi cột thứ i chuẩn bị xếp vào hàng, những cột cao hơn hay bằng cột i đang trong hàng sẽ bị hủy vì nó không còn vai trò trong việc tính $L[i]$ do chúng có chiều cao $\geq h_i$. Chúng cũng không còn vai trò trong việc tính $L[i + 1], L[i + 2], \dots, L[n - 1]$ do sẽ có cột i đứng phía sau rồi nên những cột kế tiếp khi tính $L[\cdot]$ nếu cần sẽ chọn cột i là cột gần chúng hơn và có độ cao thấp hơn hoặc bằng những cột bị hủy.

Để ý rằng các cột trong hàng luôn được xếp theo thứ tự tăng ngặt của chiều cao. Mỗi cột khi xét đến sẽ được đứng vào cuối hàng với tư cách là cột cao nhất trong hàng (những cột cao hơn hay bằng nó đứng phía trước đã bị hủy)

Giải thuật có thể cài đặt dễ dàng bằng một stack chứa các chỉ số cột để mô phỏng quá trình thực hiện với đỉnh stack là phần tử đứng cuối hàng: Bắt đầu với một stack rỗng, ta xét lần lượt các cột. Khi xét tới một cột i ta lần lượt loại bỏ chỉ số chứa ở đỉnh stack nếu nó ứng với cột có độ cao $\geq h_i$, cuối cùng là ghi nhận lại $L[i]$ là chỉ số ở đỉnh stack (hoặc -1 nếu stack đã rỗng) và đẩy cột i vào stack.

```

Stack = ∅;
for (i = 0, 1, ..., n - 1)
{
    while (Stack && chỉ số cột ở đỉnh Stack ứng với cột >= h[i])
        «Lấy ra phần tử ở đỉnh Stack»;
    if (Stack == ∅)
        L[i] = -1;
    else
        L[i] = «chỉ số cột ở đỉnh Stack»;
    «Đẩy chỉ số i vào đỉnh Stack»;
}

```

Thời gian thực hiện giải thuật có thể đánh giá qua số lệnh lấy ra phần tử ở đỉnh Stack (lệnh nằm trong lõi cả vòng lặp for và vòng lặp while). Rõ ràng mỗi chỉ số chỉ bị đẩy vào Stack 1 lần vì vậy cũng chỉ có tối đa n lần ta loại bỏ phần tử ở đỉnh Stack. Khi đánh giá bù trừ, thời gian thực hiện giải thuật là $O(n)$ do các thao tác cơ bản trên Stack có thể thực hiện trong thời gian $O(1)$.

☞ MAXRECT.CPP ✓ Hình chữ nhật lớn nhất ☞

```

#include <iostream>
#include <stack>
using namespace std;
const int maxN = 1e6;
int n, h[maxN], L[maxN], R[maxN];

void Enter() //Nhập dữ liệu
{
    cin >> n;
    for (int i = 0; i < n; i++) cin >> h[i];
}

void CallR()
{
    stack<int> Stack; //Stack chứa các chỉ số cột
    //Tính các chốt L[0...n - 1]
    for (int i = 0; i < n; ++i) //Xét các cột từ đầu đến cuối
    {
        //Chỉ số cột ở đỉnh Stack ứng với cột >= h[i] thì loại bỏ đỉnh Stack
        while (!Stack.empty() && h[Stack.top()] >= h[i]) Stack.pop();
        if (Stack.empty()) L[i] = -1; //Stack = ∅ thì chốt L[i] coi như -1
        else L[i] = Stack.top(); //Stack ≠ ∅ thì chốt L[i] đặt bằng chỉ số ở đỉnh Stack
        Stack.push(i); //Xếp chỉ số i vào đỉnh Stack
    }
    while (!Stack.empty()) Stack.pop(); //Làm rỗng Stack, chuẩn bị tính R

    //Tính các chốt R[n - 1...0]
    for (int i = n - 1; i >= 0; --i) //Xét các cột từ cuối lên đầu
    {
        //Chỉ số cột ở đỉnh Stack ứng với cột >= h[i] thì loại bỏ đỉnh Stack
        while (!Stack.empty() && h[Stack.top()] >= h[i]) Stack.pop();
        if (Stack.empty()) R[i] = n; //Stack = ∅ thì chốt R[i] coi như n
        else R[i] = Stack.top(); //Stack ≠ ∅ thì chốt R[i] đặt bằng chỉ số ở đỉnh Stack
        Stack.push(i); //Xếp chỉ số i vào đỉnh Stack
    }
}

void GetMaxRect() //Tính maxArea là giá trị lớn nhất trong các giá trị h[i] * (R[i] - L[i] - 1)

```

```

{
    long long maxArea = 0;
    for (int i = 0; i < n; ++i)
    {
        long long t = (long long)h[i] * (R[i] - L[i] - 1);
        if (maxArea < t) maxArea = t;
    }
    cout << maxArea;
}

int main()
{
    Enter();
    CallLR();
    GetMaxRect();
}



```

2.5.3. Giá trị nhỏ nhất trên các khoảng tịnh tiến

Bài toán

Có n người đánh số từ 0 tới $n - 1$ xếp thành một hàng theo đúng thứ tự, người thứ i có chiều cao h_i . Cho k là một số nguyên dương ($k \leq n$). Với mỗi người i trong dãy ($k - 1 \leq i \leq n - 1$), xác định s_i là chiều cao người thấp nhất trong số k người liên tiếp tính từ người i trở về trước. ($s_i = \min_{i-k+1 \leq j \leq i} \{h_j\}; \forall i: k - 1 \leq i \leq n - 1$).

Input

-  Dòng 1 chứa hai số nguyên dương $n \leq 10^6, k \leq n$ cách nhau bởi dấu cách
-  Dòng 2 chứa n số nguyên dương h_0, h_1, \dots, h_{n-1} ($\forall i: h_i \leq 10^9$) cách nhau bởi dấu cách

Output

Ghi ra $n - k + 1$ số $s_{k-1}, s_k, \dots, s_{n-1}$ cách nhau bởi dấu cách.

Ví dụ:

Sample Input	Sample Output
5 3	1 1 3
2 1 5 3 4	

Thuật toán

Giải pháp tầm thường là ta duyệt tất cả các khoảng gồm k phần tử liên tiếp trong mảng h và tiến hành tìm giá trị nhỏ nhất. Thuật toán có độ phức tạp $O((n - k) \times k)$. Mặc dù có những cấu trúc dữ liệu hỗ trợ tìm phần tử nhỏ nhất nhanh hơn như Heap, Segment Trees, BST, ... nhưng chúng khá phức tạp. Ta sẽ trình bày một kỹ thuật đơn giản và hiệu quả.

Bắt đầu từ mô hình xếp hàng: Lần lượt từng người đứng vào hàng từ người 0 tới người $n - 1$: Khi một người i chuẩn bị xếp vào hàng:

- ✿ Người i chuẩn bị xếp vào hàng trong tình trạng những giá trị $s_{k-1}, s_k, \dots, s_{i-1}$ đã được xác định, tức là giá trị nhỏ nhất của các đoạn k phần tử liên tiếp đứng trước vị trí i trong mảng h đã được tính
- ✿ Trước khi đứng vào, người i đuổi người đứng cuối hàng nếu người này có chiều cao $\geq h_i$ và cứ lặp lại quá trình như vậy cho tới khi hàng bị rỗng hoặc người đứng cuối hàng phải có chiều cao $< h_i$
- ✿ Người i xếp vào cuối hàng và tiến hành tính s_i

Lý do đuổi ở trên rất tự nhiên, một người đứng trước cao hơn hoặc bằng người i sẽ vô dụng trong việc tính s_i (ta chỉ cần giá trị nhỏ nhất) và vì thế cũng sẽ vô dụng trong việc tính s_{i+1}, s_{i+2}, \dots sau này. Trong khi đó thì các giá trị s_{i-1}, s_{i-2}, \dots trở về trước theo giả thiết đã được xác định.

Một đặc điểm của hàng người theo thuật toán này mà ta đã chỉ ra ở ví dụ trước: hàng luôn được xếp theo thứ tự tăng ngặt của chiều cao bởi mỗi người i khi xét đến sẽ được đứng vào cuối hàng với tư cách là người cao nhất trong hàng (những người cao hơn hay bằng anh ta đứng phía trước đã bị đuổi). Đây là nhận xét để ta triển khai thao tác tính s_i .

Vì hàng được xếp theo thứ tự tăng ngặt của chiều cao, người đứng đầu hàng luôn là người thấp nhất của hàng. Nếu người đứng đầu hàng nằm ngoài phạm vi tính min, tức là phạm vi k người tính từ vị trí i trở về trước, anh ta trở nên vô dụng khi tính s_i và dĩ nhiên cũng vô dụng trong việc tính s_{i+1}, s_{i+2}, \dots sau này (do cũng bị ngoài phạm vi), anh ta sẽ bị đuổi và lặp lại với người đứng đầu hàng mới. Khi quá trình này kết thúc, chiều cao của người đứng đầu hàng chính là đáp số s_i cần xác định.

Nhận xét cuối cùng là tại mỗi bước tính s_i ta chỉ cần đuổi tối đa 1 người đứng đầu hàng vì nếu người đứng đầu x và người kế tiếp y đều bị đuổi do ngoài phạm vi tính min thì người x đã bị người $i - 1$ đuổi ở bước trước do ngoài phạm vi tính min của anh ta rồi.

Thuật toán đã rõ ràng, bây giờ ta có thao tác đuổi ở cả hai đầu của hàng người, cấu trúc dữ liệu có thể áp dụng tự nhiên là hàng đợi hai đầu (Deque).

🌀 MINIMUM.CPP ✓ Giá trị nhỏ nhất trên các khoảng tịnh tiến 🌀

```
#include <iostream>
#include <deque>
using namespace std;
const int maxN = 1e6;
int n, k, h[maxN];
deque<int> q; //Deque q chứa các chỉ số

void Solve()
{
    for (int i = 0; i < n; i++) //Xét lần lượt từ người 0 tới người n - 1
    {
        cin >> h[i]; //Nhập chiều cao người i
        //Đuổi tất cả những người đứng cuối hàng có chiều cao  $\geq h[i]$ 
        while (!q.empty() && h[q.back()] >= h[i])
            q.pop_back();
```

```

        q.push_back(i); //Cho người i xếp vào cuối hàng
        if (q.front() + k <= i) //Người đầu hàng nằm ngoài phạm vi tính min
            q.pop_front(); //Đuổi người đầu hàng
        if (i >= k - 1) //Tính từ i trở về trước có đủ k người
            cout << h[q.front()] << ' '; //In ra chiều cao người đầu hàng là người thấp nhất
    }
}

int main()
{
    cin >> n >> k;
    Solve();
}

```

Thời gian thực hiện giải thuật có thể đánh giá qua số lần ta thực hiện các thao tác trên DEqueue: $q.pop_back()$, $q.pop_front()$ và $q.push_back()$. Rõ ràng mỗi người được vào hàng 1 lần (có n lệnh $q.push_back()$) nên sẽ có tối đa n lệnh đuổi người $q.pop_back()$ và $q.pop_front()$. Thời gian thực hiện giải thuật là $O(n)$ do các thao tác cơ bản trên DEqueue có thể cài đặt để chạy trong thời gian $O(1)$.

2.5.4. Kỹ thuật loang



Một số bài toán thực tế có thể mô hình hóa dưới dạng máy trạng thái, có thể hiểu là một “máy” xác định bởi tập các trạng thái và một tập các luật chuyển, mỗi luật chuyển cho phép máy đang ở một trạng thái chuyển sang một trạng thái khác. Yêu cầu đặt ra là phải chuyển máy từ một trạng thái khởi đầu về một trạng thái kết thúc bằng cách sử dụng một dãy hợp lý các luật chuyển.

Kỹ thuật loang là một trong những giải pháp cho vấn đề này. Nội dung của phương pháp khá trực quan: Coi mỗi trạng thái là một bể chứa và mỗi luật chuyển như đường ống thông giữa hai bể, ta đổ nước vào bể ứng với trạng thái khởi đầu và theo dõi quá trình nước “loang” đến bể ứng với trạng thái kết thúc qua các đường ống được thiết lập.

Bài toán

Ta xét một bài toán cụ thể: Cho dãy số nguyên $A = (a_0, a_1, \dots, a_{n-1})$ và một số nguyên t . Với số nguyên s , bạn được phép thực hiện các phép biến đổi: chọn một phần tử $a_i \in A$, sau đó cộng thêm s lên a_i . Hãy một số ít nhất các phép biến đổi để biến s với giá trị ban đầu là 0 thành t . Mỗi phần tử a_i có thể được sử dụng nhiều lần trong các phép biến đổi

Input

-  Dòng 1 chứa hai số nguyên n, t ($1 \leq n \leq 100; |t| \leq 10^5$)
-  Dòng 2 chứa n số nguyên a_0, a_1, \dots, a_{n-1} ($\forall i: |a_i| \leq 10^5$)

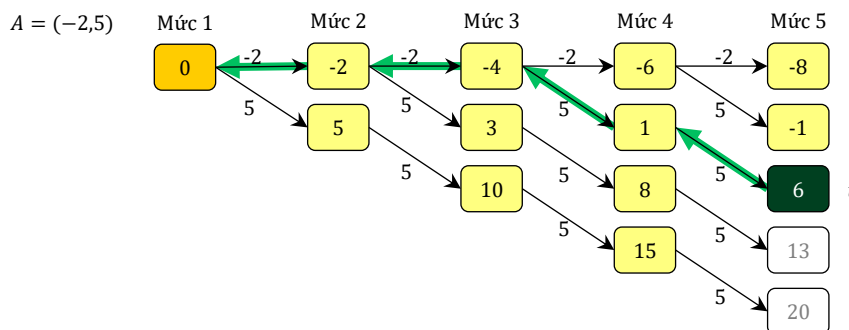
Output

Danh sách các giá trị a_i được chọn trong cách biến đổi 0 thành t , hoặc thông báo rằng không thể thực hiện được.

Cài đặt giải thuật loang theo lớp

Để in ra cụ thể các phép biến đổi, một kỹ thuật được sử dụng gọi là lưu vết: Khi một giá trị v được loang đến nhờ phép cộng một giá trị u ở mức trước với một giá trị $x \in A$. Ta lưu lại vết $trace[v] = x$ với ý nghĩa rằng phần tử v được sinh ra nhờ phép biến đổi: cộng thêm x . Khi đó dãy phép biến đổi từ 0 thành t có thể truy lại một cách đơn giản: Bắt đầu với $v = t$, ta biết rằng dãy phép biến đổi biến 0 thành v có chứa phép cộng với $trace[v]$ (chú ý $trace[v]$ là một giá trị trong A), ta in ra phép biến đổi này và lặp lại với giá trị $v_{\text{mới}} = v_{\text{cũ}} - trace[v]$ cho tới khi $v = 0$:

Ví dụ với $A = (-2,5)$ và $t = 6$, quá trình truy vết cho biết để biến $s = 0$ thành 6 ta cần thực hiện các phép cộng với 5, 5, -2 và -2 (Hình 2-5)



Hình 2-5. Sơ đồ truy vết

Cơ chế đánh dấu một giá trị thuộc phạm vi hiệu lực $[-10^5 \dots 10^5]$ là đã loang đến hay chưa được thực hiện bằng phép đánh dấu đơn giản, ở đây ta tận dụng luôn mảng *trace*: Ta biết rằng khi một giá trị *v* được loang đến thì *trace[v]* là một giá trị trong *A*, vì vậy để đánh dấu các giá trị chưa được loang đến, ta sẽ gán *trace[.]* bằng một giá trị chắc chắn không có mặt trong *A*, ký hiệu *NA* (chẳng hạn $NA = 10^5 + 1$). Riêng với giá trị 0 là giá trị khởi đầu, ta gán *trace[0] = 0* hoặc bất kỳ hằng số nào khác *NA* là được.

Cuối cùng, do C++ không chấp nhận chỉ số âm đối với mảng, có thể ánh xạ miền chỉ số $[-10^5 \dots 10^5]$ lên thành miền chỉ số không âm nhờ phép cộng thêm hằng số 10^5 . Một cách khác tiện hơn là dùng con trỏ để ánh xạ chỉ số: Khai báo mảng $_trace[0 \dots 2 \cdot 10^5]$ với miền chỉ số dài đúng bằng phạm vi hiệu lực. Sau đó đặt con trỏ $trace$ vào địa chỉ phần tử $_trace[10^5]$. Theo C++ Standard, vì $trace == _trace + 10^5$ và $trace[v]$ tương đương với $*(trace + v)$ nên phép ánh xạ chỉ số được thực hiện tự nhiên:

$$trace[v] \leftrightarrow *(trace + v) \leftrightarrow *(_trace + 10^5 + v) \leftrightarrow _trace[10^5 + v]$$

Tức là ta dùng chính mảng $_trace[0 \dots 2 \cdot 10^5]$ để chứa dữ liệu mảng $trace[-10^5 \dots 10^5]$

Cụ thể một vài trường hợp:

```
trace ↔ &\_trace[105] ↔ \_trace + 105
trace[0] ↔ *(trace + 0) ↔ *(\_trace + 105) ↔ \_trace[105]
trace[-1] ↔ *(trace - 1) ↔ *(\_trace + 105 - 1) ↔ \_trace[105 - 1]
trace[-105] ↔ *(trace - 105) ↔ *(\_trace + 105 - 105) ↔ \_trace[0]
trace[105] ↔ *(trace + 105) ↔ *(\_trace + 105 + 105) ↔ \_trace[2 · 105]
```

NUMTRANS1.CPP ✓ Biến đổi số

```
#include <iostream>
#include <vector>
using namespace std;
const int NA = 1e5 + 1;
const int lim = 1e5;
int n, t;
vector<int> a;
int \_trace[4 * lim + 1], *trace;

void Enter() //nhập dữ liệu, mảng a được biểu diễn bằng vector cho tiện
{
    cin >> n >> t;
    a.resize(n);
    for (int& x: a) cin >> x; //Nhập mảng a
    trace = \_trace + lim; //Đặt con trỏ trace dịch lên 105 phần tử so với \_trace
    fill(begin(\_trace), end(\_trace), NA); //Tất cả các giá trị đều chưa loang đến, trace[.] = NA
}

void Solve() //Thuật toán loang
{
    vector<int> p, q; //p: Tập giá trị ở mức loang trước, q: Tập giá trị sẽ loang từ p
    trace[0] = 0; //Giá trị 0 coi như đã loang đến
    if (t == 0) return; //Nếu t == 0 không phải làm gì cả
    p.push_back(0); //Mức 0 chỉ gồm mỗi giá trị 0: {0}
    while (!p.empty()) //Loang theo lớp
    {
        //Loang từ lớp p sang lớp q
        for (int u: p) //Xét mọi giá trị u ở lớp p
            for (int x: a) //Xét mọi phần tử x trong a
            {
                int v = u + x; //Thử xét giá trị nếu cộng u với x
                //Nếu giá trị v nằm trong phạm vi hiệu lực và chưa loang đến
                if (v >= -lim && v <= lim && trace[v] == NA)
                {
                    trace[v] = x; //Lưu vết: Để biến đổi thành v cần phép cộng với x
                    if (v == t) return; //Nếu đã loang đến t thì xong
                    q.push_back(v); //Đẩy v vào tập q
                }
            }
    }
}
```

```

    }
    p.swap(q); //Đổi dữ liệu q sang p
    q.clear(); //Làm rỗng q, chuẩn bị loang mức tiếp theo
}

void Print() // In kết quả
{
    if (trace[t] == NA) //trace[t] == NA: Không thể loang được đến t
        cout << "Impossible!";
    else //Truy vết in kết quả
        for (int v = t; v != 0; v -= trace[v])
            cout << trace[v] << ' ';
}

int main()
{
    Enter();
    Solve();
    Print();
}

```

🌀 Cài đặt giải thuật loang bằng queue

Trong cách cài đặt giải thuật loang theo lớp, ta đã nhận xét rằng bất kỳ cấu trúc dữ liệu nào cho phép bổ sung phần tử và duyệt các phần tử chứa trong đều có thể sử dụng để biểu diễn tập giá trị ở một mức nào đó. Nếu sử dụng cấu trúc dữ liệu queue, cơ chế thực hiện của giải thuật đơn giản là rút lần lượt các phần tử ra khỏi queue tại mức trước và đẩy các phần tử mới sinh vào queue ứng với mức sau. Do việc lấy ra và đẩy vào queue được thực hiện ở hai đầu khác nhau, ta có thể không cần dùng hai queue p, q như cách cài đặt trên mà sử dụng một queue duy nhất cho phép loang. Điều này khiến phép cài đặt ngắn gọn hơn:

Ban đầu với queue chỉ gồm số 0, số 0 được đánh dấu là đã loang đến. Tại mỗi bước ta rút một phần tử u khỏi (đầu) queue, lần lượt cộng u với các phần tử trong A , nếu sinh ra được một giá trị mới thì giá trị này lại được đánh dấu đã loang đến và được đẩy vào (cuối) queue.

Thuật toán sẽ kết thúc khi giá trị t đã được loang đến (tìm ra cách biến đổi) hoặc khi queue rỗng (không loang tiếp được nữa)

🌀 NUMTRANS2.CPP ✓ Biến đổi số 🌀

```

#include <iostream>
#include <vector>
#include <queue>
using namespace std;
const int NA = 1e5 + 1;
const int lim = 1e5;
int n, t;
vector<int> a;
int _trace[4 * lim + 1], *trace;

void Enter() //Nhập dữ liệu, không khác gì trước
{
    cin >> n >> t;
    a.resize(n);
    for (int& x: a) cin >> x;
}

```

```

        trace = _trace + lim;
        fill(begin(_trace), end(_trace), NA);
    }

void Solve() //Thuật toán loang
{
    queue<int> q; //Queue cho thuật toán loang
    trace[0] = 0; //Giá trị 0 coi như đã loang đến
    if (t == 0) return; //Nếu t == 0 không phải làm gì cả
    q.push(0); //Queue ban đầu chỉ gồm số 0
    while (!q.empty()) //Chứng nào queue còn phần tử
    {
        int u = q.front(); q.pop(); //lấy u từ đầu queue
        for (int x: a) //Xét mọi phần tử x trong a
        {
            int v = u + x; //Thử xét giá trị nếu cộng u với x
            //Nếu giá trị v nằm trong phạm vi hiệu lực và chưa loang đến
            if (v >= -lim && v <= lim && trace[v] == NA)
            {
                trace[v] = x; //Lưu vết: Để biến đổi thành v cần phép cộng với x
                if (v == t) return; //Nếu đã loang đến t thì xong
                q.push(v); //Đẩy v vào cuối queue
            }
        }
    }
}

void Print() // In kết quả, không khác gì trước
{
    if (trace[t] == NA)
        cout << "Impossible!";
    else
        for (int v = t; v != 0; v -= trace[v])
            cout << trace[v] << ' ';
}

int main()
{
    Enter();
    Solve();
    Print();
}

```

⊗ Tính đúng và độ phức tạp tính toán

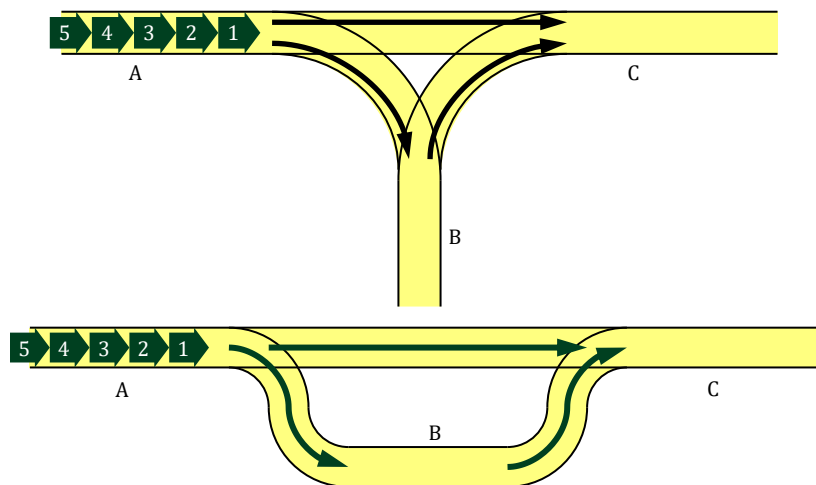
Có thể thấy rằng thuật toán có khả năng quét hết các giá trị sinh ra được thuộc phạm vi hiệu lực $[-10^5 \dots 10^5]$, ngoài ra khi một giá trị v được loang đến thì mức của giá trị đó tương ứng với số phép biến đổi tối thiểu cần thực hiện để biến 0 thành v . Từ đó suy ra tính đúng của giải thuật.

Mỗi giá trị được loang đến chắc chắn nằm trong phạm vi hiệu lực và sẽ không có giá trị nào được loang đến hai lần nhờ cơ chế đánh dấu. Khi dùng một giá trị u đã loang đến để tìm kiếm những giá trị mới phát sinh, ta cần xét tối đa n cách cộng thêm u với từng phần tử $\in A$. Vì vậy thời gian thực hiện giải thuật là $O(n \times L)$ với L là tổng số phần tử loang đến (nhỏ hơn hoặc bằng độ dài phạm vi hiệu lực).

Cũng có thể đánh giá riêng trên hai mô hình cài đặt: Ở phương pháp loang theo lớp, tổng số lượng phần tử ở các mức không vượt quá độ dài phạm vi hiệu lực và với mỗi phần tử ở mỗi mức ta phải xét n khả năng sinh phần tử ở mức kế tiếp. Ở phương pháp loang bằng queue, mỗi phần tử trong phạm vi hiệu lực được đưa vào queue tối đa 1 lần và lấy khỏi queue tối đa 1 lần. Khi một phần tử u được lấy ra, ta cũng phải xét n khả năng sinh phần tử mới bằng cách cộng thêm u với từng phần tử $\in A$. Vậy cả hai cách cài đặt đều có thời gian thực hiện $O(n \times L)$ với L là độ dài phạm vi hiệu lực.

Bài tập 2-1.

Có hai sơ đồ đường ray xe lửa bố trí như hình sau:



Ban đầu có n toa tàu xếp theo thứ tự từ 1 tới n từ phải qua trái trên đường ray A. Người ta muốn xếp lại các toa tàu theo thứ tự mới từ phải qua trái (p_1, p_2, \dots, p_n) lên đường ray C theo nguyên tắc: Các toa tàu không được “vượt nhau” trên ray, mỗi lần chỉ được chuyển một toa tàu từ $A \rightarrow B$, $A \rightarrow B$ hoặc $A \rightarrow C$. Hãy cho biết điều đó có thể thực hiện được trên sơ đồ đường ray nào trong hai sơ đồ trên.

Bài tập 2-2.

Có n người xếp hàng dọc đánh số từ 1 tới n từ đầu hàng tới cuối hàng, người thứ i có chiều cao là h_i . Ta nói hai người i, j nhìn thấy nhau nếu giữa hai người đó không tồn tại người nào khác có chiều cao $\geq \min\{h_i, h_j\}$, hay nói cách khác, tất cả những người đứng giữa người i và người j (nếu có) đều có chiều cao thấp hơn cả hai người này.

Yêu cầu: Tìm thuật toán $O(n)$ đếm số cặp chỉ số i, j ($i < j$) mà hai người i, j nhìn thấy nhau

Bài tập 2-3.

Yêu cầu như Bài tập 2-2. Nhưng với định nghĩa hai người (i, j) là nhìn thấy nhau nếu trong số những người đứng giữa i và j không có người nào thực sự cao hơn i và cũng không có người nào thực sự cao hơn j .

Bài tập 2-4.

Cho một bảng kích thước $m \times n$ được chia thành lưới ô vuông đơn vị. Mỗi ô được tô bởi một trong hai màu: Đen (B) hoặc Trắng (W). Tìm thuật toán $O(m \times n)$ xác định một hình chữ nhật có diện tích lớn nhất thỏa mãn các điều kiện: Cạnh hình chữ nhật song song với cạnh bảng, đồng thời hình chữ nhật chiếm trọn một số ô của bảng và chỉ gồm các ô trắng

Bài tập 2-5.

Cho dãy số nguyên $A = (a_0, a_1, \dots, a_{n-1})$ và một số nguyên S các phần tử a_i và S đều có giá trị tuyệt đối không vượt quá L . Hãy chọn ra một dãy con gồm ít phần tử nhất của A có tổng bằng S . Yêu cầu tìm thuật toán với độ phức tạp $O(n \times L)$

Bài tập 2-6.

Cho số nguyên dương n và một tập S gồm các chữ số thập phân $\{0 \dots 9\}$. Hãy tìm thuật toán $O(n \times |S|)$ để xác định số nguyên dương m thỏa mãn các điều kiện sau đây:

- ✿ m có biểu diễn thập phân chỉ gồm các chữ số trong tập S .
- ✿ m chia hết cho n
- ✿ m nhỏ nhất có thể

Bài tập 2-7.

Một xe chạy trên con đường dài n km tính từ km số 0 (nơi xuất phát) tới km số n (nơi kết thúc). Ở mỗi mốc km đều có trạm xăng, giá xăng ở mốc km thứ i là c_i một lít. Bình xăng của xe có dung tích k , mỗi lít xăng chỉ cho phép chạy 1km. Hãy tìm cách đổ xăng để thực hiện hành trình với chi phí thấp nhất biết rằng khi xuất phát bình xăng được đổ đầy. Yêu cầu thuật toán với độ phức tạp $O(n \times k)$

Bài tập 2-8.

Cho dãy số nguyên không âm $A = (a_0, a_1, \dots, a_{n-1})$ và một số nguyên không âm S . Hãy chọn ra một dãy con $a_{i_1}, a_{i_2}, \dots, a_{i_k}$ của dãy A gồm ít phần tử nhất thỏa mãn:

$$a_{i_1} \oplus a_{i_2} \oplus \dots \oplus a_{i_k} = S$$

Ở đây \oplus là ký hiệu phép toán XOR (exclusive – OR), trong C++ ký hiệu là “^”.

Yêu cầu tìm thuật toán với độ phức tạp $O(n \times \max\{a_i\})$