

Chương 6. Ngăn xếp và Hàng đợi

Ngăn xếp và Hàng đợi là hai kiểu dữ liệu trừu tượng rất quan trọng và được sử dụng nhiều trong thiết kế thuật toán. Về bản chất, ngăn xếp và hàng đợi 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 ngăn xếp và hàng đợi cũng như 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 ngăn xếp và hàng đợi là T .

6.1. Ngăn xếp

Ngăn xếp (stack) 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 ngăn xếp 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”, ngăn xếp còn có tên gọi là *danh sách kiểu LIFO (Last In First Out)*. Vị trí cuối danh sách được gọi là *đỉnh (top)* của ngăn xếp.

Đối với ngăn xếp có các thao tác cơ bản:

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

6.1.1. Biểu diễn ngăn xếp bằng mảng

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

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

Các thao tác cơ bản của ngăn xếp:

```
void init() //Khởi tạo ngăn xếp rỗng
{
    n = 0;
}

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

T& top() //Truy cập phần tử ở đỉnh ngăn xếp
{
    return Items[n - 1];
}
```

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

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

6.1.2. Biểu diễn ngăn xếp 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:

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

PNode head; //Chốt của danh sách móc nối

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

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

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

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

void pop() //Loại bỏ phần tử ở đỉnh ngăn xếp
{
    PNode p = head;
    head = head->link;
    delete p;
}
```

6.2. Hàng đợi

Hàng đợi (queue) 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 hàng đợi, 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 hàng đợi 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”, hàng đợi còn có tên gọi là *danh sách kiểu FIFO (First In First Out)*.

Các thao tác cơ bản trên hàng đợi:

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

6.2.1. Biểu diễn hàng đợi bằng mảng

Ta có thể biểu diễn hàng đợi bằng một mảng *Items* để lưu các phần tử trong hàng đợi, một biến nguyên *i* để lưu chỉ số phần tử đầu hàng đợi và một biến nguyên *j* để lưu chỉ số đứng sau phần tử cuối hàng đợi. Chỉ một phần của mảng *Items* từ vị trí *i* tới *j* – 1 được sử dụng lưu trữ các phần tử trong hàng đợi. Ví dụ:

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

Các thao tác cơ bản trên hàng đợi:

```
void init() //Khởi tạo hàng đợi rỗng
{
    i = j = 0;
}

bool empty() //Kiểm tra hàng đợi có rỗng không
{
    return i == j;
}

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

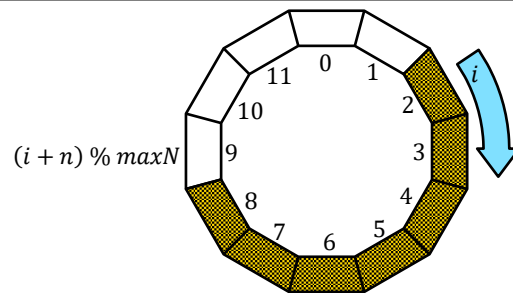
void push(const T &x) //Đẩy x vào hàng đợi
{
    Items[j++] = x;
}

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

6.2.2. Biểu diễn hàng đợi bằng danh sách vòng

Xét việc biểu diễn ngăn xếp và hàng đợi 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 ngăn xếp rỗng ($top == 0$), 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 số phần tử của ngăn xếp sẽ được tăng lên thành 6, rồi giảm về 2, sau đó lại tăng lên thành 10. Nhưng nếu ta thực hiện các thao tác đó đối với cách cài đặt hàng đợi như trên thì sẽ gặp lỗi tràn mảng, bởi mỗi lần đẩy phần tử vào hàng đợi (*push*), chỉ số cuối hàng đợi 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 $j - 1$ là thuộc hàng đợi, các phần tử ở vị trí khác là vô nghĩa.

Để khắc phục điều này, ta có thể biểu diễn hàng đợi bằng một danh sách vòng. Khi dùng mảng $Items[0 \dots maxN - 1]$ 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 hàng đợi được xếp quanh vòng tròn theo một chiều nào đó (chẳng hạn chiều kim đồng hồ) bắt đầu từ vị trí i . Nếu số phần tử trong hàng đợi là n thì việc đẩy thêm một phần tử vào hàng đợi tương đương với việc ta đặt phần tử vào vị trí $(i + n) \% maxN$ và tăng n lên 1; việc lấy ra một phần tử trong hàng đợi tương đương với việc lấy ra phần tử tại vị trí i rồi dịch chỉ số i theo chiều vòng ($i = (i + 1) \% maxN$). (Hình 6-1)



Hình 6-1. Dùng danh sách vòng mô tả hàng đợi

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

Các thao tác cơ bản trên hàng đợi cài đặt trên danh sách vòng có thể cài đặt như sau:

```
void init() //Khởi tạo hàng đợi rỗng
{
    i = n = 0;
}

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

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

void push(const T &x) //Đẩy x vào hàng đợi
{
    Items[(i + n++) % maxN] = x;
}

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

6.2.3. Biểu diễn hàng đợi bằng danh sách móc nối đơn kiểu FIFO

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

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

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

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

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

void push(const T &x) //Đẩy x vào hàng đợi
{
    PNode 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 hàng đợi
{
    PNode p = head;
    head = head->link;
    delete p;
}
```

6.3. Hàng đợi hai đầu

Tổng quát hơn so với ngăn xếp và hàng đợi là kiểu dữ liệu trừu tượng *Hàng đợi hai đầu* (*Double-Ended Queue - DEqueue*). 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.

Hàng đợi hai đầu cung cấp các thao tác căn bản:

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

Hàng đợi hai đầu 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.

6.4. Một số chú ý về kỹ thuật cài đặt

Ngăn xếp, hàng đợi và hàng đợi hai đầu là những kiểu dữ liệu trừu tượng tương đối dễ cài đặt, các 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 ngăn xếp, hàng đợi và hàng đợi hai đầu.

6.4.1. Ngăn xếp trong C++ STL

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

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

6.4.2. Hàng đợi trong C++ STL

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

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

6.4.3. Hàng đợi hai đầu trong C++ STL

Lớp mẫu `std::deque<T>` là hàng đợi hai đầu 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` vào đầu `DEqueue`
- ✿ `void push_back(x)`: Đẩy `x` 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(...)`: 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(...)`: 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`

6.5. Một số bài toán ứng dụng

6.5.1. 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 xâu ký tự $S = s[0 \dots n]$ là một dãy ngoặc đúng. Với mỗi dấu ')', cho biết vị trí dấu '(' cặp với nó.

Input

Một dòng chứa xâu 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 xâu 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 xâu, 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 xâu (Thuật toán có thời gian thực hiện $O(n^2)$)

Bằng một ngăn xếp chứa vị trí các dấu ngoặc mở '(', ta có một thuật toán khác hiệu quả hơn: Duyệt xâu từ đầu đến cuối:

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

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();
        }
}
```

6.5.2. 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 ngăn xếp.

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

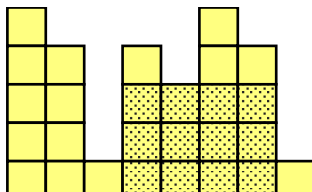
Input

- ✧ 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.

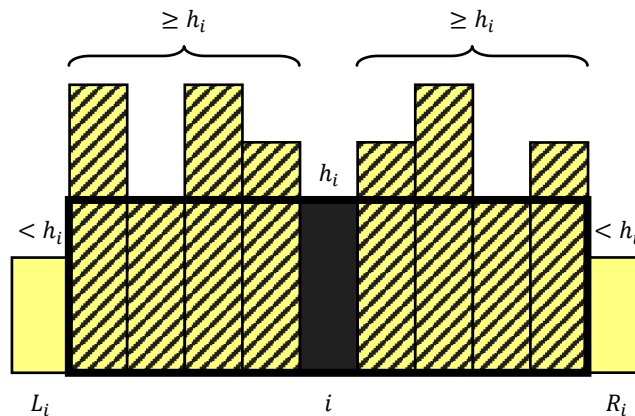
Nhận xét rằng hình chữ nhật có diện tích lớn nhất chắc chắn sẽ chứa trọn vẹn một cột nào đó (nếu không ta có thể giữ nguyên chiều rộng và nâng chiều cao lên thêm). Nếu hình chữ nhật cần tìm chứa trọn vẹn cột i thì chiều cao của nó chắc chắn bằng h_i . Chiều rộng của nó phủ qua cột i và vấn đề còn lại 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 6-2). 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 6-2. 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.

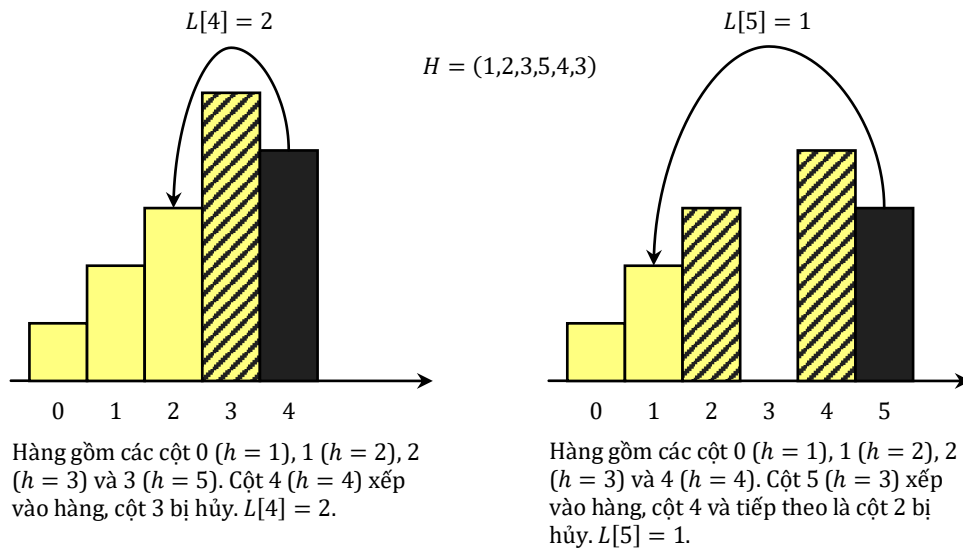
Mô hình xếp hàng: Tưởng tượng như ta cho các cột lần lượt xếp hàng với một hàng cột ban đầu được khởi tạo rỗng. Xếp lần lượt các cột vào hàng theo thứ tự $0, 1, 2, \dots, n - 1$. Trước khi xếp một cột i vào, ta hủy ngay cột đứng cuối hàng nếu thấy cột đứng cuối hàng

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 6-3 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)



Hình 6-3. Mô hình xếp hàng

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 có 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 $L[.]$ 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 ngăn xếp chứa các chỉ số cột với đỉnh ngăn xếp là phần tử đứng cuối hàng: Bắt đầu với một ngăn xếp rỗng, ta xét lần lượt các cột. Khi xét tới một cột i ta lần lượt loại bỏ chỉ số ở đỉnh ngăn xếp nếu nó ứng với cột có độ cao $\geq h_i$, cuối cùng ta ghi nhận $L[i]$ là chỉ số ở đỉnh ngăn xếp (hoặc -1 nếu ngăn xếp đã rỗng) và đẩy cột i vào ngăn xếp.

```
Stack = ∅;
for (i = 0, 1, ..., n - 1)
{
    while (Stack ≠ ∅ && chỉ số cột ở đỉnh ngăn xếp ứng với cột >= h[i])
        «Lấy ra chỉ số cột ở đỉnh ngăn xếp»;
    L[i] = Stack == ∅ ? -1 : «chỉ số cột ở đỉnh ngăn xếp»;
    «Đẩy chỉ số i vào đỉnh ngăn xếp»;
}
```

Thời gian thực hiện giải thuật có thể đánh giá qua số lệnh lấy ra phần tử ở đỉnh ngăn xếp (lệnh nằm trong lõi cả vòng lặp for và vòng lặp while). Rõ ràng mỗi chỉ số chỉ bị đẩy vào ngăn xếp 1 lần vì vậy cũng chỉ có tối đa n lần ta loại bỏ phần tử ở đỉnh ngăn xếp. Khi đánh giá bù trừ, thời gian thực hiện giải thuật là $O(n)$ do các thao tác cơ bản trên ngăn xếp 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();
        L[i] = Stack.empty() ? -1 : Stack.top(); //Tính chốt L[i];
        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();
        R[i] = Stack.empty() ? n : Stack.top(); //Tính chốt R[i]
        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();
}

```

6.5.3. 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.
- ✧ 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 một người bị đuổi được giải thích khá tự nhiên, một người đứng trước cao hơn hoặc bằng người i sẽ vô dụng trong việc tính s_i 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 người được xếp theo thứ tự tăng ngặt của chiều cao, người đứng đầu luôn là người thấp nhất của hàng. Nếu người đứng đầu nằm ngoài 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 tính min), anh ta sẽ bị đuổi và lấp lại với người đứng đầu mới. Khi quá trình này kết thúc, chiều cao của người đứng đầu 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 mà thôi vì nếu ta phải đuổi lần lượt hai người x, y ở đầu hàng thì người x chắc chắn đã bị người $i - 1$ đuổi ở bước trước do người x đứng 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.

🔗 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; //Hàng đợi hai đầu 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 ≥ 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 hàng đợi hai đầu: $q.pop_back()$, $q.pop_front()$ 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 DQueue có thể cài đặt để chạy trong thời gian $O(1)$.

6.5.4. 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 $A = a[0 \dots n)$ và một số nguyên dương 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.

Sample Input	Sample Output
3 10 -9 3 8	8 -9 8 3
2 7 -4 5	5 -4 5 -4 5
2 9 -6 8	Impossible!

* Thuật toán

Ta có thể mô hình hóa dưới dạng một máy biến đổi số với trạng thái ban đầu là số 0. Với luật biến đổi như trong bài, máy có thể biến đổi trạng thái từ số này sang số khác và ta cần tìm cách để đưa máy về trạng thái ứng với số t bằng ít phép biến đổi nhất.

Vì thứ tự các phép biến đổi không quan trọng, dựa vào ràng buộc dữ liệu: $1 \leq t \leq 10^5$ và $|a_i| \leq 10^5$ ($\forall i$), ta có nhận xét rằng với một phương án bất kỳ, ta có thể điều chỉnh lại thứ tự các phép biến đổi cộng với số dương hay cộng với số âm một cách hợp lý để các kết quả trung gian của s trong quá trình biến đổi luôn nằm trong phạm vi $[0 \dots 2 \times 10^5]$. Điều này cho phép ta mô tả tập trạng thái của một “máy” là các số nguyên trong phạm vi $[0 \dots 2 \times 10^5]$ gọi là phạm vi hiệu lực.

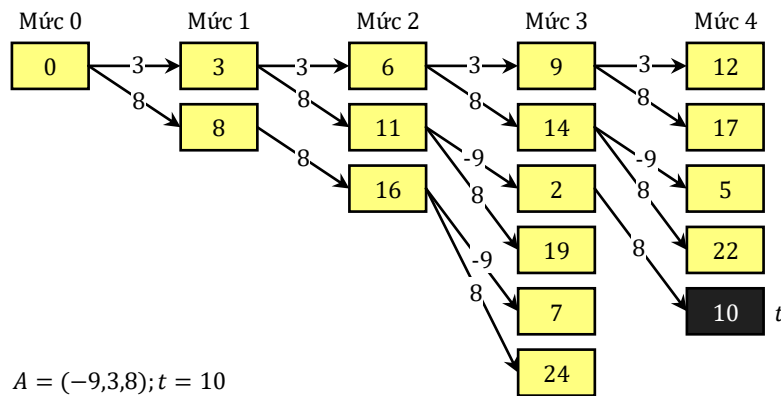
Giải thuật loang:

Ban đầu ta chỉ có một tập gồm đúng số 0 là trạng thái khởi đầu của máy, số 0 gọi là giá trị ở mức 0, tức là giá trị mà ta có thể có mà không cần qua một phép biến đổi nào. Số 0 được đánh dấu là đã loang đến.

Với giá trị 0 ở mức 0, ta lần lượt cộng nó với các phần tử trong A , nếu sinh ra được một giá trị mới trong phạm vi hiệu lực thì giá trị mới được đánh dấu là đã loang đến và được đưa vào tập các giá trị mức 1.

Tương tự như vậy với mỗi giá trị mức 1, ta lần lượt cộng nó với các phần tử trong A , nếu sinh ra được một giá trị mới trong phạm vi hiệu lực thì giá trị này lại được đánh dấu đã loang đến và được đưa vào tập các giá trị mức 2...

Cứ tiếp tục như vậy khi ta đã loang đến giá trị t hoặc đến một mức nào đó ta không loang tiếp được thêm phần tử mới nào nữa.



$$A = (-9, 3, 8); t = 10$$

Hình 6-4. Sơ đồ loang

Ví dụ với $A = (-9, 3, 8)$ và $t = 10$, các bước loang được thể hiện trong Hình 6-4. Khi quá trình loang kết thúc, ta biết giá trị 10 ở mức 4, tức là để biến đổi 0 thành 10 cần dùng 4 phép biến đổi.

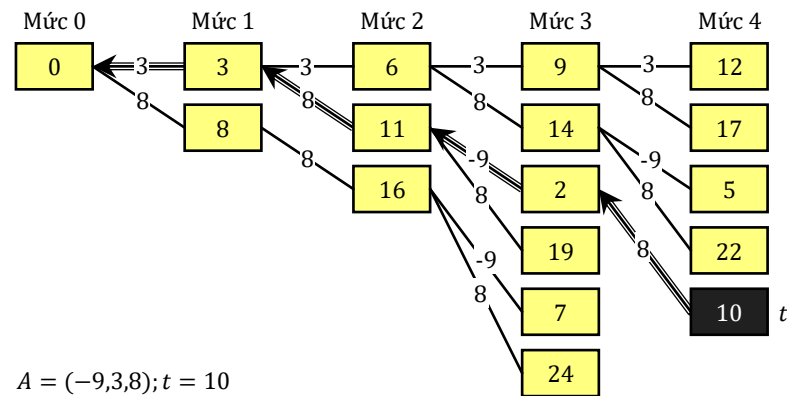
* Cài đặt giải thuật loang theo lớp

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. Trong trường hợp này ta sử dụng vector. Hai vector p và q được duy trì trong thuật toán với p chứa tập giá trị ở mức hiện tại đã loang đến và q sẽ được xây dựng từ p và dãy A để chứa tập giá trị ở mức tiếp theo sẽ loang đến. Sau mỗi bước lặp, giá trị trong q được đẩy sang cho p và q được làm rỗng để chuẩn bị cho bước loang mức tiếp theo...

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 $v_{\text{mới}} = v_{\text{cũ}} - trace[v_{\text{cũ}}]$ cho tới khi $v = 0$:


```
for (v = t; v != 0; v -= trace[v])
    cout << trace[v] << ' ';
```

Ví dụ với $A = (-9, 3, 8)$ và $t = 10$, quá trình truy vết trong Hình 6-5 cho biết để biến $s = 0$ thành 10 ta cần thực hiện lần lượt các phép cộng với 3, 8, -9 và 8 (đoạn chương trình trên sẽ đưa ra các phép biến đổi theo thứ tự ngược lại nhưng điều đó không quan trọng).



Hình 6-5. Sơ đồ truy vết

Cơ chế đánh dấu một giá trị thuộc phạm vi hiệu lực $[0 \dots 2 \times 10^5)$ 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.

NUMTRANS1.cpp ✓ Biến đổi số

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

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
    fill(begin(trace), end(trace), NA); //Tất cả các giá trị đều chưa loang đến
}

void Solve() //Thuật toán loang
{
    vector<int> p{0}, q; //Ban đầu p chỉ gồm giá trị 0
    trace[0] = 0; //Giá trị 0 đã loang đến
    if (t == 0) return; //Nếu t == 0 không phải làm gì cả
    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 >= 0 && 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 hàng đợi

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 loang ra ở mỗi mức. Nếu sử dụng cấu trúc dữ liệu hàng đợi, cơ chế thực hiện của giải thuật đơn giản là rút lần lượt các phần tử ra khỏi hàng đợi tại mức trước và đẩy các phần tử mới sinh vào hàng đợi ứng với mức sau. Do việc lấy ra và đẩy vào hàng đợi được thực hiện ở hai đầu khác nhau, ta có thể không cần dùng hai danh sách p, q như cách cài đặt trên mà sử dụng một hàng đợi duy nhất cho phép loang. Điều này khiến phép cài đặt ngắn gọn hơn:

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

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

✳ NUMTRANS2.cpp ✓ Biến đổi số ✳

```
#include <iostream>
#include <vector>
```



```
#include <queue>
using namespace std;
const int maxT = 1e5;
const int maxA = 1e5;
const int lim = maxT + maxA;
const int NA = maxA + 1;
int n, t;
vector<int> a;
int trace[lim];

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

void Solve() //Thuật toán loang
{
    queue<int> q({0}); //Hàng đợi cho thuật toán loang, ban đầu chỉ gồm số 0
    trace[0] = 0; //Giá trị 0 coi như đã loang đến
    if (t == 0) return; //Nếu t == 0 không phải làm gì cả
    do
    {
        int u = q.front(); q.pop(); //lấy u từ đầu hàng đợi
        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 >= 0 && 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 hàng đợi
            }
        }
    }
    while (!q.empty()) //Chừng nào hàng đợi còn phần tử
}

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 $[0 \dots 2 \times 10^5]$, ngoài ra khi một giá trị v được loang đến thì mức của giá trị đó

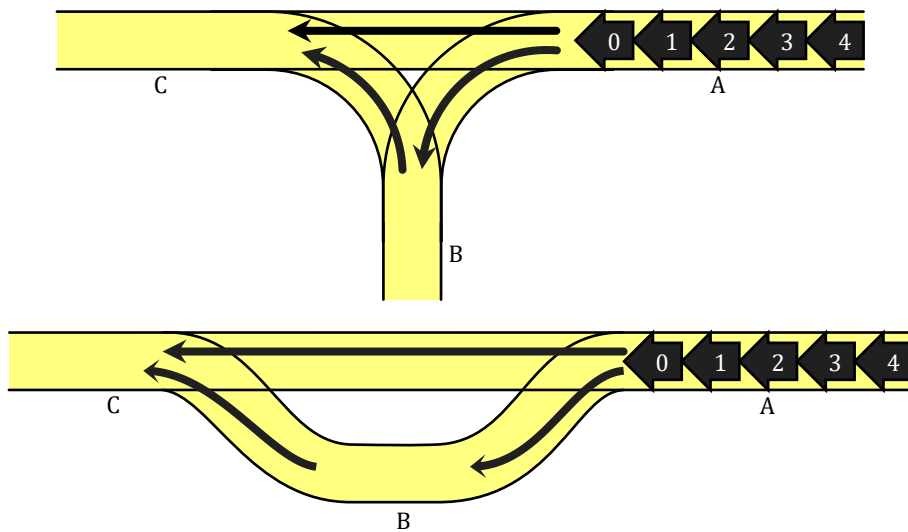
tương ứng với số phép biến đổi tối thiểu cần thực hiện để biến 0 thành v . Từ đó suy ra tính đúng của giải thuật.

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 hàng đợi, mỗi phần tử trong phạm vi hiệu lực được đưa vào hàng đợi tối đa 1 lần và lấy khỏi hàng đợi 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 6-1

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



Ban đầu có n toa tàu số hiệu từ 0 tới $n - 1$ xếp từ trái qua phải trên đường ray A. Người ta muốn xếp lại các toa tàu lên đường ray C để các toa có số hiệu p_0, p_1, \dots, p_{n-1} xếp theo đúng thứ tự từ trái qua phải. Các chuyển các toa phải tuân 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 6-2

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 6-3

Yêu cầu như Bài tập 6-2 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 6-4

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 6-5

Trong ví dụ về thuật toán loang (Mục 6.5.4), đưa ra chứng minh về cách chọn phạm vi hiệu lực, sửa đổi chương trình để thực hiện được ngay cả trong trường hợp giá trị t là số âm, cụ thể là $-10^5 \leq t \leq 10^5$.

Bài tập 6-6

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 6-7

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 6-8

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 rỗng. Yêu cầu thuật toán với độ phức tạp $O(n)$.

Gợi ý: Quản lý bình xăng, ban đầu đổ đầy bình, mỗi km ta sẽ tiêu thụ lít xăng mua rẻ nhất đang có trong bình. Tại mỗi trạm xăng, "bán lại" những lít xăng có giá cao hơn giá tại trạm xăng đó (với giá đã mua) và lại mua đầy bình. Chỉ cần quan tâm tới những lít xăng đã tiêu thụ.

Bài tập 6-9

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 $O(n \times \max\{a_i\})$