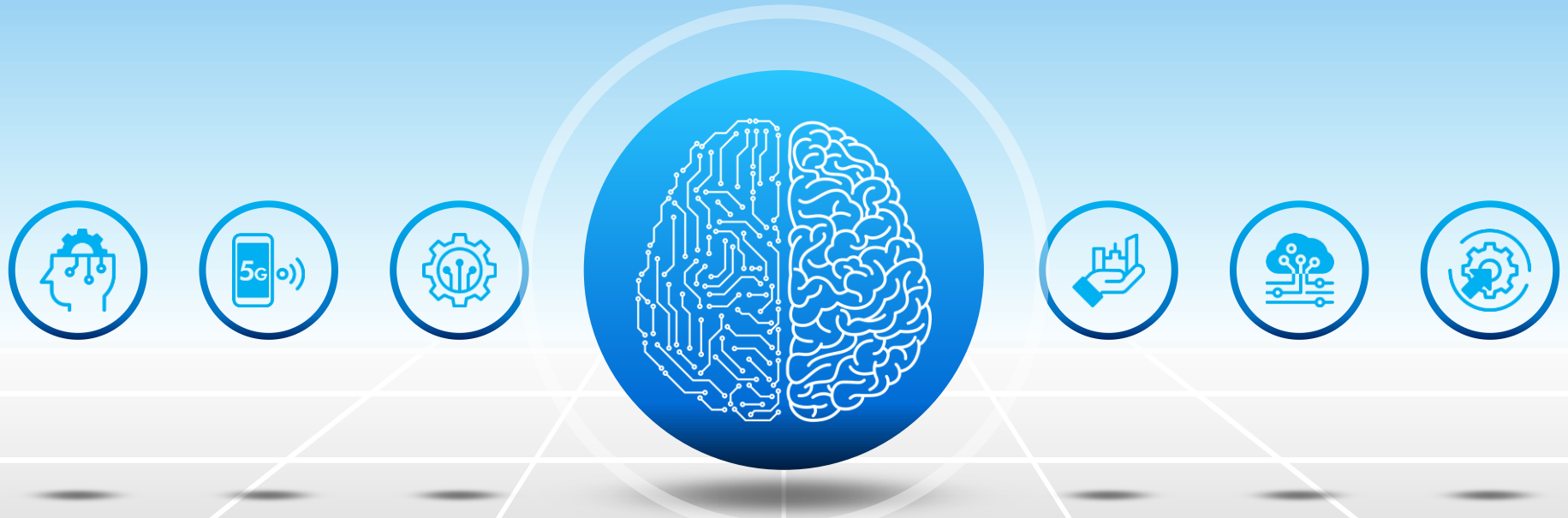
CIS3034 문제해결프로젝트

스택을 활용한 문제해결



목 차
CONTENTS

자료구조

자료구조의 정의

- **컴퓨터에서 처리할 자료를 효율적으로 관리하고 구조화시키기 위한 학문**
- **즉. 자료를 효율적으로 사용하기 위해 자료의 특성에 따라 분류하여 구성하고 저장 및 처리하는 모든 작업을 의미**
  - 자료 구조는 자료의 표현과 그것과 관련된 연산
  - 자료 구조는 일련의 자료들을 조직하고 구조화 하는 것
  - 자료 구조에 따라 프로그램 실행 시간이 달라짐

자료구조의 분류


자료구조
단순구조
정수
실수
문자
문자열
선형구조
리스트
연결리스트
단순 연결 리스트
이중 연결 리스트
원형 연결 리스트
스택
큐
덱
비선형구조
트리
일반트리
이진트리
그래프
방향그래프
무방향그래프
파일구조
순차파일
색인파일
직접파일

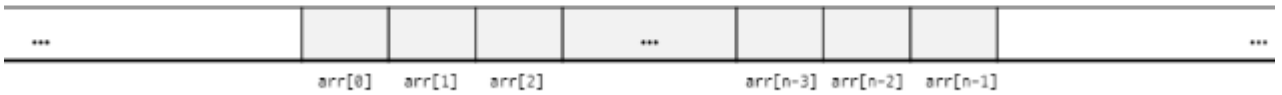
자료구조의 이용

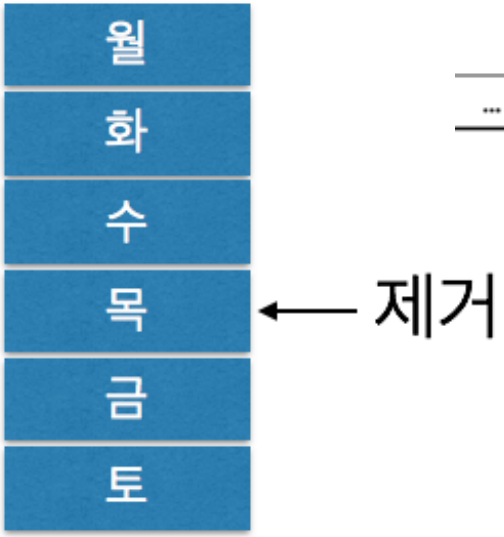
- 정렬(Sort) : 기억장치 내의 자료를 일정한 순서에 따라 나열하는 것
- 검색(Search) : 기억장치 내의 자료를 찾는 것
- 파일 편성 : 자료를 기억 매체에 저장할 때의 파일 구조
- 인덱스 : 파일에서 특정 자료를 빠르게 찾기 위한 색인표
- 데이터의 특징과 사용 용도에 알맞은 자료구조를 선택해야 함.
- 언어별로 바로 사용할 수 있는 형태로 제공
  - 배열, 리스트, 맵, 세트, 스택, 큐 등

리스트

- **선형 리스트(Linear List) : 배열**
  - 배열과 같이 연속되는 기억 장소에 저장되는 리스트
  - 다음과 같은 특징을 가짐
    - 가장 간단한 자료구조
    - 접근 속도가 빠름
    - 기억 장소를 연속으로 배정받기 때문에 기억 공간 효율이 가장 좋음
    - 삽입, 삭제 시 자료의 이동이 필요하기 때문에 작업이 번거로움.

```
int[] arr = new int[n];
```



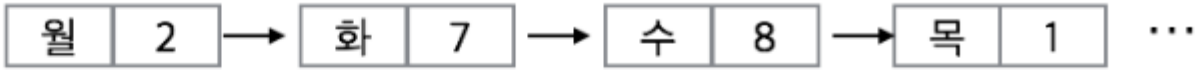


리스트

- **연결 리스트(Linked List)**
  - 자료들을 반드시 연속적으로 배열시키지는 않고, 임의의 기억 공간에 기억시키되, 자료 항목의 순서에 따라 노드의 포인터 부분을 이용하여 서로 연결시킴.
  - 특징
    - 노드의 삽입, 삭제 작업 용이
    - 연결을 위한 링크 부분이 필요하기 때문에 선형 리스트에 비해 기억 공간 효율이 좋지 않음.
    - 포인터 찾는 시간 필요하기 때문에 접근 속도 느림
    - 트리를 표현할 때 가장 적합한 구조
    - 종류: 단순 연결 리스트, 이중 연결 리스트 등...

- **연결 리스트**(Linked List)
  - 구조 예시

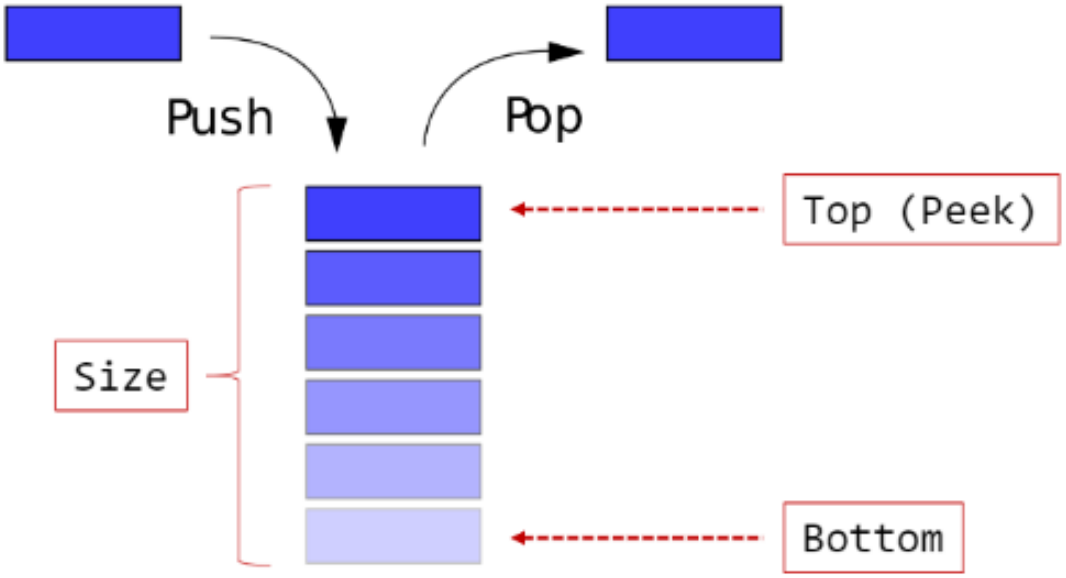
금	10
화	7
월	2
수	8
목	1
일	^



스택을 활용한 문제해결

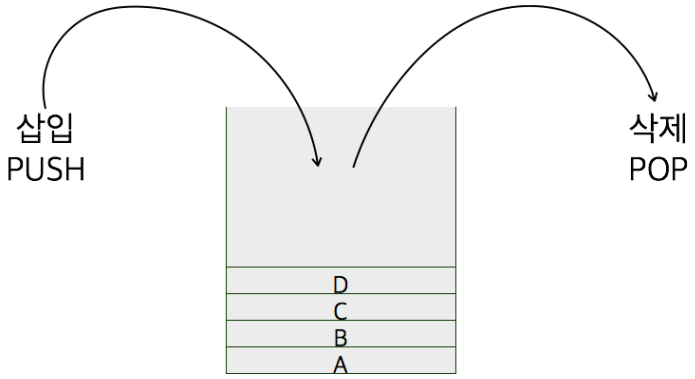
스택

- **스택의 개념**
  - 리스트의 한 쪽 끝(Top)으로만 자료의 삽입과 삭제가 이루어짐
  - 가장 나중에 삽입된 자료가 가장 먼저 삭제되는 후입선출 (LIFO : Last In First Out)
  - 내부 구현은 보통 Linked List 로 되어 있음.



스택

- **스택의 사용 사례**
  - 재귀 알고리즘
    - 재귀적으로 함수를 호출해야 하는 경우 임시 데이터를 스택에 넣음.
    - 재귀 함수를 빠져나와 검색을 할 때는 스택에 넣어 두었던 임시 데이터를 빼 준다.
    - 스택은 재귀 알고리즘을 반복적 형태를 통해서 구현할 수 있게 함.
  - 웹 브라우저 방문 기록(뒤로 가기), 실행 취소(Undo)
    - 인터럽트 발생 시 복귀 주소를 저장
  - 역순 문자열 만들기
  - 수식의 괄호 검사 (실습)
    - 올바른 괄호 문자열(VPS) 판단하기
  - 후위표기법
    - 연산자를 피연산자의 뒤에 놓는 방법



- ((1+(2*3)+((4+2)/2))) => ((1(23)*)+((42)+2)/))+ // 연산자를 괄호뒤로
  => 123*+42+2/+ (괄호삭제)

스택

- **스택의 사용 사례**
  - 헤더파일 추가

    #include <stack>

  - 표준 네임스페이스 사용하면 편리

    using namespace std;

  - 스택 변수 선언

    stack<datatype> 변수명;

    stack<char> st;

- **bool empty() const;**
  - 비어있는지 확인.

  ex) st.empty();

- **size_type size() const;**
  - 사이즈 반환

  ex) st.size();

- **value_type& top();**
- **const value_type& top() const;**
  - 맨위에 있는 인자 반환

  ex) st.top();

- **void push(const value_type& val);**
  - 데이터(value) 삽입

  ex) st.push(999);

- **void pop();**
  - top 이 가리키는 원소를 삭제합니다.

  ex) st.pop();

스택

- 스택의 사용 사례

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

int main(void){
    stack<int> st;

    st.push(10);
    st.push(20);
    st.push(30);
    st.push(40);
    st.push(50);

    cout << "st.size() : " << st.size() << endl;
    cout << "st.top() -> st.pop()" << endl;
    while(!st.empty()){
        cout << st.top() << endl;
        st.pop();
    }
    return 0;
}
```

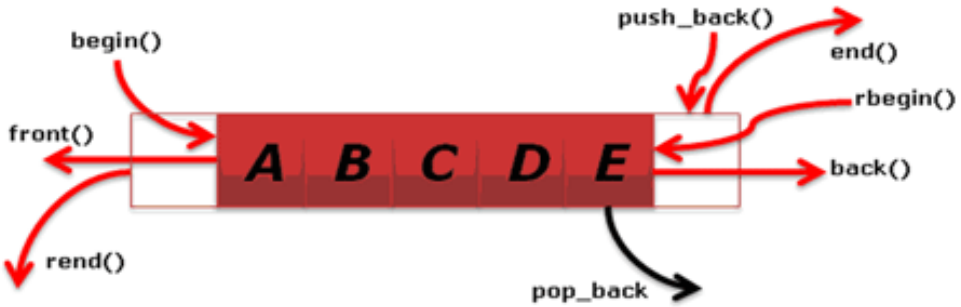
멤버 함수
empty()
size()
top()
push()
pop()

- 결과.

```
st.size() : 5
st.top() -> st.pop()
 50
 40
 30
 20
 10
```

스택

- **스택 vs. 벡터**
  - 데크 컨테이너 - 내부 구조가 리스트
  - 벡터 컨테이너 - 내부 구조가 동적 배열
  - 평균적으로 데크가 벡터보다 더 빠르게 동작
  - 스택은 LIFO만 가능, 벡터는 맨 앞 데이터 삽입(push_front)/삭제(pop_front) 및 중간 데이터 삽입(insert)/삭제(erase)도 가능
  - LIFO 로만 변수를 사용할 경우, 스택으로 선언하는 것이 코드의 가독성에 유리함.
  - 벡터로 선언할 경우, 배열로도 사용할 수 있기 때문에 목적이 스택이라 해도 바로 알기 어려움.
  - 용도에 따라 적절한 컨테이너 사용이 필요



스택 예제 풀이

문제6A. 괄호문자열

괄호 문자열(Parenthesis String, PS)은 두 개의 괄호 기호인 '(' 와 ')' 만으로 구성되어 있는 문자열이다. 그 중에서 괄호의 모양이 바르게 구성된 문자열을 올바른 괄호 문자열(Valid PS, VPS)이라고 부른다. 한 쌍의 괄호 기호로 된 "( )" 문자열은 기본 VPS 이라고 부른다. 만일 x 가 VPS 라면 이것을 하나의 괄호에 넣은 새로운 문자열 "(x)"도 VPS 가 된다. 그리고 두 VPS x 와 y를 접합(concatenation)시킨 새로운 문자열 xy도 VPS 가 된다. 예를 들어 "(())()"와 "((()))" 는 VPS 이지만 "(()(", "(())())))" , 그리고 "(()" 는 모두 VPS 가 아닌 문자열이다.

여러분은 입력으로 주어진 괄호 문자열이 VPS 인지 아닌지를 판단해서 그 결과를 YES 와 NO 로 나타내어야 한다.

문제6A. 괄호문자열

입력 형식

입력 데이터는 표준 입력을 사용한다. 입력은 T개의 테스트 데이터로 주어진다. 입력의 첫 번째 줄에는 입력 데이터의 수를 나타내는 정수 T가 주어진다. 각 테스트 데이터의 첫째 줄에는 괄호 문자열이 한 줄에 주어진다. 하나의 괄호 문자열의 길이는 2 이상 50 이하이다.

출력 형식

출력은 표준 출력을 사용한다. 만일 입력 괄호 문자열이 올바른 괄호 문자열(VPS)이면 `YES`, 아니면 `NO`를 한 줄에 하나씩 차례대로 출력해야 한다.

문제6A. 괄호문자열

입력

```
6
(())())
(((()())()
(()())((()))
((()()(()))(((())))()
()()()()(()()())()
((()(()))()(
```

출력

```
NO
NO
YES
NO
YES
NO
```

문제6A. 괄호문자열

```cpp
#include <iostream>
#include <string>
#include <stack>
#include <vector>

using namespace std;


class Parenthesis {
public:
  bool type;  // 열린 괄호면 true, 닫힌 괄호면 false
  int index;  // 해당 괄호의 인덱스

  Parenthesis(int index, bool type) {
    this->index = index;
    this->type = type;
  }

  Parenthesis(int index, char c) {
    this->index = index;
    if (c == '(') {
      this->type = true;
    } else {
      this->type = false;
    }
  }
};
```

```cpp
bool isValidParentheses(const vector<Parenthesis>& parentheses) {
  // 현재 짝을 찾지 못한 열린 괄호들
  stack<Parenthesis> myStack;

  return true;
}

void process(int caseIndex) {
  string str;
  cin >> str;

  vector<Parenthesis> parentheses;
  for (int i = 0; i < str.length(); ++i) {
    parentheses.push_back(Parenthesis(i,str[i]));
  }

  bool isValid = isValidParentheses(parentheses);

  if (isValid) {
    cout << "YES" << endl;
  } else {
    cout << "NO" << endl;
  }
}

int main() {
  int caseSize;
  cin >> caseSize;

  for (int caseIndex = 1; caseIndex <= caseSize; caseIndex += 1) {
    process(caseIndex);
  }
}
```

문제6B. 탑

KOI 통신연구소는 레이저를 이용한 새로운 비밀 통신 시스템 개발을 위한 실험을 하고 있다. 실험을 위하여 일직선 위에 N개의 높이가 서로 다른 탑을 수평 직선의 왼쪽부터 오른쪽 방향으로 차례로 세우고, 각 탑의 꼭대기에 레이저 송신기를 설치하였다. 모든 탑의 레이저 송신기는 레이저 신호를 지표면과 평행하게 수평 직선의 왼쪽 방향으로 발사하고, 탑의 기둥 모두에는 레이저 신호를 수신하는 장치가 설치되어 있다. 하나의 탑에서 발사된 레이저 신호는 가장 먼저 만나는 단 하나의 탑에서만 수신이 가능하다.

예를 들어 높이가 6, 9, 5, 7, 4인 다섯 개의 탑이 수평 직선에 일렬로 서 있고, 모든 탑에서는 주어진 탑 순서의 반대 방향(왼쪽 방향)으로 동시에 레이저 신호를 발사한다고 하자. 그러면, 높이가 4인 다섯 번째 탑에서 발사한 레이저 신호는 높이가 7인 네 번째 탑이 수신을 하고, 높이가 7인 네 번째 탑의 신호는 높이가 9인 두 번째 탑이, 높이가 5인 세 번째 탑의 신호도 높이가 9인 두 번째 탑이 수신을 한다. 높이가 9인 두 번째 탑과 높이가 6인 첫 번째 탑이 보낸 레이저 신호는 어떤 탑에서도 수신을 하지 못한다.

탑들의 개수 N과 탑들의 높이가 주어질 때, 각 각의 탑에서 발사한 레이저 신호를 어느 탑에서 수신하는지를 알아내는 프로그램을 작성하라.

문제6B. 탑

입력 형식

첫째 줄에 탑의 수를 나타내는 정수 N이 주어진다. N은 1 이상 500,000 이하이다. 둘째 줄에는 N개의 탑들의 높이가 직선상에 놓인 순서대로 하나의 빈칸을 사이에 두고 주어진다. 탑들의 높이는 1 이상 100,000,000 이하의 정수이다.

출력 형식

첫째 줄에 주어진 탑들의 순서대로 각각의 탑들에서 발사한 레이저 신호를 수신한 탑들의 번호를 하나의 빈칸을 사이에 두고 출력한다. 만약 레이저 신호를 수신하는 탑이 존재하지 않으면 `0`을 출력한다.

입력

```
5
6 9 5 7 4
```

출력

```
0 0 2 2 4
```

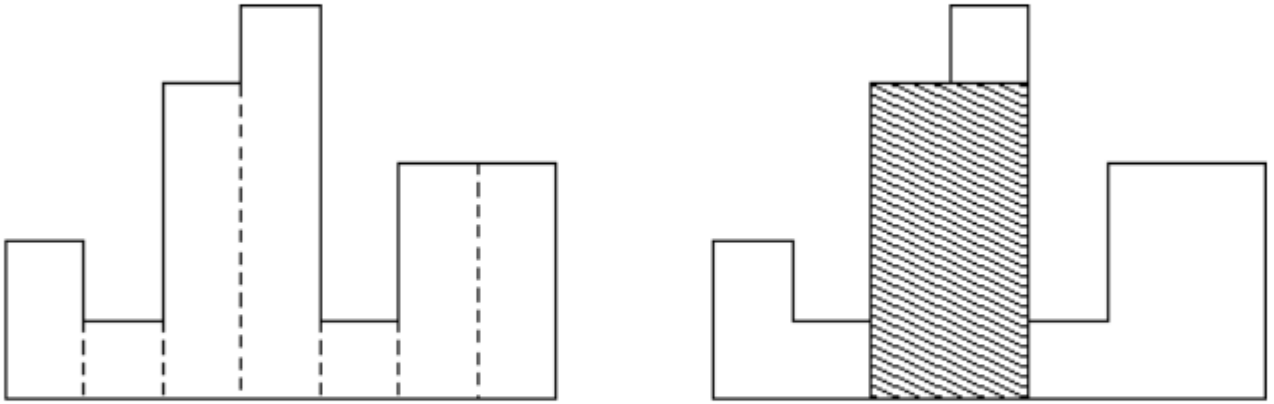
문제6B. 탑

```cpp
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

class Tower {
public:
  int index; // 타워의 인덱스
  int height; // 타워의 높이
  int targetTowerIndex; // 이 타워의 레이저를 수신하는

  Tower(int index, int height) {
    this->index = index;
    this->height = height;
    this->targetTowerIndex = 0;
  }

  void setTargetTowerIndex(int targetTowerIndex) {
    this->targetTowerIndex = targetTowerIndex;
  }

  int getTargetTowerIndex() {
    return this->targetTowerIndex;
  }
};
```

```cpp
void findTargetTowers(vector<Tower>& towers) {
  // 현재 다른 타워의 신호를 수신할 가능성이 있는 타워
  stack<Tower> touchableTowers;
}

int main() {
  int n;
  cin >> n;

  vector<Tower> towers;
  for(int i = 0; i < n; ++i) {
    int hi;
    cin >> hi;
    towers.push_back(Tower(i + 1, hi)); // 인덱스
  }

  // 각 타워가 송신하는 레이저에 대해 타겟을 모두 계산
  findTargetTowers(towers);

  for(int i = 0 ; i < n; i ++) {
    if(i > 0) {
      cout << " ";
    }

    Tower t = towers[i];
    int targetIndex = t.getTargetTowerIndex();
    cout << targetIndex;
  }
}
```

문제6C. 히스토그램

아래에서 보는 그림처럼 서로 폭이 모두 같은 직사각형들이 서로 밀착하여 바닥에 붙어있는 모양을 상상해보자. 이 도형의 내부를 2차원적으로 생각해 보았을 때, 이 히스토그램의 바닥면과 옆면의 방향에 평행한 변을 가지는 직사각형이 존재할 수 있다.



<직사각형으로 이루어진 히스토그램(왼쪽)과 그 내부에 존재하는 직사각형의 예시(오른쪽)>

그렇다면 이러한 직사각형 들이 히스토그램 내부에서 가질 수 있는 가장 큰 넓이는 얼마일까? 이를 계산하는 프로그램을 작성해보자.

문제6C. 히스토그램

입력 형식

첫 줄에는 테스트케이스의 수를 나타내는 자연수 T가 주어진다. 이후 총 T개의 테스트케이스에 대한 입력이 순서대로 주어지며 각 테스트케이스는 총 두 줄의 입력으로 이루어 져 있다.

- 테스트케이스의 첫 줄에는 히스토그램을 구성하는 직사각형 모양 기둥의 수를 나타내는 10만 이하의 자연수 N이 주어진다.
- 테스트케이스의 두 번째 줄에는 각 직사각형 모양 기둥의 높이가 공백으로 구분되어 왼쪽부터 순서대로 주어진다.
    - 각 직사각형의 높이는 10만 이하의 자연수이다.

출력 형식

각 테스트케이스 별로 한 줄에 히스토그램 내부에 존재할 수 있는 직사각형의 최대 넓이를 계산하여 출력한다.

입력

```
2
7
2 1 4 5 1 3 3
4
1000 1000 1000 1000
```

출력

```
8
4000
```

문제6C. 히스토그램

```cpp
#include <iostream>
#include <vector>
#include <stack>
#include <cmath>

using namespace std;

class Histogram {
public:
  int height;    // 히스토그램의 높이
  int leftX;     // 인덱스 혹은 히스토그램의 왼쪽
  int rightX;    // 히스토그램의 오른쪽 변의 x좌표

  Histogram() { }

  Histogram(int index, int height) {
    this->leftX = index;
    this->rightX = this->leftX + 1;    // 각 히
    this->height = height;
  }

};
```

```cpp
long long getLargestRectangleArea(const vector<Histogram>& histograms) {
  long long answer = 0;    // 최대 직사각형의 넓이

  // 현재 우측으로 확장 가능성이 있는 히스토그램들
  stack<Histogram> continuedHistograms;


  return answer;
}

void process(int caseIndex) {
  int n;
  cin >> n;

  vector<Histogram> histograms;
  for(int i = 0 ; i < n ; i ++) {
    int height;
    cin >> height;
    histograms.push_back(Histogram(i, height));
  }

  long long answer = getLargestRectangleArea(histograms);
  cout << answer << endl;
}

int main() {
  int caseSize;
  cin >> caseSize;

  for (int caseIndex = 1; caseIndex <= caseSize; caseIndex += 1) {
    process(caseIndex);
  }
}
```