알고리즘 강의 1주차입니다. 시간복잡도, 빅오표기법, 공간복잡도, 누적합, 구현까지 알아보겠습니다.

시간복잡도

복잡도는 시간복잡도와 공간복잡도로 나뉘어지는데 먼저 시간복잡도에 대해 알아보겠습니다.

시간복잡도란 입력크기에 대해 어떠한 알고리즘이 실행되는데 걸리는 시간이며 주요로직의 반복횟수를 중점으로 측정됩니다.

아니 시간이라고?
그렇다면 시간복잡도를 측정하기 위해서 항상 시간을 재야 할까요?
만약 어떠한 로직이 있고 그 로직에 걸리는 시간을 재려면 이렇게 재곤 해야 합니다.

```
console.time("test")
let sum = 0;
for(let i = 0; i < 1000000; i++){
  sum += 1;
}
console.timeEnd("test")
// test: 1.575ms
```

하지만 이러한 시간이라는 것은 컴퓨터 사양 등 여러가지 요소에 영향을 받곤 합니다.
그래서 시간복잡도를 설명할 때는 시간이 아니라 어떠한 알고리즘이 주어진 입력크기를 기반으로 어떠한 로직이 몇번 반복되었는가를 중점으로 설명합니다.

예를 들어 다음코드는 어떠한 시간복잡도를 가질까요?

```
for(int i = 0; i < 10; i++){
    for(int j =0; j < n; j++){
        for(int k = 0; k < n; k++){
            if(true) cout << k << '\n';
        }
    }
}
for(int i = 0; i < n; i++){
    if(true) cout << i << '\n';
}
```

if(true) cout << k << '\n';라는 로직이 이만큼 반복되는 것을 알 수 있습니다.

$$10n^2 + n$$

이것이 바로 시간복잡도입니다.

> 빅오표기법

앞의 코드는 다음과 같은 시간복잡도를 가진다고 했죠?

$$10n^2 + n$$

이를 빅오 표기법으로 나타내면 다음과 같습니다.

$$O(n^2)$$

빅오 표기법(Big - O notation) 이란 **복잡도에 가장 영향을 많이 끼치는 항의 상수인자를 빼고 나머지 항을 없애서 복잡도를 나타내는 표기법**입니다.

참고로 다른 표기법들도 있지만 가장 많이 쓰이는 것은 빅오표기법입니다.

예를 들어 다음과 같은 시간복잡도에서 복잡도에 가장 많은 영향을 끼치는 항은 무엇일까요?

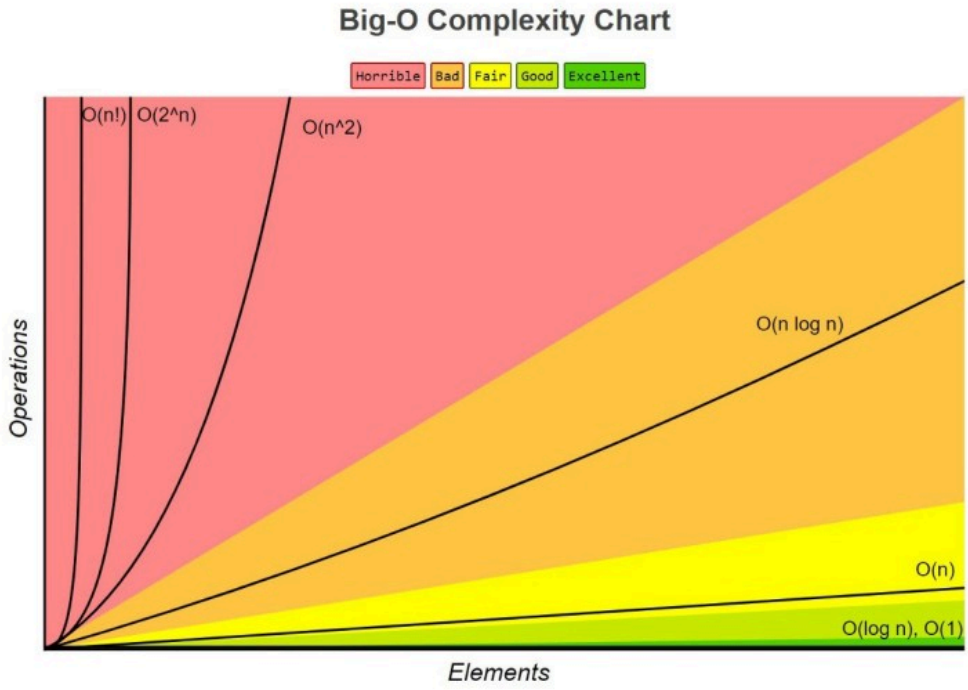
$$10n^2 + n$$

바로 n의 제곱항이고 다른 것은 그에 비해 미미하기 때문에 이것만 신경쓰면 된다는 것입니다.

아니, n도 신경써야 되지 않나요? 라고 생각할 수 있지만 입력크기가 커질 때의 연산이 커지는 것을 볼까요? 예를 들어 n이 1, 3, 9, 12로 증가한다고 할 때 n^2은 1, 9, 81, 144로 증가하죠? n ^ 2이 월등히 크게 증가하는 것을 볼 수 있습니다.

예를 들어서 10 * n ^2 + n^ 2정도의 시간복잡도를 빅오표기법으로 표현하면 어떻게 될까요? n^2이 되게 됩니다.

자 다음 그림을 외워볼까요? 이를 외우고 있어야 나중에 빅오표기법을 나타낼 때 쉽게 표현할 수 있습니다. n! > 2 ^ n > n ^2 > nlogn > n > logn > 1 순입니다.



상수시간 시간복잡도 O(1)

상수시간 시간복잡도는 입력크기와 상관없이 일정한 시간복잡도를 가지는 것을 말하며 O(1)의 시간복잡도를 씁니다.

예를 들어 다음과 같은 부분들이 모두 O(1)의 시간복잡도를 가집니다.

입력과 출력
ex) cin, cout, scanf, printf

곱하기

```
a[2] *= 2;
```

이외에도 곱하기, 나누기, 나머지연산, 빼기 등은 O(1)을 가집니다.

간단한 비교 if문

```
if(a[2] == 2)
```

배열의 인덱스 참조

```
int a[3] = {1, 2, 3};
int b = a[2];
```

> 문제로 연습하는 시간복잡도

자 그러면 문제를 통해 배워보도록 하겠습니다.

Q1. 다음 코드의 시간복잡도는?

```
#include<bits/stdc++.h>
using namespace std;
int n;
int main(){
    cin >> n;
    int a = 0;
    for(int i = 0; i < n; i++){
        for(int j = 0; j < i; j++){
            a += i + j;
        }
    }
    cout << a << '\n';
    return 0;
}
```

> *O(n ^ 2)*

Q2. 다음 코드의 시간복잡도는?

```
#include<bits/stdc++.h>
using namespace std;
int N, M;
void solve(int N, int M){
    int a = 1;
    for (int i = 0; i < N; i++) {
        a *= i;
    }
    for (int j = 0; j < M; j++) {
        a *= j;
    }
    cout << a << "\n";
}
int main(){
    cin >> N >> M;
    solve(N, M);
    return 0;
}
```

> $O(N + M)$

Q3. 다음 코드의 시간복잡도는?

```cpp
#include<bits/stdc++.h>
using namespace std;
int n, a[1004], cnt;
int go(int l, int r){
    if(l == r) return a[l];
    int mid = (l + r) / 2;
    int sum = go(l, mid) + go(mid + 1, r);
    return sum;
}
int main(){
    cin >> n;
    for(int i = 1; i <= n; i++){
        a[i - 1] = i;
    }
    int sum = go(0, n - 1);
    cout << sum << '\n';
}
/*
입력
10

출력
45
*/
```

> $O(n)$

Q4. 다음 코드의 시간복잡도는?
로그(log)는 **지수 함수의 역함수**이다. 어떤 수를 나타내기 위해 고정된 밑을 몇 번 곱하여야 하는지를 나타냅니다.

```
#include<bits/stdc++.h>
using namespace std;
int N;
void solve(int N){
    int a = 0, i = N;
    while (i > 0) {
        a += i;
        i /= 2;
    }
    cout << a << '\n';
}
int main(){
    cin >> N;
    solve(N);
    return 0;
}
```

> *O(logN)*

Q5. 다음 코드의 시간복잡도는?

```
#include<bits/stdc++.h>
using namespace std;
int N, cnt;
void solve(int N){
    cnt++;
    cout << cnt << '\n';
    if(N == 0) return;
    for(int i = 0; i < 3; i++){
        solve(N - 1);
    }
    return;
}
int main(){
    cin >> N;
    solve(N);
    return 0;
}
```

등비수열의 합.

① $\frac{a(r^n-1)}{r-1}$ $(a:\text{초항} \ / \ r:\text{공비} \ / \ n:\text{더하는것의 개수})$

② $\frac{a}{1-r}$ $(a:\text{초항} \ / \ r:\text{공비})$

> *3 ^ n*

Q. 재귀함수에서의 메인로직은 무엇인가요?
메인로직은 해당 함수에서 시간복잡도가 어느정도 큰 주요 로직이라고 보시면 됩니다.

```
#include<bits/stdc++.h>
using namespace std;
int N, cnt;
void solve(int N){
    cnt++;
    cout << cnt << '\n';
    if(N == 0) return;
    for(int i = 0; i < 3; i++){
        solve(N - 1);
    }
    return;
}
int main(){
    cin >> N;
    solve(N);
    return 0;
}
```

이 solve라는 함수에서 주요한 로직은

```
    for(int i = 0; i < 3; i++){
        solve(N - 1);
    }
```

이부분입니다.

정확히는... 두가지의 로직이 합쳐져 있죠?

로직1 : 주어진 N에 대해 solve(N - 1)을 3번 재귀 호출
로직2 : 각 호출마다 cnt를 증가시키고 출력합니다.
여기서 메인로직이란 반복적으로(재귀적으로) 호출하는 부분이 시간복잡도가 크기 때문에 로직1이 됩니다.

다만 재귀함수에서 시간복잡도를 산정할 때는 여러가지 로직 중에서 재귀적으로 호출되는 함수부분과 다른 부분을 분리시켜서 봐야 하고 재귀적으로 호출되지 않는 부분중에 가장 시간복잡도가 큰 부분이 중요한 메인로직이 됩니다.

그리고 재귀함수의 시간복잡도는 해당 메인 로직 * 반복횟수로 결정됩니다.

```
void solve(int N){
    cnt++;
    cout << cnt << '\n';
    if(N == 0) return;
    for(int i = 0; i < 3; i++){
        solve(N - 1);
    }
    return;
}
```

이코드에서 반복되는 for문을 제외하고 가장 큰 부분은 cout 부분 출력 = O(1)이기 때문에(로직2가 메인로직이 된다.)

```
cout << cnt << '\n';
```

총 시간복잡도는 O(1) * 반복되는 횟수 = 3^n이 되어 3^n이 됩니다.
자 그러면... 이코드의 시간복잡도는 얼마일까요?

```
void solve(int N){
    cnt++;
    for(int i = 0; i < N; i++){
        cout << cnt << '\n';
    }
    if(N == 0) return;
    for(int i = 0; i < 3; i++){
        solve(N - 1);
    }
    return;
}
```

O(N) * 3^n이 되겠죠? 반복적으로 호출되는 부분 제외 -> 가장 큰부분 = for문 x 반복횟수가 되기 때문입니다.

> 시간복잡도의 존재이유 : 효율적인 코드의 척도

자 그러면 이 시간복잡도는 왜 필요할까요? 바로 효율적인 코드로 개선하는데 쓰이는 척도가 됩니다.

시간복잡도 $O(n^2)$을 $O(n)$으로 줄였다

라고 해봅시다. 이는 어떤 버튼을 누르고 화면이 나타나는데 9초이며 이 시간복잡도가 n^2인데 n으로 개선시킨다면 3초가 될 수도 있는 것이죠.

자 그러면 알고리즘 문제를 풀 때를 보죠. n의 크기가 10만이라고 해봅시다. 그러면 n ^ 2의 시간복잡도면 100억입니다. 문제에 따라 다르긴 하지만 보통 100억이면 잘 안됩니다...
그럴 때는 "아 조금 더 작은 시간복잡도의 알고리즘을 써야 하겠구나. 로직을 개선해볼까? O(nlogn) 이하의 알고리즘을 써볼까?" 라고 하는 생각의 기준점이 되는 것입니다.

이 부분은 문제를 풀 때 매우 매우 중요합니다.
항상 문제를 보고 문제로 부터 어떻게 풀어야 하는지 로직을 생각한 후 그 로직의 시간복잡도를 얼추 계산하고 그리고 나서 시간초과가 뜰 것 같은면 다른 낮은 시간복잡도를 가진 로직을 빠르게 생각하는 것이 코딩테스트 합격, 알고리즘 대회에서 높은 순위를 가질 수 있는 것의 기초가 됩니다.

>

자료구조의 시간복잡도

지금까지 배운 시간복잡도를 기반으로 자료구조를 쓸 때 적용해보겠습니다.
어떠한 로직을 생각하고 그 로직에 알맞는 자료구조를 선택할 때 기준이 되는 것이 자료구조의 시간복잡도입니다.
예를 들어 어떠한 요소들의 모음에서 k번째 요소를 계속해서 "참조"해야 하는 로직이 있습니다.
그렇다면 이에 걸맞는 자료구조는 무엇일까요?
바로 배열입니다.

배열(Array)
- 참조 : O(1)
- 탐색 : O(n)

연결리스트(doubly linked list)
- 참조 : O(n)
- 탐색 : O(n)
- 삽입 / 삭제 : O(1)

만약 연결리스트를 쓰게 된다면 참조에 매번 O(k)의 시간복잡도가 드는 반면 배열은 O(1)의 시간복잡도가 걸리기 때문입니다.

이러한 생각을 자연스레 하려면 자주 쓰는 자료구조들의 시간복잡도가 대략적으로 얼마인지를 알고 있어야 합니다.

보통 시간복잡도를 고려할 때 평균 또는 최악의 시간복잡도를 기반으로 생각을 하는데 문제를 풀 때는 최악의 시간복잡도를 기반으로 해야 합니다.
예를 들어 어떠한 코딩테스트나 알고리즘 문제가 있다고 해봅시다. 이 문제에 나의 코드를 제출했는데 "어라? 시간초과"가 떴습니다. 이 상황은 어떠한 상황일까요? 바로 제가 제출한 코드가 문제 내부에 있는 (외부로 주어지는 테스트케이스 말고) 테스트케이스에 나의 코드가 시간초과가 된 상황입니다. 평균적으로 나의 자료구조는 시간복잡도가 O(1)이고 최악은 O(n)일 때 해당 테스트케이스에서는 O(1)이 아니라 O(n)인 상황이 발생해버린 것이죠.

자, 그러면 최악의 시간복잡도를 기반으로 대표적인 자료구조들을 설명해보겠습니다.
자주 일어나는 연산만을 기반으로 설명하며 자주 일어나지 않은 연산은 제외해서 설명합니다.
예를 들어 정적 배열은 삭제 연산이 자주 일어나지 않습니다.
다음과 같은 코드를 구축했을 때 해당 요소를 삭제하는 것이 아니라 삭제했다는 표기를 위해 a[k] = -1; 이런식으로 어떠한 값을 설정하는 것이 일반적입니다.

```
int a[1004];
```

배열(Array)

- 참조 : O(1)
- 탐색 : O(n)

배열(vector)

- 참조 : O(1)
- 탐색 : O(n)
- 맨 끝, 앞에 삽입/삭제 : O(1)
- 중간에 삽입 / 삭제 : O(n)

스택(stack)

- n번째 참조 : O(n)
- 가장 앞부분 참조 : O(1)
- 탐색 : O(n)
- 삽입 / 삭제(n번째 제외) : O(1)

큐(queue)

- n번째 참조 : O(n)
- 가장 앞부분 참조 : O(1)
- 탐색 : O(n)
- 삽입 / 삭제(n번째 제외) : O(1)

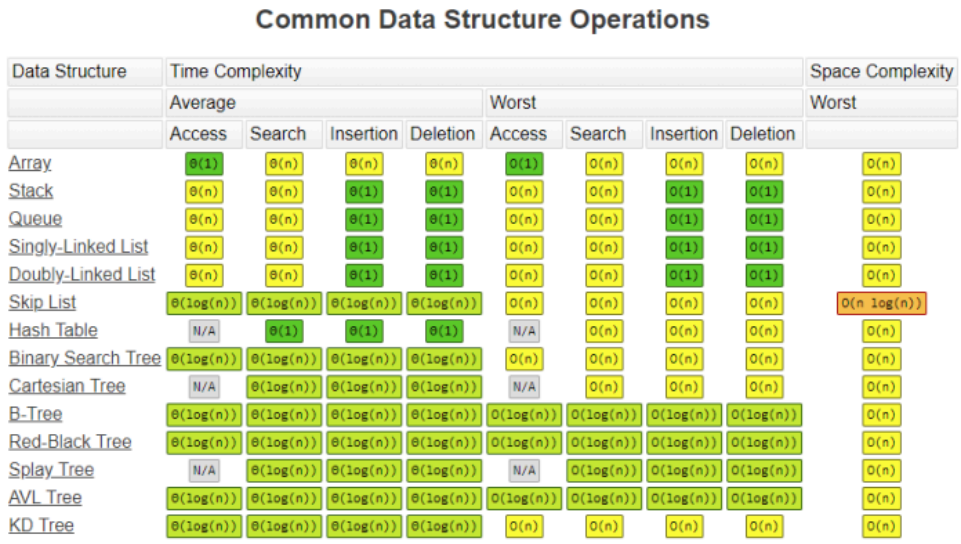
연결리스트(doubly linked list)

- 참조 : O(n)
- 탐색 : O(n)
- 삽입 / 삭제 : O(1)

맵(map)

- 참조 : O(logn)
- 탐색 : O(logn)
- 삽입 / 삭제 : O(logn)

이 외에 자료구조들의 시간복잡도는 다음과 같습니다.
앞서 설명한 것은 외워주시고 아래의 표는 참고삼아 봐주시면 됩니다.



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

> 공간복잡도

공간복잡도는 "입력크기에 대해 어떠한 알고리즘이 실행되는데 필요한 메모리 공간의 양 "을 가리킵니다.
이는 정적변수로 선언된 것 말고도 동적으로 재귀적인 함수로 인해 공간을 계속해서 필요로 할경우도 포함하며 배열이든 맵이든 셋이든 요소들을 담을 공간이면 다 적용됩니다.

다음 코드에서 int형 배열 1000만짜리 1개, 1004개짜리 1개를 선언했죠? 둘 다 포함해서 공간복잡도를 계산합니다.

```
int a[10000000];
void f(){
    int b[1004];
}
int main(){

}
```

자 예를 들어 입력이 N이 들어왔고 거기에 따라 N개 짜리 int형 요소를 담아야 할 공간이 필요하다면 4N바이트의 공간이 필요하겠죠?
이를 빅오표기법으로 나타내면 O(N)이 됩니다. 상수는 버리니까요.

```
int a[N];
```

근데 사실, 이러한 공간복잡도의 개념은
"문제"를 푸는데 잘 사용되지 않아요.

보통 문제를 풀 때 배열의 범위 등을 잡을 때는 2가지 방법을 기반으로 잡습니다.

1. 최대범위
하지만 대부분은 문제의 최대범위를 기반으로 배열을 미리 만들곤 합니다.
다음 문제에서 N의 최대범위는 얼마죠? 1000000입니다. 무려 100만이죠?

입력

첫째 줄에 수의 개수 N(1 ≤ N ≤ 1,000,000)과 M(1 ≤ M ≤ 10,000), K(1 ≤ K ≤ 10,000) 가 주어진다. M은 수의 변경이 일어나는 횟수이고, K는 구간의 합을 구하는 횟수이다. 그리고 둘째 줄부터 N+1번째 줄까지 N개의 수가 주어진다. 그리고 N+2번째 줄부터 N+M+K+1번째 줄까지 세 개의 정수 a, b, c가 주어지는데, a가 1인 경우 b(1 ≤ b ≤ N)번째 수를 c로 바꾸고 a가 2인 경우에는 b(1 ≤ b ≤ N)번째 수부터 c(b ≤ c ≤ N)번째 수까지의 합을 구하여 출력하면 된다.

그러면 보통은 다음과 같이 선언을 하곤 합니다.

```
int a[1000000];
```

2. 메모리제한
다음과 같은 문제에서 메모리 제한은 512MB네요. 아 그러면 계산해볼까요? 512MB는 512,000,000 바이트네요? 그러면
int a[128000000];을 선언할 수 있겠네요?

연구소 성공

시간 제한	메모리 제한	제출	정답	맞힌 사람	정답 비율
2 초	512 MB	69853	40210	22145	55.015%

자, 근데 이 문제는 2차원배열이 필요하니 음 쪼개보죠.
적당히 10000 * 10000으로 해서 집어넣어볼게요.

```cpp
#include<bits/stdc++.h>
using namespace std;
int a[10000][10000], visited[10][10], n, m, ret;
vector<pair<int, int>> virusList, wallList;
const int dy[] = {-1, 0, 1, 0};
const int dx[] = {0, 1, 0, -1};
void dfs(int y, int x){
    for(int i = 0; i < 4; i++){
        int ny = y + dy[i];
        int nx = x + dx[i];
        if(ny < 0 || ny >= n || nx < 0 || nx >= m || visited[ny][nx] || a[ny][nx] == 1)
        visited[ny][nx] = 1;
        dfs(ny, nx);
    }
    return;
}
int solve(){
    fill(&visited[0][0], &visited[0][0] + 10 * 10, 0);
    for(pair<int, int> b : virusList){
        visited[b.first][b.second] = 1;
        dfs(b.first, b.second);
    }

    int cnt = 0;
    for(int i = 0; i < n; i++){
        for(int j = 0; j < m; j++){
            if(a[i][j] == 0 && !visited[i][j])cnt++;
        }
    }
    return cnt;
}
int main(){
    cin >> n >> m;
    for(int i = 0; i <n; i++){
        for(int j = 0; j < m; j++){
            cin >> a[i][j];
            if(a[i][j] == 2) virusList.push_back({i, j});
            if(a[i][j] == 0) wallList.push_back({i, j});
        }
    }
    for(int i = 0; i < wallList.size(); i++){
        for(int j = 0; j < i; j++){
            for(int k = 0; k < j; k++){
                a[wallList[i].first][wallList[i].second] = 1;
                a[wallList[j].first][wallList[j].second] = 1;
                a[wallList[k].first][wallList[k].second] = 1;
                ret = max(ret, solve());
                a[wallList[i].first][wallList[i].second] = 0;
                a[wallList[j].first][wallList[j].second] = 0;
                a[wallList[k].first][wallList[k].second] = 0;
            }
        }
    }
    cout << ret << "\n";
    return 0;
}
```

제출 번호	아이디	문제	결과	메모리	시간	언어	코드 길이	제출한 시간
51220076	zagabi	14502	맞았습니다!!	392648 KB	40 ms	C++17 / 수정	1708 B	38초 전

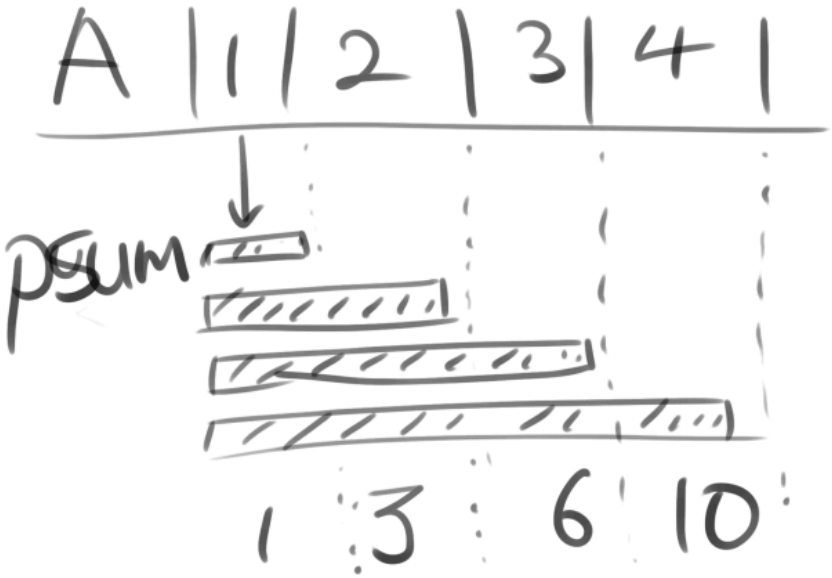
자, 이렇게 맞는다는 것을 볼 수 있습니다.
그렇다면 문제를 풀 때마다 이렇게 해야할까요?
메모리제한을 매번 보기에는 좀 힘들어요.. 그래서 이부분 같은 경우는 "**1000만까지는 어느정도는 된다**" 라고 잡고 들어가시는게 좋아요.
앞의 코드는 무려 1억짜리 배열을 만들었는데도 불구하고 잘 진행되었습니다.

문제마다 다르긴 하지만... 보통은 1000만일 때는 잘 안되는 경우가 많습니다.
유연하게 접근하되 일단은 이러한 사실을 기반으로 문제를 풀면 빠르게 문제를 풀 수 있을겁니다.

나중에 배우겠지만 어떠한 문제를 보았을 때 유동적인 배열이고 O(logn)시간만에 어떠한 요소를 탐색하는 로직이 들어간다면 이분탐색과 펜윅트리가 생각이 나야 합니다. 이 때 이 2가지 알고리즘의 차이는 펜윅트리는 새로이 배열을 만들어야 하지만 이분탐색은 새로이 배열을 만들지 않아도 쓸 수 있는 알고리즘입니다.
이 때 배열의 크기가 1000만이 필요하다면 펜윅트리로 했을 때 안되겠네? 이분탐색으로 해볼까? 하면서 들어가야 하는 것입니다.
유연하게. 그러나 빠르게
가 중요합니다.

> 누적합

누적합이란 요소들의 누적된 합의 의미로 어떠한 배열을 기반으로 앞에서 부터 요소들의 누적된 합을 저장해 새로이 배열을 만들어서 이를 활용하는 것을 말합니다.
이는 앞에서부터 더하는 prefix sum과 뒤에서 부터 더하는 suffix sum 이 있지만 코딩테스트에는 prefix sum만 나오니 prefix sum만을 배우면 됩니다.



위그림처럼 누적합이란 누적해서 더해진 값들의 배열을 의미합니다. 1, 2, 3, 4라는 배열의 누적합은 1, 3, 6, 10이란 배열이 됩니다.
이렇게 만들어놓으면 뭐가 좋을까요? 바로 구간쿼리에 대응하기가 쉽습니다. 예를 들어 0번째 요소부터 3번째요소까지 다 더해라고 했을 때 누적합배열 3번째요소를 이용하면 되니까요.

문제를 풀 때 "구간"에 대한 많은 "쿼리"가 나올 때 생각해야 될 것은 트리 또는 누적합입니다. 여기서 트리는 세그먼트, 펜윅트리를 뜻합니다. 이 때 그 구간 안에 있는 요소들이 변하지 않는 정적 요소라면 누적합을 쓰면 됩니다.

예시문제

승철이는 뇌를 잃어버렸다. 학교에 갔더니 선생님이 자연수로 이루어진 N개의 카드를 주며 M개의 질문을 던진다. 그 질문은 나열한 카드 중 A번째부터 B번째까지의 합을 구하는 것이다. 뇌를 잃어버렸기 때문에 승철이는 이 문제를 풀 수 없다. 문제를 풀 수 있는 프로그램을 작성해보자.

입력

수의 개수 N, 합을 구해야 하는 횟수 M, 그 이후 N개의 수가 주어진다. 수는 100 이하의 자연수. 그 이후 M개의 줄에는 합을 구해야 하는 구간 A, B가 주어진다.

출력

M개의 줄에 A부터 B까지의 합을 구하라.

범위

1 <= N <= 100,000

1 <= M <= 100,000

1 <= A <= B <= N

예제입력

```
8 3
1 2 3 4 5 6 7 8
1 4
1 5
3 5
```

예제출력

```
10
15
12
```

이문제를 무식하게 구현했을 때 어떻게 될까요?

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
int a[100004], b, c, psum[100004], n ,m;
int main(){
    cin >> n >> m;
    for(int i = 1; i <= n; i++){
        cin >> a[i];
    }
    for(int i = 0 ; i < m; i++){
        cin >> b >> c;
        int sum = 0;
        for(int j = b; j <= c; j++) sum += a[j];
        cout << sum << '\n';
    }
    return 0;
}
```

이런꼴이 되겠죠? 위의 코드는 시간복잡도는 O(10만 * 10만) 이며 A와 B가 주어졌을 때 한 연산당 시간복잡도는 O(10만) 입니다. 즉, 단순하게 접근하면 시간초과가 나는 것이죠.

이럴 때 "누적합"을 써야 합니다. 쿼리가 주어졌고 배열은 변함없는 정적요소이기 때문이죠.

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
int a[100004], b, c, psum[100004], n ,m;
int main(){
    ios_base::sync_with_stdio(false); cin.tie(NULL); cout.tie(NULL);
    cin >> n >> m;
    for(int i = 1; i <= n; i++){
        cin >> a[i];
        psum[i] = psum[i - 1] + a[i];
    }
    for(int i = 0 ; i < m; i++){
        cin >> b >> c;
        cout << psum[c] - psum[b - 1] << "\n";
    }
    return 0;
}
```

코드설명

```
psum[i] = psum[i - 1] + a[i];
```

누적합을 만들 때는 반드시 "1번째 요소부터" 만드는 것이 좋습니다. 왜냐면 i = i - 1 이부분이 있기 때문에 0 부터 시작한다면 -1이 되기 때문입니다.

```
psum[c] - psum[b - 1]
```

그 다음 이렇게 빼기만 하면 A부터 B까지 구해라 라는 것을 쉽게 O(1)만에 할 수 있습니다.

구현

사실 구현은 아주 쉬운 알고리즘입니다. 말그대로 문제 그대로 구현을 하면 됩니다. 예를 들어서 배열을 회전하라 스택에 넣어라 이러면 말그대로 rotate 함수를 사용하고 stack.push 등을 하면 되는 것입니다.

다음과 같이 문자열을 선언했다고 하고 아래의 문제를 풀어보겠습니다.

```
string dopa = "umzunsik";
```

Q1. 앞에서부터 3개의 문자열을 출력하라
Q2. 해당 문자열을 거꾸로 해서 출력하라.

Q3. 해당 문자열 끝에 "umzunsik"이란 문자열을 추가하라.

라고 하면 다음과 같은 코드를 구축해야 합니다.

```
#include<bits/stdc++.h>
using namespace std;
string dopa = "abcde";
int main(){
	cout << dopa << "\n";
	//문자열에서 부분배열(이 부분만을 끄집어낼 수 있겠죠?)
	cout << dopa.substr(0, 3) << "\n";
	// 반대로
	reverse(dopa.begin(), dopa.end());
	cout << dopa << "\n";
	// 추가한다.
	dopa += " umzunsik";
	cout << dopa << '\n';
	return 0;
}
/*
abcde

abc
edcba
edcba umzunsik
*/
```

 # 함수설명
reverse : 원래의 문자열을 바꿔버립니다. begin과 end를 통해 전체를 바꿔도 되고 dopa.begin(), dopa.begin() + 3 이런식으로 부분만 바꿔버릴 수도 있습니다.
substr : 시작지점으로부터 몇개의 문자열을 뽑아냅니다.

문제를 푸는 방법

1주차에서 중요한 점은 문제 푸는 방법을 아는 것입니다.
우리는 어떻게 문제를 풀어야 할까요?
1. 문제를 봅니다.
2. 문제를 해석합니다.
3. 코드를 작성합니다.

크게 3가지의 과정을 통해서 문제를 풀게 됩니다.
여기서 중요한 점은
2번 문제를 해석하는 것입니다. 문제를 처음에 봤을 때 이러한 일련의 생각들이 일어나야 합니다.
* 먼저 코드 작성하지 말고 해석에 좀 더 시간을 써야 되요!!

입력

첫째 줄에 지도의 세로 크기 N과 가로 크기 M이 주어진다. (3 ≤ N, M ≤ 8)

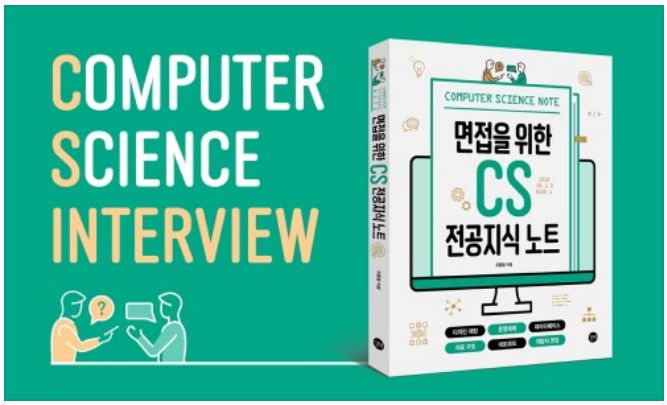
둘째 줄부터 N개의 줄에 지도의 모양이 주어진다. 0은 빈 칸, 1은 벽, 2는 바이러스가 있는 위치이다. 2의 개수는 2보다 크거나 같고, 10보다 작거나 같은 자연수이다.

빈 칸의 개수는 3개 이상이다.

1. 최대, 최소 범위를 파악합니다.
2. 단순 구현이라면 구현하자.
3. 무식하게 풀 수 있다면 무식하게 풀자.
4. 아니라면 다른 알고리즘을 생각하자.
5. 제출하기전, 반례를 항상 생각하자.

이런 순서로 들어가야 하는 것이죠.

https://inf.run/KPcs



CS지식의 정석 | CS면접 디자인패턴 네트워크 운영체제 데이터…

국내 1위 "면접을 위한 CS전공지식노트" 저자의 디자인패턴, 네트워크, …

inf.run

https://inf.run/YfJa

10주완성 C++ 코딩테스트 | 알고리즘 IT취업 - 인프런 | 강의

네이버, 카카오, 삼성의 코딩테스트를 합격시켰다! 10주 완성 C++ 코딩…

inf.run