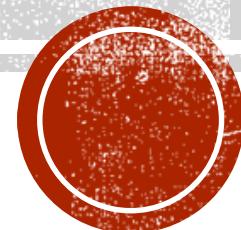


C++11 AND ETC.

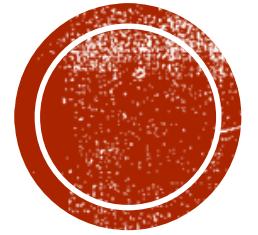
Seo youngsun(nein) Soongsil Univ.



CONTENT

- C++11
 - Auto
 - Initializer Lists
 - Range-based for
 - Tuple
 - Lambda
 - STL
- ETC





c++11

C++11

- **Problem Solving**을 하는데 C++11로 해야하는 이유
 - 기존 C++으로는 표현하기 어려웠던 코드들을 비교적 간단하게 짤 수 있다.(=디버깅이 쉬워진다.)
 - 성능이 조금이지만 향상된다고 한다.



AUTO

- Auto는 컴파일 타임에 변수 타입을 자동으로 추론하여 선언한다.

```
vector<int> a = {1,2,3,4,5};  
for (vector<int>::iterator it = a.begin(); it != a.end(); it++) {  
    cout << *it << '\n';  
}
```

```
vector<int> a = {1,2,3,4,5};  
for (auto it = a.begin(); it != a.end(); it++) {  
    cout << *it << '\n';  
}
```



INITIALIZER LISTS

- STL 컨테이너나 구조체, 클래스의 초기화나 `min`, `max` 등 다양하게 사용된다.

```
vector<int> a = {1,2,3,4,5};  
pair<string, int> p = {"Year", 2015};  
map<string, int> age = { {"Sunyoung", 26}, {"Sangkeun", 26} };  
vector<vector<int>> matrix = {{1, 2}, {3, 4}};  
vector<pair<int,int>> l = {{1, 2}, {3, 4}, {5, 6}};
```

```
vector<pair<int,int>> a;  
  
a.push_back(make_pair(1,2));  
a.push_back({1,2});
```



INITIALIZER LISTS

```
struct Point {  
    double x, y;  
    Point(double x, double y) : x(x), y(y) {}  
};  
vector<Point> a;  
  
a.push_back(Point(1,2));  
a.push_back({1,2});
```

```
min(1,2);  
min({1, 2, 3});  
min({1, 2, 3, 4});
```



RANGE-BASED FOR

- STL의 컨테이너나 배열 등등에서 모든 원소를 순회하는 **for**문

```
vector<int> a = {1,2,3,4,5};
for (vector<int>::iterator it = a.begin(); it != a.end(); it++) {
    cout << *it << '\n';
}
for (int i=0; i<a.size(); i++) {
    cout << a[i] << '\n';
}
for (auto i : a) {
    printf("%d\n",i);
}

set<string> s = {"Choi", "Kim", "Park", "Lee"};
for (set<string>::iterator it = s.begin(); it != s.end(); it++) {
    const string &name = *it;
    cout << name << '\n';
}
for (const string &name : s) {
    cout << name << '\n';
}
for (auto &name: s) {
    cout << name << '\n';
}

map<string, int> d = {"brown",1}, {"red",2}, {"blue",6};
for (auto &p : d) {
    auto &key = p.first;
    int val = p.second;
    cout << key << ' ' << val << '\n';
}
```



RANGE-BASED FOR

- **String**의 경우는 다음 아래를 주의해야 한다.

```
for (char c : "RGB") {  
    // 4번  
}  
for (char c : string("RGB")) {  
    // 3번  
}
```



RANGE-BASED FOR

- **Initializer Lists**도 사용 가능하다.

for문으로 돌리기도 뭐하고, copy&paste 신공으로 여러줄 쓰기도
뭐한데 특정 부분 또는 argument만 달라지는 경우에, range-based for
문과 함께 유용함.

```
for(const auto x : {2, 3, 5, 7}) {  
    foo(x);  
}
```



RANGE-BASED FOR

- Range-based for문에서 선언하는 변수가 레퍼런스 타입이 아니면, 값 복사가 발생하므로 주의

```
vector<LargeStruct> data;

// Wrong! (copy by value)
for(LargeStruct a : data) { ... }
for(auto a : data) { ... }

// Preferred way (avoid copying)
for(auto& a : data) { ... }
for(const auto& a : data) { ... }
```



TUPLE

- Tuple은 pair의 확장형으로 1~10개의 원소를 담을 수 있다.

```
pair<int,pair<int,int>> p = make_pair(1,make_pair(2,3));
cout << p.first << ' ' << p.second.first << ' ' << p.second.second << '\n';
```



```
tuple<int, int, int> t(1,2,3);
tuple<int, int, int> t2 = make_tuple(1,2,3);
```



TUPLE

- 원소는 `get`을 통하여 접근할 수 있다.

```
tuple<int, int, int> t(1,2,3);
int t1 = get<0>(t);
int t2 = get<1>(t);
int t3 = get<2>(t);
cout << t1 << ' ' << t2 << ' ' << t3 << '\n';
```



TUPLE

- 어렵게도 아래와 같은 사용이 불가능하다.

```
tuple<int, int, int> t(1,2,3);
for (int i=0; i<3; i++) {
    int t = get<i>(t);
}
```



TIE

- Tie는 C++에서는 지원을 안하는 multi return이나 tuple, pair의 원소를 한번에 복사할 때 사용된다.

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

tuple<int,string> multi_return(){
    return make_tuple(1,"abc");
}

int main(){
    int a;
    string b;
    tie(a,b)=multi_return();
    cout<<a<<b;
}
```

```
tuple<int, int, int> t(1,2,3);
int t1,t2,t3;
tie(t1,t2,t3) = t;
```



TIE

- Minmax 함수에서 return 되는 값이 pair이기 때문에 다음과 같아도 쓸 수 있다.

```
int t1, t2;
tie(t1,t2) = minmax({1,2,3});
cout << t1 << ' ' << t2 << '\n';
```



LAMBDA

- Lambda는 익명 함수로, 기본적인 형태는 아래와 같다.

```
// [] : captures  
// () : parameters  
// {} : body  
auto f = [](){};
```



LAMBDA

- C++에 수많은 STL과 함수들이 Lambda함수를 지원한다.

```
#include<stdio.h>
#include<algorithm>
#include<vector>
using namespace std;
int main(){
    vector<int> arr={1,2,3,4,5};
    sort(arr.begin(),arr.end(),[](const int a,const int b){
        return a>b;
    });
}
```



LAMBDA

- 만약 바깥의 원소를 참조하여 정렬을 할 경우, 다음과 같이 할 수 있다.

```
#include<stdio.h>
#include<algorithm>
#include<vector>
using namespace std;

struct point {
    int x;
    int y;
};

// [=]은 call by value
// [&]은 call by reference
int main() {
    vector<point> arr = { { 1,2 },{ 3,4 },{ 5,6 } };
    sort(arr.begin(), arr.end(), [=](const point a, const point b) {
        return atan2(a.y - arr[0].y, a.x - arr[0].x) < atan2(b.y - arr[0].y, b.x - arr[0].x);
    });
}
```



LAMBDA

- Lambda함수의 자료형을 정의하고 싶으면, 아래와 같이 정의할 수 있다. **Funtion**은 **functional**헤더 파일 안에 정의되어 있다.

```
#include <functional>
using namespace std;
int main() {
    function<int(int,int)> sum = [](int a, int b) { return a+b; };
    printf("%d\n",sum(1,2));
}
```

LAMBDA

- Lambda함수의 리턴타입을 지정하려 한다면 아래와 같이 할 수 있다.

```
#include <functional>
using namespace std;
int main() {
    function<int(int, int)> sum = [] (int a, int b) -> int { return a + b; };
    printf("%d\n", sum(1, 2));
}
```



LAMBDA - EXAMPLE

■ 1) sort함수

```
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
using namespace std;
struct Person {
    string name;
    int kor, eng, math;
};
int main() {
    int n;
    cin >> n;

    vector<Person> a(n);
    for (int i=0; i<n; i++) {
        cin >> a[i].name >> a[i].kor >> a[i].eng >> a[i].math;
    }

    sort(a.begin(), a.end(), [](const Person &u, const Person &v) {
        return make_tuple(-u.kor, u.eng, -u.math, u.name) < make_tuple(-v.kor, v.eng, -v.math, v.name);
    });

    for (Person &p : a) {
        cout << p.name << '\n';
    }

    return 0;
}
```



LAMBDA - EXAMPLE

- 2) 정점 분류

```
int solve(vector<int> &a, vector<int> &b) {
    int n = (int) a.size();
    int S = 0, T = V - 1;
    int V = 2 * n + 2;

    auto L = [&](int x) { return x + 1; };
    auto R = [&](int x) { return x + 1 + n; };

    NetworkFlow f(V);
    for(int i=0; i<n; ++i) {
        f.addEdge(S, L(i), a[i]);
        f.addEdge(L(i), R(i), a[i]);
        f.addEdge(R(i), T, b[i]);
    }
    /* ... */
}
```

LAMBDA - EXAMPLE

- 3) DFS

```
bool isGraphConnected(vector<vector<int>> &G) {
    vector<bool> visited(G.size(), false);
    std::function<void (int)> dfs = [&](int u) {
        if (visited[u]) return;
        for (int v : G[u]) dfs(v);
    };
    dfs(0);
    for(size_t i = 0; i < G.size(); ++ i)
        if (!visited[i]) return false;
    return true;
}
```



LAMBDA - EXAMPLE

- 4) all_of

```
vector<int> primes={2,3,5,7,11};  
bool allEven=all_of(primes.begin(),primes.end(),[](int i){  
    return i%2==0;  
})
```



EMPLACE

- **vector**나 **queue**, **stack** 등에서 원소를 넣을 때 **emplace**를 사용하면 **constructor**에 맞춰서 원소가 들어간다.

```
vector<pair<int,int>> a;
a.push_back(make_pair(1,2));
a.emplace_back(1,2);

queue<tuple<int,int,int>> q;
q.push(make_tuple(1,2,3));
q.emplace(3,4,5);
```

```
#include<vector>

struct node {
    int a;
    int b;
    node(int a, int b) :a(a), b(b) {}
};

std::vector<node> vec;
int main() {
    vec.emplace_back(1, 2);
}
```



STL

- C++11이 되면서 추가된 STL이 몇가지 있다.
 - `unordered_map`
 - `unordered_multimap`
 - `unordered_set`
 - `unordered_multiset`
 - `array`



STL – UNORDERED MAP

- Hash를 이용하여 구현된 map으로 이론적으로는 O(1)의 시간복잡도를 가진 dictionary이다.
- Hash를 이용하기 때문에 원소가 정렬이 되어있지는 않다.
- 기존의 std::map과 매우 유사하므로 거의 그대로 사용하면 된다.
- 다른 STL 또한 큰 차이가 없으며, multi가 붙은 것은 원소를 중복해서 가질 수 있다.



STL – ARRAY

- **Vector**와 달리 크기가 고정된 정적 배열이다.
- 하지만 **int[]**와 같은 배열과 달리 **STL** 알고리즘 함수들에 적용이 가능하며, 그 외에도 다양한 것을 지원한다.
- 그러나 메모리를 여유 있게 잡아 사용하는 대회에서는 크게 활용성은 없는 듯 하다;;



REGEX

- C++11 부터 정규표현식을 지원한다.
- 사용하기에 따라서 꽤 쓸 만하겠지만, 일단 패스



BITSET

- C++11에 추가된 것은 아니고, 기존에 있는 클래스이지만 간단하게 나마 다뤄볼까 한다.
- Bit의 경우 Bitmask를 통한 동적계획법 해결을 하는 경우가 있기 때문에 PS에서 종종 사용된다.
- 대표적인 Bit 연산을 외워서 사용하여도 괜찮지만, 그것이 싫다면 Bitset을 찾아서 사용해보는 것도 한 방법이 될 것이다.



BITSET

- 아래와 같이 사용이 가능하며, []연산자와 기존의 bit 연산자를 지원한다.

```
// constructing bitsets
#include <iostream>          // std::cout
#include <string>            // std::string
#include <bitset>             // std::bitset

int main ()
{
    std::bitset<16> foo;
    std::bitset<16> bar (0xfa2);
    std::bitset<16> baz (std::string("0101111001"));

    std::cout << "foo: " << foo << '\n';
    std::cout << "bar: " << bar << '\n';
    std::cout << "baz: " << baz << '\n';

    return 0;
}
```

```
// bitset operators
#include <iostream>          // std::cout
#include <string>            // std::string
#include <bitset>             // std::bitset

int main ()
{
    std::bitset<4> foo (std::string("1001"));
    std::bitset<4> bar (std::string("0011"));

    std::cout << (foo^=bar) << '\n';      // 1010 (XOR,assign)
    std::cout << (foo&=bar) << '\n';      // 0010 (AND,assign)
    std::cout << (foo|=bar) << '\n';      // 0011 (OR,assign)

    std::cout << (foo<<2) << '\n';      // 1100 (SHL,assign)
    std::cout << (foo>>1) << '\n';      // 0110 (SHR,assign)

    std::cout << (~bar) << '\n';        // 1100 (NOT)
    std::cout << (bar<<1) << '\n';      // 0110 (SHL)
    std::cout << (bar>>1) << '\n';      // 0001 (SHR)

    std::cout << (foo==bar) << '\n';    // false (0110==0011)
    std::cout << (foo!=bar) << '\n';    // true  (0110!=0011)

    std::cout << (foo&bar) << '\n';    // 0010
    std::cout << (foo|bar) << '\n';    // 0111
    std::cout << (foo^bar) << '\n';    // 0101

    return 0;
}
```

```
// bitset::operator[]
#include <iostream>          // std::cout
#include <bitset>             // std::bitset

int main ()
{
    std::bitset<4> foo;

    foo[1]=1;                  // 0010
    foo[2]=foo[1];              // 0110

    std::cout << "foo: " << foo << '\n';

    return 0;
}
```



BITSET

- 다음과 같은 것도 지원한다.

```
1 // bitset::to_ulong
2 #include <iostream>      // std::cout
3 #include <bitset>        // std::bitset
4
5 int main ()
6 {
7     std::bitset<4> foo;    // foo: 0000
8     foo.set();              // foo: 1111
9
10    std::cout << foo << " as an integer is: " << foo.to_ulong() << '\n';
11
12    return 0;
13}
```

Output:

```
1111 as an integer is: 15
```

```
1 // bitset::count
2 #include <iostream>      // std::cout
3 #include <string>        // std::string
4 #include <bitset>        // std::bitset
5
6 int main ()
7 {
8     std::bitset<8> foo (std::string("10110011"));
9
10    std::cout << foo << " has ";
11    std::cout << foo.count() << " ones and ";
12    std::cout << (foo.size()-foo.count()) << " zeros.\n";
13
14    return 0;
15}
```

Output:

```
10110011 has 5 ones and 3 zeros.
```

```
1 // bitset::all
2 #include <iostream>      // std::cin, std::cout, std::boolalpha
3 #include <bitset>        // std::bitset
4
5 int main ()
6 {
7     std::bitset<8> foo;
8
9     std::cout << "Please, enter an 8-bit binary number: ";
10    std::cin >> foo;
11
12    std::cout << std::boolalpha;
13    std::cout << "all: " << foo.all() << '\n';
14    std::cout << "any: " << foo.any() << '\n';
15    std::cout << "none: " << foo.none() << '\n';
16
17    return 0;
18}
```

Possible output:

```
Please, enter an 8-bit binary number: 11111111
all: true
any: true
none: false
```



IN-CLASS MEMBER INITIALIZERS

- 많이들 당연시 된다고 착각하는 거지만 C++11부터 되는 것이다.
- **class**나 **struct**에서 땡버 변수를 정의할 때 초기화 할 수 있다.

```
class AwesomeLibrary {  
    int sum = 0;    // C++03 : Error, C++11 : OK  
    int n;  
};
```



USING

- `using`으로 C++11으로 오면서 `typedef`과 거의 같은 효과를 가지게 되었다.

```
using matrix = vector<vector<long>>;
```



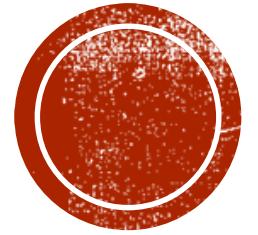
RANDOM

더 좋은 랜덤 성능! (`rand()` 는 uniformity가 떨어지고, `RAND_MAX`가 32767인 머신도 있다)

```
std::mt19937 eng; // Mersenne Twister
std::uniform_int_distribution<int> U(-100, 100);
for (int i = 0; i < n; ++i)
    cout << U(eng) << std;
```

~~마라톤 매치 같은거 할 거 아니면 쓸모 없다~~





ETC



CODING STYLE

- 문제를 풀 때 각 상황별로 해결해야하는 사소한 문제들이 많다.
- 입력부터 시작하여 데이터는 어떻게 저장해야 하는가 등 아주 여러가지 상황에 직면한다.
- 이러한 문제들을 해결할 때 각자 매우 다양하고 참신한 방법들로 해결한다.
- 이러한 문제들을 각자 해결하고 공유해보는 시간을 가지자.



입력

1. 붙어있는 숫자 떨어트려 받기
2. EOF
3. **char** 입력
4. 문자열 파싱해서 받기



붙어있는 숫자

- <http://codeforces.com/contest/431/problem/A>

input

```
1 2 3 4  
123214
```



붙어있는 숫자

```
int arr[5];
scanf("%d%d%d%d", &arr[1], &arr[2], &arr[3], &arr[4]);
int sum = 0;
int a;
while (scanf("%1d", &a) != EOF) {
    sum += arr[a];
}
printf("%d", sum);
```



CHAR 입력

- <https://www.acmicpc.net/problem/13993>

```
5 4
-
+ 4 1
-
+ 7 2
0 3 1 2
```



CHAR 입력

```
int n, q;
scanf("%d%d", &n, &q);
int before = 0;
lint cnt = 0;
vector<segment> S;
for (int i = 0; i < n; i++) {
    char ch;
    int t, k;
    scanf("%*[\\n\\t ]%c%d%d", &ch, &t, &k);
    S.push_back({ before,t,cnt });
    if (ch == '-') {
        cnt -= k;
    }
    else {
        cnt += k;
    }
    before = t;
}
```



입력 문자열 파싱

- <http://codeforces.com/problemset/gymProblem/100513/C>

input

```
4
0 2
enabled=true
foreground=white
1 2
enabled=false
fontSize=12
2 3
enabled=true
foreground=black
caption=OK
2 1
fontSize=10
5
3 enabled
4 enabled
3 fontSize
4 foreground
2 caption
```



입력 문자열 파싱

```
int n;
scanf("%d", &n);
vector<vector<pii>> vec;
for (int i = 1; i <= n; i++) {
    int k;
    scanf("%d%d\n", &parent[i], &k);
    vec.push_back(vector<pii>());
    for (int i = 0; i < k; i++) {
        scanf("%[^=]=%s\n", A, B);
        vec.back().push_back({num(A, key), num(B, value)}));
    }
}
```



테스트 케이스

- 많은 문제들이 한번 실행에 여러 테스트 케이스를 통과하여야 한다.
- 문제를 많이 풀다보면 이 부분을 코딩하기가 점점 귀찮아져 코드가 짧아진다.



테스트 케이스

```
int main(){
    int tc;
    scanf("%d",&tc);
    while(tc--){
        //solve problem
    }
}
```



테스트 케이스 종료조건

- 테스트 케이스의 수를 주는 경우도 있지만, 처음 입력이 전부 0이거나 아예 아무 입력도 안주어지는 경우 등 다양한 경우가 있다.



테스트 케이스 종료조건

```
// 0 0 일 때 종료  
int n,m;  
scanf("%d%d",&n,&m);  
if(n+m==0)break;  
  
// EOF일 때 종료  
if(scanf("%d%d",&n,&m)==EOF)break;
```



초기화

- **for loop**를 돌려 초기화를 해주는 경우도 있지만, **memset**함수를 이용하여 전체를 초기화를 하는 경우가 많다.
- 간혹 모든 테스트 케이스에 대해서 전체를 초기화하여 시간초과가 나는 경우가 있지만, 드문 경우라 많이들 사용한다.



MEMSET

- `memset`(초기화 시작할 주소, 초기화 할 값(1바이트), 초기화할 바이트 수)
- `int dy[50][50];`
- `memset(dy,0x3f,sizeof(dy))`
- `0x7f : 0x7f7f7f7f` `memset`으로 초기화 할 수 있는 최대 `int` 크기
- `0x3f : 0x3f3f3f3f` 오버 플로우 방지를 위한 적당한 `int` 크기
- `0 : 0`으로 초기화
- `0xff :-1`로 초기화, 메모이제이션에서 많이 사용
- `0x80 : 0x80808080` `memset`으로 초기화 할 수 있는 최소 `int` 크기



상수

- 문제를 풀다 보면 다양한 상수 값을 코드에 넣는다.
- `const int MOD=1000000007;`
- `double PI=acos(-1);`
- `const int INF=0x7fffffff;`
- `const int INF=0x7fffffff/2; // overflow 방지`
- `const int MIN=0x80000000;`



중복 제거하기 & 인덱스 변환

- 간혹 입력으로 주어진 수열의 원소들의 중복을 제거하고 몇번째 원소인지 구해야 하는 경우가 있다.
- <https://www.acmicpc.net/problem/10867>
- 좌표 압축



중복 제거하기 & 인덱스 변환

set or map 사용

```
vector<int> arr ={1,3,6,3,2,3,1};  
vector<int> X=arr;  
std::sort(X.begin(),X.end());  
X.erase(std::unique(X.begin(),X.end()),X.end());  
for(int num:arr){  
    int idx=std::lower_bound(X.begin(),X.end(),num)-X.begin();  
}
```



DFS & BFS 구현하기

- 간선의 정보가 주어졌을 때, 모든 정점을 한번씩 방문
- $n \times n$ 격자판에서 최단거리 찾기
- <https://www.acmicpc.net/problem/7576>



USING

- c++ 11 문법의 **typedef**과 같은 역할 외에도 많은 일을 한다.
- 많은 사람들이 **using namespace std;**를 사용하는데 이는 전역 변수를 선언할 때, **std namespace** 안에 있는 무언가와 겹치는 경우 컴파일 에러가 난다.
- **using std::vector;** 와 같은 방법을 사용하면 필요한 것만 가져와 명시적으로 표현하기에 이에 대한 에러를 대비할 수 있다.



STRING을 숫자로 바꾸기

- 문제 중 **string** 을 숫자로 바꿔서 풀어야 하는 경우가 있다.
- <https://www.acmicpc.net/problem/13168>



STRING을 숫자로 바꾸기

```
std::map<std::string,int> map;  
int num(std::string& str){  
    if(!map.count(str))map[str]=map.size();  
    return map[str];  
}
```



ABSOLUTE

- 절대값 함수 **abs**는 특정 라이브러리만 **include**하면 실수 자료형에 대해서만 선언된다.
- 그냥 만들어 쓰는게 마음 편하다.



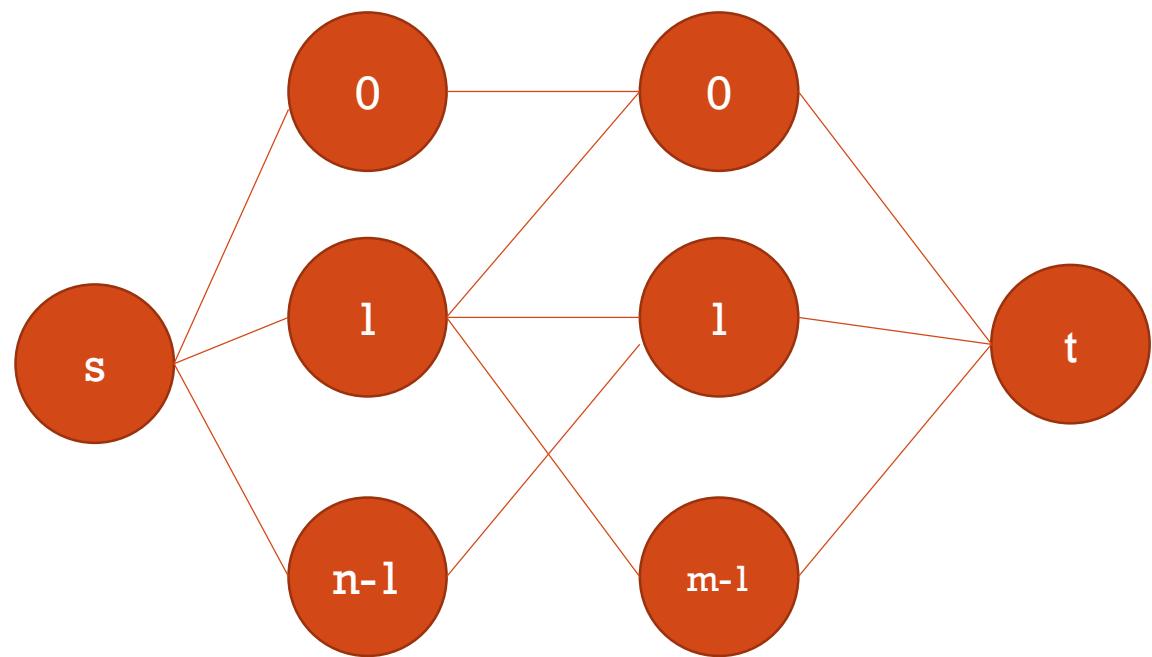
정점 분류

- **maximum flow** 문제들을 풀다보면 정점들에게 번호를 잘 매겨야 하는 경우가 있다.
- 보통 수식을 만들어 직접 때려 넣는데, 중간에 오타라도 나면은 바로 헬게이트가 열린다.



정점 분류

```
int source=0;  
int sink=n+m+1;  
auto L=[&](int i){return i+1;};  
auto R=[&](int i){return i+n+1;};
```



실수 연산에 도움이 되는 연산들

- `fabs(double)` // 실수 절대값
- `sqrt(double)` // 제곱근
- `hypot(double,double)` // $\sqrt{a^2 + b^2}$, 유클리드 거리



자주쓰는 자료구조

- disjoint_set(union&find)
- segement_tree(indexed_tree)



참고용 코드들

- 평소 코딩을 하면서 간단하게 사용하고 있다는 것을 모아봤다.
- 개인적으로 괜찮다고 생각하는 것이니 별로라고 생각하면 안써도 괜찮다.



C STYLE STRING FOR LOOP

- 맨 끝에 **NULL**문자가 있음을 이용해 loop의 끝을 지정

```
char str[100];
for(int i=0;str[i]!='\0';i++){
    //...
}
```



특정 FORMAT 입출력

- 특정 **format**(e.g., time, 공백없는 숫자)의 입력과 출력에 대해서는 **scanf**와 **printf**를 잘 사용하면 편해진다.

```
#include<stdio.h>

int main() {
    int sh, sm, ss;
    int eh, em, es;
    scanf("%d:%d:%d", &sh, &sm, &ss);
    scanf("%d:%d:%d", &eh, &em, &es);
    int start = sh * 60 * 60 + sm * 60 + ss;
    int end = eh * 60 * 60 + em * 60 + es;
    int diff = end - start;
    diff += 24 * 60 * 60;
    diff %= 24 * 60 * 60;
    printf("%02d:%02d:%02d", diff / (60 * 60), diff % (60 * 60) / 60, diff % 60);
}
```

```
#include<stdio.h>

int map[100][100];
int main(){
    int n,m;
    scanf("%d%d",&n,&m);
    for(int i=0;i<n;i++){
        for(int j=0;j<m;j++){
            scanf("%1d",&map[i][j]);
        }
    }
    //....
}
```



DFS – INNER, ETC...

- 내부의 있는지 판단하는 것에 대해서 따로 함수를 만들어 사용하면 코드가 깔끔해진다.
- 또한 4방향으로 이동하는 것에서 다음과 같이 하면 복붙신공을 발휘하지 않아도 코딩을 할 수 있다.

```
#include<stdio.h>
#include<algorithm>
int r, c;

bool inner(int x, int y) {
    return x >= 0 && x < r && y >= 0 && y < c;
}

char map[20][25];
const int X[4] = { +1,-1,+0,-0 };
const int Y[4] = { +0,-0,+1,-1 };
bool chk[20][25];
int ans = 0;

void dfs(int s, int num, int i, int j) {
    if (s & (1 << (map[i][j] - 'A'))) {
        return;
    }
    chk[i][j] = true;
    s |= 1 << (map[i][j] - 'A');
    num++;
    ans = std::max(ans, num);
    for (int k = 0; k < 4; k++) {
        int nx=i+X[k];
        int ny=j+Y[k];
        if(inner(nx,ny) && !chk[nx][ny]) dfs(s, num, nx, ny);
    }
    chk[i][j] = false;
}
```



DFS – INNER, ETC...

- 내부로 나가는 것에 대한 처리를 따로 처리하기 싫다면 배열 크기를 겉으로 한 칸씩 더 잡아 벽으로 초기화 해두는 방법도 있다.

```
#include<stdio.h>
#include<queue>
#include<algorithm>
using std::pair;
const int X[4] = { +1,-1,+0,-0 };
const int Y[4] = { +0,-0,+1,-1 };

int map[1010][1010];

int main() {
    int n,m;
    for (int i = 0; i < 1010; i++) {
        for (int j = 0; j < 1010; j++) {
            map[i][j] = -1;
        }
    }
    scanf("%d%d", &m, &n);

    std::queue<pair<int, int>> que;
    std::queue<int> num;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            scanf("%d", &map[i][j]);
            if (map[i][j] == 1) {
                que.push({ i,j });
                num.push(0);
            }
        }
    }
    int ans = 0;
    while (!que.empty()) {
        auto temp = que.front();
        int day = num.front();
        que.pop();
        num.pop();

        ans = std::max(ans, day);

        for (int i = 0; i < 4; i++) {
            int nx = temp.first + X[i];
            int ny = temp.second + Y[i];
            if (map[nx][ny] == 0) {
                map[nx][ny] = 1;
                que.push({ nx,ny });
                num.push(day + 1);
            }
        }
    }
}
```

ASCII CODE

- 종종 아스키 코드 값을 외워서 하시는 분들이 있는데 작은따옴표안에 문자를 넣으면 자동으로 아스키 코드로 변환된다.

```
#include<stdio.h>

char str[300];
int main() {
    int b;
    scanf("%s%d", str, &b);
    int ans = 0;
    for (int i = 0; str[i] != '\0'; i++) {
        int num;
        if (str[i] >= 'A' && str[i] <= 'Z') {
            num = str[i] - 'A' + 10;
        }
        else {
            num = str[i] - '0';
        }
        ans *= b;
        ans += num;
    }
    printf("%d", ans);
}
```



자리수

- 정수값에서 자리마다 처리를 할 때에는 나누기와 나머지를 적당히 사용한다,

```
#include<stdio.h>

int Rev(int num) {
    int ret = 0;
    while (num != 0) {
        ret *= 10;
        ret += num % 10;
        num /= 10;
    }
    return ret;
}
```



MODULO

- 나머지연산은 때에 따라서 귀찮은 코드를 간단하게 만들어준다.(e.g. ACM Hotel)
- 보통 나머지 연산을 하면 $0 \sim n-1$ 값이 나오지만 때에 따라서 $1 \sim n$ 값이 필요 할 때가 있다.
- 다음과 같은 상황에서는 옆과 같이 하면 된다.

```
#include<stdio.h>

int main() {
    int ans = 1;
    int e, s, m;
    scanf("%d%d%d", &e, &s, &m);
    int a = 1, b = 1, c = 1;
    while (a != e || b != s || c != m) {
        a++;
        b++;
        c++;
        a = (a - 1) % 15 + 1;
        b = (b - 1) % 28 + 1;
        c = (c - 1) % 19 + 1;
        ans++;
    }
    printf("%d", ans);
}
```



올림

- 일반적으로 정수값 나누기는 내림으로 계산 되지만 때에 따라 올림, 반올림이 필요할 수 있다.
- 다음과 같은 방법으로 올림, 반올림을 구현할 수 있다.

```
#include<stdio.h>

int main() {
    int n, a, b;
    scanf("%d%d%d", &n, &a, &b);
    int ans = 0;
    while (a != b) {
        a = (a + 1) / 2;
        b = (b + 1) / 2;
        ans++;
    }
    printf("%d", ans);
}
```



단위

- 시간과 같이 단위가 여러 개가 입력으로 들어오는 경우가 있다.
- 그러한 문제는 최소 단위로 변환한 후 풀면 편하다.

```
#include<stdio.h>

int main() {
    int h, m;
    scanf("%d%d", &h, &m);
    int M = h * 60 + m;
    M += 24 * 60 - 45;
    M %= 24 * 60;
    printf("%d %d", M / 60, M % 60);
}
```



진법 변환

- 작년 인터넷 예선에 특정 B진법으로 변환하는 문제가 있는데 이런 문제를 대비해서 진법 변환을 일반화 해놓은 것을 알아두면 편하다.
- 각 자릿수는 B^n 의 나머지 값들이기 때문에 다음과 같이 짜면 된다.

```
std::string func(int num,int b){  
    std::string str="";  
    while(num!=0){  
        str+=num%b+'0';  
        num/=b;  
    }  
    std::reverse(str.begin(),str.end());  
    return str;  
}
```



상수값 처리

- 자주 사용되는 상수(나머지 값)의 경우, 오타로 인한 오답을 방지하기 위해서 따로 선언해서 쓰는 것이 좋다.
- 또한, 여러 개가 있는 경우 배열로 묶어서 처리해주면 좋다.

```
#include<stdio.h>

const double price[5] = { 350.34, 230.90, 190.55, 125.30, 180.90 };

int main() {
    int tc;
    scanf("%d", &tc);
    while (tc--) {
        double ans = 0;
        for (int i = 0; i < 5; i++) {
            int a;
            scanf("%d", &a);
            ans += a*price[i];
        }
        printf("%.2lf\n", ans);
    }
}
```



정답이 없는 경우

- 문제에 따라 정답이 없을 때 -1와 같은 특정 값을 출력하고 할 때가 있다.
- 이러한 경우 원래 정답이 담겨있는 곳에 그 값을 넣는 방법을 써도 괜찮다.

```
#include<stdio.h>
#include<string.h>
#include<algorithm>
int dy[1001];
int main() {
    memset(dy, 0x7f, sizeof(dy));
    dy[0] = 0;
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        int a;
        scanf("%d", &a);
        for (int j = 1; j <= a; j++) {
            if (i + j < n) {
                dy[i + j] = std::min(dy[i + j], dy[i] + 1);
            }
        }
    }
    if (dy[n - 1] > n) {
        dy[n - 1]=-1;
    }
    printf("%d", dy[n - 1]);
}
```



점화식 초기화

- 피보나치와 같은 간단한 점화식의 경우 다음과 같이 간단하게 초기화를 할 수 있다.

```
#include<stdio.h>

unsigned long long dy[68] = { 1,1,2,4 };

int main() {
    for (int i = 4; i <= 67; i++) {
        dy[i] = dy[i - 1] + dy[i - 2] + dy[i - 3] + dy[i - 4];
    }
    int tc;
    scanf("%d", &tc);
    while (tc--) {
        int n;
        scanf("%d", &n);
        printf("%llu\n", dy[n]);
    }
}
```



STRING

- 문자열 처리의 경우 `char[]`보다 `std::string`이 확실히 효과적일 때가 많다.
- 하지만, 입출력 속도 문제로 `std::cin, std::cout`은 꺼려하기가 보통인데, `scanf`나 `printf`에서는 다음과 같은 방법으로 쓸 수 있다.

```
#include<stdio.h>
#include<string>

int main(){
    char s[100];
    scanf("%s",s);
    std::string str(s);
    printf("%s",str.c_str());
}
```

출처

- C++ reference - <http://www.cplusplus.com/>
- C++11 – choi backjoon, startlink <https://www.acmicpc.net/blog/view/10>
- 프로그래밍 대회 : C++11 이 야기 – choi Jongwook
http://www.slideshare.net/JongwookChoi/c11-draft?qid=9b8d3626-73a4-4d09-a5e5-4f5afaefb2f9&v=&b=&from_search=2

