9. pair : 2개의 다른 자료형을 한꺼번에 저장
#include <utility>
pair<int, char> p(15, 'H');
p.first --> 15
p.second --> 'H'
// 두 개의 pair를 비교할 때는 첫번째 값이 큰 pair가 더 크고, 첫번째 값이 같은 pair는 두번째 값이 큰 pair가 더 큰 pair이다.

10. map
#include <map>
map<char, int> m;
map[char] = int; 형식으로, char대신에 unsigned int가 들어가면 일반 배열과 같은 형식
m.count();
iter->first;, iter->second; // 접근

11. algorithm
#include <algorithm>
1) max(int, int)
리턴값 : 숫자
2) min(int, int)
리턴값 : 숫자
3) max_element(arr/iterator-처음, arr/iterator+size-끝) / min_element(arr/iterator, arr/iterator+size)
max_element(v.begin(), v.end())
min_element(v.begin(), v.end())
리턴값 : iterator
4) swap(int, int)
5) swap_ranges(바꿀 구간의 시작, 구간의 끝, 두번째 구간의 시작)
ex) swap_ranges(a, a+3, b) : 배열 a의 처음부터 3번째 값까지 b의 처음부터 3번째 값과 바꿔라
6) copy(복사할 대상의 시작점, 끝점, 복사할 위치의 시작점)
ex) copy(a+1, a+4, b+1) : 배열 a의 1~4 element를 b+1에서부터 차례로 붙여넣어라
7) fill(범위의 시작점, 끝점, 채울 값)
ex) fill(a+2, a+5, 0) : a+2부터 a+5까지를 0으로 채워넣어라
8) reverse(구간의 시작점, 끝점)
ex) reverse(a, a+5) : a부터 a+5까지를 뒤집음
9) rotate(구간의 시작점, 끝점, 옮겨질 위치)
ex) rotate(a, a+2, a+5) :
10) for_each(구간의 시작점, 끝점, 함수이름)
ex) for_each(v.begin(), v.end(), print) : v의 처음부터 끝까지를 print함수가 하라는 대로 수행
주의 : 매개변수 리스트는 같은 자료형 변수 하나여야 함
11) transform(구간의 시작점, 끝점, 결과 저장 범위의 시작점, 함수포인터)
ex) transform(v.begin(), v.end(), v1.begin(), twice) : v의 처음부터 끝까지를 twice함수의 리턴값을 v1에 넣어라
12) generate(구간의 시작점, 끝점, 함수포인터)
ex) generate(a, a+5, increase) : a의 처음부터 a[4]까지 increase함수의 리턴값으로 채워라

```
ios_base::sync_with_stdio(false);
cin.tie(NULL);
cout.tie(NULL);
```

13) find(구간의 시작점, 끝점, 찾을 값)
ex) find(a, a+10, 8) : a부터 a+10에서 값 8을 찾아서 주소를 리턴
==> 만약 리턴 값이 a+10이라면 값이 존재하지 않는 것으로 알 수 있음
14) find_if(구간의 시작점, 끝점, 함수 포인터)
ex) bool greaterThan4(int n){return (n>4);}
find_if(a, a+10, greaterThan4) : greaterThan4가 true인 값을 찾음
15) count(구간의 시작점, 끝점, 찾을 값), count_if(구간의 시작점, 끝점, 함수 포인터)
16) replace(시작점, 끝점, 대체할 값, 결과 값)
ex) replace(a, a+10, 2, 7) : 구간 안의 모든 2는 7로 대체
17) remove(시작점, 끝점, 지울 값)
ex) remove(a, a+10, 2) : 구간 안의 모든 2는 제거한 후 구간의 끝점의 주소 반환
18) remove_if(시작점, 끝점, 함수 포인터)
19) equal(시작점, 끝점, 비교할 구간의 시작점) : 두 구간이 같으면 true, 다르면 false 리턴
ex) if(equal(a, a+5, b)) : a에서 a+5까지, b에서 b+5까지가 같으면 treu, 다르면 false 리턴
20) mismatch(시작점, 끝점, 비교할 구간의 시작점) : 두 구간에서 최초로 다른 부분의 첫번째 구간의 다른 지점의 주소, 두번째 구간의 다른 지점의 주소를 pair의 형태로 리턴
ex) mismatch(a, a+5, b) : 두 구간의 처음으로 다른 부분을 pair로 리턴
==> pair는 algorithm을 선언했을 때 utility를 굳이 선언해 주지 않아도 사용 가능
21) adjacent_find(시작점, 끝점) : 구간에서 연속되면서 같은 값이 나올 때 첫번째로 등장하는 주소 리턴
22) unique(시작점, 끝점) : 인접한 여러 개의 동일한 값을 1개만 남기고 모두 지움
ex) unique(a, a+5) : 배열의 경우 마지막 칸 바로 뒤의 주소, vector는 end()의 위치 리턴
23) partition(시작점, 끝점, 함수포인터) : 전달한 함수의 값이 true인 것이 앞쪽으로, false인 것이 뒤쪽으로 위치를 바꾸고, 두번째 그룹의 시작점을 리턴함
ex) partition(a, a+10, isOdd) : 홀수는 앞으로, 짝수는 뒤로 배치
24) stable_partition(시작점, 끝점, 함수포인터) : stable한 partition함수
25) is_partitioned(시작점, 끝점, 함수포인터) : partition이 조건에 맞게 되어있는지 확인
26) sort(시작점, 끝점) : 매우 빠른 sorting함수 -> 우리가 짜는 것보다 매우 빠름
=> sort(a, a+10, greater) : 함수 포인터를 전달해서 전달한 함수가 false를 리턴할 때 두 값의 자리를 바꾸어 정렬
=> sort함수 내부에서 사용하는 연산자 오버로딩을 통해 정렬도 가능

```
- Scores 클래스 내부에서 연산자 오버로딩 -
bool operator <(const Scores& other){
    return math + english < other.math.other.english;
} // 총합이 낮은 학생부터 정렬
```

27) binary_search(시작점, 끝점, 찾을 값) : 배열이 이미 정렬되어 있을 때 binary search를 구현해줌
ex) binary_search(a, a+10, 60) : 값이 존재하면 true, 존재하지 않으면 false 리턴
28) lower_bound() : 주어진 값보다 크거나 같으면서 제일 작은 값을 찾고 주소를 반환하며 값이 없을 때 구간의 끝 반환
ex)lower_bound(a, a+10, 44)
29) upper_bound() : 주어진 값보다 크면서 제일 작은 값을 찾고 주소를 반환하며 값이 없을 때 구간의 끝 반환
ex)upper_bound(a, a+10, 44)
30) merge(첫번재 범위의 시작, 끝, 두번째 범위의 시작, 결과를 저장할 범위의 시작점) : merge sort에서의 merge와 같은 역할 -> sort되어있을 때 사용 가능
=> 결과를 저장할 곳에 충분한 공간이 할당되어있어야 함

// 여기부터는 2개의 리스트가 정렬되어있고, 중복된 값이 없어야 사용할 수 있음
// 결과의 끝 주소 또는 iterator를 반환 -> 시작주소와 연산 후 결과값의 개수를 알 수 있음
31) set_union(첫번재 범위의 시작, 끝, 두번째 범위의 시작, 결과를 저장할 범위의 시작점) : 합집합 연산
32) set_intersection(첫번재 범위의 시작, 끝, 두번째 범위의 시작, 결과를 저장할 범위의 시작점) : 교집합 연산
33) set_difference(첫번재 범위의 시작, 끝, 두번째 범위의 시작, 결과를 저장할 범위의 시작점) : 차집합 연산
34) set_symmetric_difference(첫번재 범위의 시작, 끝, 두번째 범위의 시작, 결과를 저장할 범위의 시작점) : 대칭 차집합 연산

www.cplusplus.com

DATA STRUCTURE WITH LIBRARY

date : . .

Doubly Linked List, Stack, Queue의 경우 STL에 라이브러리 존재

Tree

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

class node {
protected:
  int data;
  node* leftChild;
  node* rightChild;
public:
  node(int data, node* left = NULL, node* right = NULL);
  void setData(int data) { this->data = data; }
  void setLeft(node* n) { leftChild = n; }
  void setRight(node* n) { rightChild = n; }
  int getData() { return this->data; }
  node* getLeft() { return this->leftChild; }
  node* getRight() { return this->rightChild; }
  bool isLeaf() { return (leftChild == NULL && rightChild
== NULL); }
};

node::node(int data, node* left, node* right) {
  this->data = data;
  leftChild = left;
  rightChild = right;
}

#include "node.hpp"
#include <iostream>
#include <queue>
#include <algorithm>
using namespace std;
class tree {
protected:
  node* root;
public:
  tree(node* root = NULL) { this->root = root; }
  void setRoot(node* n) { root = n; }
  node* getRoot() { return root; }
  bool isEmpty() { return (root == NULL); }

  void inorderTraversal() { inorder(root); }
  void inorder(node* n);
  void preorderTraversal() { preorder(root); }
  void preorder(node* n);
  void postorderTraversal() { postorder(root); }
  void postorder(node* n);
  void levelorderTraversal();

  int getCount(); // return the number of nodes
  int getCount(node* n);
  int getHeight(); // return the height of the tree
  int getHeight(node* n);
  int getLeafCount(); // return the number of leaves
  int getLeafCount(node* n);
};
```

```
void tree::inorder(node* n) {
  if (n == NULL) { return; }
  else {
    inorder(n->getLeft());
    cout << n->getData() << " ";
    inorder(n->getRight());
  }
}

void tree::preorder(node* n) {
  if (n == NULL) { return; }
  else {
    cout << n->getData() << " ";
    preorder(n->getLeft());
    preorder(n->getRight());
  }
}

void tree::postorder(node* n) {
  if (n == NULL) { return; }
  else {
    postorder(n->getLeft());
    postorder(n->getRight());
    cout << n->getData() << " ";
  }
}

void tree::levelorderTraversal() {
  queue<node*> traversal;
  traversal.push(root);
  while (!traversal.empty()) {
    node* x = traversal.front();
    traversal.pop();
    if (x != NULL) {
      cout << x->getData() << " ";
      traversal.push(x->getLeft());
      traversal.push(x->getRight());
    }
  }
}

int tree::getCount() {
  if (root == NULL) { return 0; }
  else { return getCount(root); }
}

int tree::getCount(node* n) {
  if (n == NULL) { return 0; }
  else {
    return 1 + getCount(n->getLeft()) + getCount(n-
>getRight());
  }
}
```

```
int tree::getHeight() {
  if (root == NULL) { return 0; }
  else { return getHeight(root); }
}

int tree::getHeight(node* n) {
  if (n == NULL) { return 0; }
  else {
    return 1+ max(getHeight(n->getRight()), getHeight(n->getLeft()));
  }
}

int tree::getLeafCount() {
  if (root == NULL) { return 0; }
  else { return getLeafCount(root); }
}

int tree::getLeafCount(node* n) {
  if (n == NULL) { return 0; }
  if (n->isLeaf()) { return 1; }
  else {
    return getLeafCount(n->getLeft()) + getLeafCount(n->getRight());
  }
}
```

Graph
2차원 -> 2차원 저장

```
#include <iostream>
#include <vector>
#include <map>
#include <utility>
using namespace std;

class graph {
private:
  map<pair<int, int>, vector<pair<int, int>>> gra;
  int n;
  int rkfh, tpfh;
public:
  graph(int n, int rkfh, int tpfh, vector<vector<int>> a);
  vector<int> aDFS();
  vector<int> DFS(pair<int, int> p, vector<int>& numbers);
  void doDFS(pair<int, int> p, int& apt, map<pair<int, int>, int>& visited);
};
```

```
graph::graph(int n, int rkfh, int tpfh, vector<vector<int>> a) {
  this->n = n;
  this->rkfh = rkfh;
  this->tpfh = tpfh;
  for (int i = 0; i < a.size(); i++) {
    for (int j = 0; j < a[i].size(); j++) {
      pair<int, int> p(i, j);
      vector<pair<int, int>> asd;
      this->gra[p] = asd;
      if (a[i][j]) {
        this->gra[p].push_back(p);
        if (i != 0 && a[i - 1][j] == 1) {
          pair<int, int> pa(i - 1, j);
          this->gra[p].push_back(pa);
        }
        if (i != a.size() - 1 && a[i + 1][j] == 1) {
          pair<int, int> pa(i + 1, j);
          this->gra[p].push_back(pa);
        }
        if (j != 0 && a[i][j - 1] == 1) {
          pair<int, int> pa(i, j - 1);
          this->gra[p].push_back(pa);
        }
        if (j != a[i].size() - 1 && a[i][j + 1] == 1) {
          pair<int, int> pa(i, j + 1);
          this->gra[p].push_back(pa);
        }
      }
    }
  }
}

vector<int> graph::aDFS() {
  vector<int> numbers;
  for (int i = 0; i < rkfh; i++) {
    for (int j = 0; j < tpfh; j++) {
      pair<int, int> ps(i, j);
      if (gra[ps].size()) {
        return this->DFS(ps, numbers);
      }
    }
  }
  return numbers;
}
```

```
vector<int> graph::DFS(pair<int, int> p, vector<int>&
numbers) {
  map<pair<int, int>, int> visited;
  for (int i = 0; i < rkfh; i++) {
    for (int j = 0; j < tpfh; j++) {
      pair<int, int> pa(i, j);
      if (gra[pa].size()) {
        visited[pa] = 2; // not visited
      }
      else {
        visited[pa] = 0; // not a vertex
      }
    }
  }

  int apt = 1;
  visited[p] = 1;
  vector<pair<int, int>>::iterator it;
  it = gra[p].begin();
  if (gra[p].size() == 1) { apt = 1; } // no linked vertex
  else {
    it++;
    for (it; it != gra[p].end(); it++) {
      if (visited[*it] == 2) {
        visited[*it] = 1;
        apt++;
        doDFS(*it, apt, visited);
      }
    }
  }

  numbers.push_back(apt);

  for (int i = 0; i < rkfh; i++) {
    for (int j = 0; j < tpfh; j++) {
      pair<int, int> ps(i, j);
      if (gra[ps].size()) {
        if (visited[ps] != 1) {
          apt = 1;
          visited[ps] = 1;
          this->doDFS(ps, apt, visited);
          numbers.push_back(apt);
        } // if it is not visited
      }
    }
  }

  return numbers;
}
```

```
void graph::doDFS(pair<int, int> p, int& apt,
map<pair<int, int>, int>& visited) {
  vector<pair<int, int>>::iterator it;
  it = gra[p].begin();
  if (gra[p].size() == 1) { apt = 1; } // no linked vertex
  else {
    it++;
    for (it; it != gra[p].end(); it++) {
      if (visited[*it] == 2) {
        visited[*it] = 1;
        apt++;
        doDFS(*it, apt, visited);
      }
    }
  }
}
```

2차원 -> 1차원 저장

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int n;

class graph {
private:
  vector<vector<int>> gra;
  int n, m;
public:
  graph(vector<vector<int>> a, int m, int n);
  vector<int> DFS();
  int doDFS(vector<int>& visited, int point, int& space);
};

graph::graph(vector<vector<int>> a, int m, int n) {
  vector<int> gras;
  for (int i = 0; i < m * n; i++) {
    gra.push_back(gras);
  }
  this->m = m;
  this->n = n;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (a[i][j] == 0) {
        gra[m * i + j].push_back(m * i + j);
        if (i != 0 && a[i - 1][j] == 0) { gra[i * m +
j].push_back((i - 1) * m + j); }
        if (i != a.size() - 1 && a[i + 1][j] == 0) { gra[i * m +
j].push_back((i + 1) * m + j); }
        if (j != 0 && a[i][j - 1] == 0) { gra[i * m +
j].push_back(i * m + (j - 1)); }
        if (j != a[i].size() - 1 && a[i][j + 1] == 0) { gra[i * m
+ j].push_back(i * m + (j + 1)); }
      }
    }
  }
}
```

```
vector<int> graph::DFS() {
  vector<int> visited(m * n);
  for (int i = 0; i < m * n; i++) {
    if (gra[i].size() == 0) {
      visited[i] = 2;
    }
    else { visited[i] = 0; }
  }
  vector<int> space;

  for (int i = 0; i < m * n; i++) {
    if (visited[i] == 0) { visited[i] = 1; int spa = 1;
space.push_back(doDFS(visited, i, spa)); }
  }

  return space;
}

int graph::doDFS(vector<int>& visited, int point, int&
space) {
  for (vector<int>::iterator it = gra[point].begin(); it !=
gra[point].end(); it++) {
    if (visited[*it] == 0) {
      visited[*it] = 1;
      space++;
      doDFS(visited, *it, space);
    }
  }

  return space;
}
```

1차원 -> 1차원 저장

```
#include <iostream>
#include <vector>
#include <queue>
#include <map>
using namespace std;

class graph {
private:
  map<int, vector<int>> gra;
  int n;
public:
  graph(int n);
  void addEdge(int from, int to);
  void BFS(int from, vector<bool> &visited);
  int BFS();
};

graph::graph(int n) {
  vector<int> au;
  for (int i = 0; i <= n; i++) {
    gra[i] = au;
  }
  this->n = n;
}

void graph::addEdge(int from, int to) {
  gra[from].push_back(to);
  gra[to].push_back(from);
}

void graph::BFS(int from, vector<bool> &visited) {
  queue<int> aux;
  aux.push(from);
  while (!aux.empty()) {
    int th = aux.front();
    for (vector<int>::iterator it = gra[th].begin(); it !=
gra[th].end(); it++) {
      if (visited[*it] == false) {
        visited[*it] = true;
        aux.push(*it);
      }
    }
    aux.pop();
  }
}

int graph::BFS() {
  vector<bool> visited;
  for (int i = 0; i < n + 1; i++) {
    visited.push_back(false);
  }
  bool checker = true;
  int ccomponent = 0;

  while (checker) {
    checker = false;
    for (int i = 1; i < n + 1; i++) {
      if (visited[i] == false) {
        this->BFS(i, visited);
        checker = true;
        ccomponent++;
        break;
      }
    }
  }
  return ccomponent;
}
```