

DATA STRUCTURE



HAMM

INTRODUCTION

Curve of forgetting study planner

1HOUR 1DAY 2DAY 1WEEK 2WEEK 1MONTH 3MONTH

date : 2019 . 04 . 13

Measuring Program Efficiency

Two ways of measuring Program Efficiency

① Empirically (정량적으로)

② Analytically

// input: a[0] ... a[n-1]
// output: min

```
n번 loop      n번 addition      (n-1)번 comparison
for (i = 0; i < n; i++) {           // check if a[i] is the min
    isMin = true;
    for (j = 0; j < n; j++) {       n번 addition      (n-1)번 comparison
        if (a[j] < a[i]) isMin = false;
    }                                n번 comparison
    if (isMin) min = a[i];
}
```

$n^2 \{ n + (n-1)n \} + n + (n-1)$

→ n번 addition, (n-1)번 comparison, n번 comparison이 n번 진행

& n번 addition, (n-1)번 comparison

// input: a[0] ... a[n-1]
// output: min

```
// a[0] is the tentative min
min = a[0];
for (i = 1; i < n; i++) {           (n-1)번 addition      n번 comparison
    // see if a[i] is smaller than min-so-far
    if (a[i] < min) min = a[i];
}
```

$(n-1) + (n-1) + n = 3n - 2$

→ (n-1)번 addition, (n-1)번 comparison, n번 comparison

⇒ different platform (설정 조건 차이), ~~meaningful~~

→ 실제로 a[i]는 아니고 a[0]에서 a[n-1]이 됨이 남

→ a[0]를 대신 a[i]로 바꾸면 문제를 통계적으로 해결 가능

Order of growth

Big-O notation

If $f(n)$ can be bounded by $g(n)$ from above after multiplying an latter by an appropriate constant, $f(n) = O(g(n))$

For $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$, if there exist constants k and C s.t

$f(n) \leq C \cdot g(n)$ for all $n > k$, we say $f(n) = O(g(n))$

For $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$, if there exist constants k and C s.t

$f(n) \geq C \cdot g(n)$ for all $n > k$, we say $f(n) = \Omega(g(n))$

If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, we say $f(n) = \Theta(g(n))$

→ input의 size 뿐만 아니라 input의 contents가 running time에 영향

- usually focus on the worst case (= worst-case complexity)

// preprocessing
// do nothing

// query (x, y) comes



```
answer = 0;
for (i = x; i <= y; i++) {
    answer += a[i];
}
```

```
// preprocessing
for (i = 0; i < n; i++) {
    sum[i][i] = a[i];
    for (j = i+1; j < n; j++) {
        sum[i][j] = sum[i][j-1] + a[j];
    }
}
```

// now for all $x \leq y$, $\text{sum}[x][y]$ holds $a[x] + \dots + a[y]$

// query (x, y) comes
answer = sum[x][y];

구체적 쿼리가 있다고 하자.

	첫방법	둘째 방법
time complexity	$O(n^2)$	$O(n \log n)$
space complexity	$O(1)$	$O(n^2)$

⇒ N와 p에 따라서 더 미숙되는 방식이 물리적

설계적으로 time complexity 절도 고려할 것들이

많으면 주로 problem은 time complexity에 의함

- 또, overflow가 일어날 가능성도 있음

JAVA FOR C++ PROGRAMMERS

Curve of forgetting study planner

1HOUR 1DAY 2DAY 1WEEK 2WEEK 1MONTH 3MONTH

date : 2019 . 04 . 13

Basic Java Structures

Hello world!

```
Class Hello {  
    public static void main (String[] args)  
    { System.out.println("Hello, world!"); }
```

- Class 밖에 함수 선언 X
- class 이름과 파일 이름이 같아야 함

Primitives and objects

- Object variables work like "pointers"

```
class MyInt{  
    private int val;  
    public int getVal() {return val;}  
    public void setVal(int x){val=x;}  
}
```

```
선언: Myint a = new MyInt();  
MyInt b = new MyInt(), c = new MyInt();  
a.setVal(3); b.setVal(3);  
if(a==b) {System.out.println("true");}  
else {System.out.println("false");}  
⇒ 출력: false
```

- == 가 호는 같은 object를 가리고 있는지 확인한다.

Strings

```
String a="abcdeabcde";  
- a.length() : String의 길이 return.  
- a.indexOf("bc") : first occurrence return.  
- a.substring(2,4) : index 2이상 4미만의 String 리턴
```

Arrays

```
int[] a; a= new int[10];
```

* in JAVA, there is no delete operation.

FILE I/O

```
import java.io.*;  
  
Class Hello {  
    public static void main (String[] args){  
        String s, int i, x;  
        try{  
            BufferedReader rd = new BufferedReader(new FileReader("input.txt"));  
            BufferedWriter wr = new BufferedWriter(new FileWriter("output.txt"));  
  
            ...  
  
            rd.close(); wr.close();  
        }  
        catch (Exception e){  
            System.out.println ("ERROR");  
        }  
    }  
}
```

ARRAYS

Curve of forgetting study planner

1HOUR 1DAY 2DAY 1WEEK 2WEEK 1MONTH 3MONTH

date : 2019 . 04 . 14

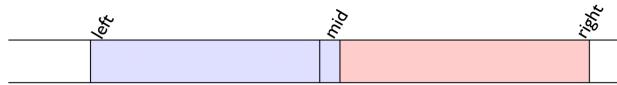
Array as a Data Structure

- Array: a set of "homogeneous" data
- Indexed by consecutive integers
- $O(1)$ time access

* the size needs to be specified when creation.

Binary Search

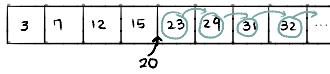
```
left = 0;  
right = n - 1;  
while (left < right) {  
    mid = (left + right) / 2;  
    if (a[mid] < x) left = mid + 1; else right = mid;  
} a[mid]가 x보다 작으면 무한루프에 빠질 수 있음. ex) left=6, right=7, a[6]=2인 경우  
if ((left == right) && (a[left] == x)) {  
    found = true; n=0일때를 체크하기 위해 필요  
    foundpos = left;  
} else found = false;
```



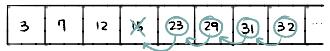
- $O(\log n) = O(\log n)$ 으로 표시.

- 하지만 sorted array로 계속 유지 필요.

.. Insertion: $O(n)$



.. deletion: $O(n)$



LINKED LISTS

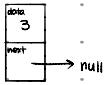
Curve of forgetting study planner

1HOUR 1DAY 2DAY 1WEEK 2WEEK 1MONTH 3MONTH

date : 2019 . 04 . 14

Linked Lists

Node: data + next pointer to next node
- 하나의 object로 취급

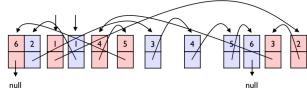


Linked List

access: O(n) → 첫 포인터부터 순차적인 접근

insertion: O(1) → 삽입 위치를 미리 알고 있을 때

deletion: O(1)



```
class Node {
    public int data; public Node next;
}
```

```
}
```

```
class LinkedList {
    public Node first, last;
    public LinkedList() {
        first = null; }
    public boolean isEmpty() {
        return (first == null); }
    public void InsertAtFront(int x) {
        Node newnode = new Node();
        newnode.data = x;
        newnode.next = first;
        first = newnode;
        if (last == null) last = newnode; }
    public int DeleteFirst() {
        int ret;
        if (first == null) return -1; // if the list is empty.
        ret = first.data;
        first = first.next;
        if (first == null) last = null; // 원래 list 끝나면 next는 null
        return ret; } // JAVA는 자동 garbage 처리이므로 explicitly handle X
    public void DisplayAll() {
        Node cur;
        for (cur = first; cur != null; cur = cur.next) {
            System.out.println(cur.data); }}
```

```
public void InsertAfter(Node n, int x) {
    Node newnode;
    if (n == null) return;
    newnode = new Node();
    newnode.data = x;
    newnode.next = n.next;
    n.next = newnode;
    if (newnode.next == null) last = newnode; }
```

```
public void InsertAtEnd(int x) {
    Node newnode;
    newnode = new Node();
    newnode.data = x;
    newnode.next = null;
    if (last == null) { first = last = newnode; }
    else { last.next = newnode; last = newnode; }}
```

```
}
```

Iterator

Class Iterator{

private Node cur, prev;

private LinkedList list;

public Iterator(Node first, LinkedList caller) {

list = caller; cur = first; prev = null; }

public boolean atEnd() {

return (cur == null); }

public int getData() {

if (atEnd()) { return 0; }

return cur.data; }

public void next() {

if (!atEnd()) { prev = cur; cur = cur.next; }

public void InsertAfter(int x) {

list.InsertAfter(x); }

public int DeleteCurrent() {

int ret;

if (cur == null) { return -1; }

ret = cur.data; cur = cur.next;

if (prev == null) { list.first = cur; } // 만약 curr가 first였다면

else { prev.next = cur; }

if (cur == null) list.last = prev; // 만약 cur가 last였다면

return ret; }

→ LinkedList 클래스에 iterator 선언부 추가.

class LinkedList{

public Iterator getIterator() {

return new Iterator(first); }

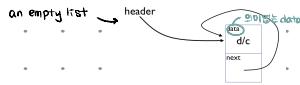
Circular linked lists

- no inherent "first" & "last" item

ex) Round-robin scheduling

- code becomes simpler

- Header node : never deleted



Class CircularLinkedList{

private final Node header;

public CircularLinkedList() {

Node newnode = new Node();

newnode.next = newnode;

header = newnode; }

public boolean isEmpty() {

return (header.next == header); }

public CircularIterator getIterator() {

return new CircularIterator(header); }

Class CircularIterator{

private Node header, prev, cur;

public CircularIterator(Node h) {

header = h; prev = h; cur = h.next; }

public boolean atEnd() {

return (cur == header); }

public int getData() {

if (atEnd()) { return -1; }

return cur.data; }

public void next() {

prev = cur; cur = cur.next; }

... to be continued ...

LINKED LISTS

Curve of forgetting study planner

1HOUR 1DAY 2DAY 1WEEK 2WEEK 1MONTH 3MONTH

date : 2019 . 04 . 14

```
public void insertAfter(int x) {  
    Node newnode = new Node();  
    newnode.data = x;  
    newnode.next = cur.next;  
    cur.next = newnode; }  
  
public int deleteCurrent() {  
    int ret;  
    if (cur == header) { ret=-1; }  
    else {  
        ret = cur.data;  
        prev.next = cur.next;  
        cur = cur.next; }  
    return ret; }
```

Doubly Linked Lists

— Node의 차이



이전 노드를 가리키는 포인터 prev

STACKS & QUEUES

RECURSION

Curve of forgetting study planner

1HOUR 1DAY 2DAY 1WEEK 2WEEK 1MONTH 3MONTH

date : 2019. 4. 15

Queues

- homogeneous items forming a line
- FIFO (First In First Out)
- Enqueue : adds an item to the rear
- Dequeue : removes data from the front

Implementing : linked list

class Queue {

```
private final LinkedList llist;
public Queue() { llist = new LinkedList(); }
public void enqueue(int x) { llist.insertAtEnd(x); }
public int dequeue() { return llist.deleteFirst(); }
public boolean isEmpty() { return llist.isEmpty(); }
}
```

Implementing 2: array

			3	5	13	9	
--	--	--	---	---	----	---	--

- enqueue, dequeue 하다보면 Component가 끌리는 경우 생김.

Circular structure을 이용

class Queue {

```
private final int[] data;
private final int size;
private int front, rear;
public Queue(int s) {
    data = new int[s];
    size=s; front=0; rear=-1+s;
}
public boolean isEmpty() { return ((rear+1)%size)==front; }
public boolean isFull() { return ((rear+2)%size)==front; }
public enqueue(int x) {
    if(!isfull()) {
        rear = (rear+1)%size;
        data[rear]=x;
    }
}
public int dequeue() {
    int ret;
    if(isEmpty()) return -1;
    ret= data[front];
    front = (front+1)%size;
    return ret;
}
```

★ 구현시 주의사항

length를 array 길이로 접근하지, 아니면 element 최대 수 접근지에 따라 나누는 수 다음
(front + last) 연산 X ⇒ 복잡한 티 연산, +2 연산으로 해결하기.

Stacks

- homogeneous items forming a stack w/ one end open.
- LIFO (Last In First Out)
- push adds an item to the top
- pop removes an item from the top

Implementing 1: linked list

class Stack {

```
private final LinkedList llist;
public Stack() { llist = new LinkedList(); }
public void push(int x) { llist.insertAtFront(x); }
public int pop() { return llist.deleteFirst(); }
public boolean isEmpty() { return llist.isEmpty(); }
}
```

Implementing 2: array

+ maximum size를 알 때.

class Stack {

```
private final int[] data;
private final int size;
private int top;
public Stack(int s) {
    data = new int[s];
    size=s;
    top=-1;
}
public boolean isEmpty() { return top == -1; }
public boolean isFull() { return top == (size-1); }
public void push(int x) { if(!isFull()) data[++top] = x; }
public int top() { if(isEmpty()) return -1; else return data[top]; }
}
```

Example of using Stacks

① matching parentheses

pseudo code.

```
if "(" or "{" push to a Stack
else if ")" and tos is "(" pop
else if "}" and tos is "{" pop
if stack is empty return true
else return false.
```

② Recursion

Tower of Hanoi

X Stack

class Hanoi {

```
public static void Hanoi(int n, int from, int to) {
    if(n<=0) return;
    int aux = 6 - from - to;
    Hanoi(n-1, from, aux);
    System.out.println("Move a disk from rod " + from);
    System.out.println("to rod " + to);
    Hanoi(n-1, aux, to);
}
```

STACKS & QUEUES

RECURSION

Curve of forgetting study planner

1HOUR 1DAY 2DAY 1WEEK 2WEEK 1MONTH 3MONTH

date :

with stack

```

class Context{
    public int n, from,to;
    public int whereToReturn;
    public Context(int n, int f, int t, int w){
        this.n=n, from=f, to=t, whereToReturn=w;
    }
}

public static void Hanoi (int n, int from, int to) {
    Stack s = new Stack(n);
    Context ctx; boolean calldone;
    ctx = new Context (n,from,to,0);
    while (true) {
        calldone = false;
        switch (ctx.whereToReturn) {
            case 0:
                if (ctx.n <=0) {calldone = true;}
                else {
                    s.push (new Context (ctx.n,ctx.from, ctx.to,1));
                    ctx = new Context (ctx.n-1, ctx.from, 6-ctx.from-ctx.to ,0);
                }
                break;
            case 1:
                System.out.println ("Move a disk from rod " + ctx.from + " to rod",
                        ctx.to);
                s.push (new Context (ctx.n,ctx.from,ctx.to,2));
                ctx = new Context (ctx.n-1, 6-ctx.from-ctx.to,ctx.to,0);
                break;
            case 2:
                calldone = true;
                break;
        }
        if (calldone) {
            if (s.isEmpty()) break; else ctx=s.pop();
        }
    }
}

```

③ postfix calculation.

if 이번 content operand push
else pop 필요한 operand를 & 계산 & 결과 push.
마지막 결과 pop

④ Converting from infix to postfix.

- Stack 외부의 < 는 제일 높은 우선순위, 내부의 < 는 제일 낮은 우선순위를 갖는다

pseudo code

```

Initialize an empty stack s of operators
for each token x of the expression, from left to right
    if x is an operand then output x
    else if x is ) then
        while top of stack is not (
            pop and output an operator
            pop & output에만 결정한다
        else
            while top Stack이 Empty일 때 of stack has higher or equal precedence than x
                pop and output an operator
                push x
    repeat pop and output an operator until s is empty

```


TREES

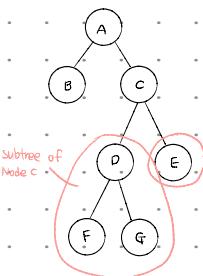
Curve of forgetting study planner

1HOUR 1DAY 2DAY 1WEEK 2WEEK 1MONTH 3MONTH

date : 2019 . 4 . 17

Trees

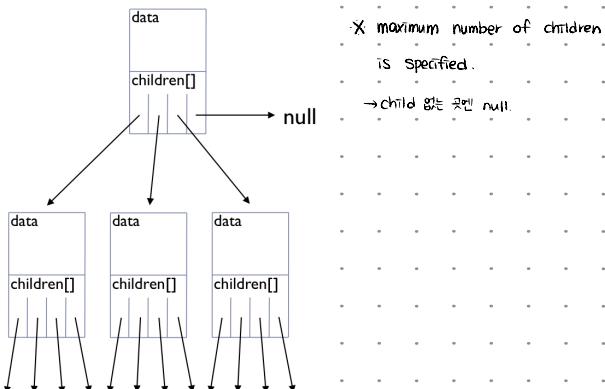
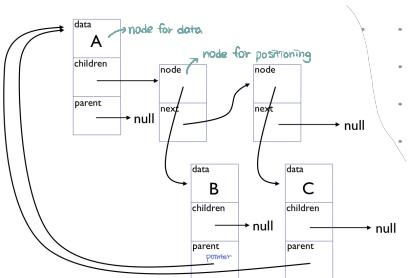
Terminology



- Nodes : data를 담고 있는 부분
- edges : Node를 잇고 있는 선.
- child & parent : Edge로 이어져 있는 상위는 parent, 하위노드는 child node.
- (ancestor - parent - child - descendant)
- Sibling : parent Node의 같은 Node들
- leaves Nodes : child Node가 없는 Node (\leftrightarrow internal nodes)

- A : root Node
- tree consists D, F, G : Subtree rooted at D
- degree : number of the Node itself's children.
- path : unique(simple) line between a pair of nodes
- level : length of the unique path from root to vertex.
- height : maximum value of level of the node.

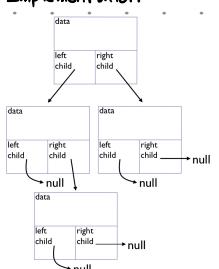
Implementation



Binary Trees

- ordered tree that every node has equal or less than 2 children.
- Child node : left child, right child
- level i \rightarrow node maximum # : 2^i
- height \Rightarrow maximum node #: 2^{h-1} (\rightarrow full binary tree)

Implementation



① pointer 사용

max # of children이 정해져 있으므로 한 노드 당 Children pointer를 두개씩 만들어 사용

② array

- root node : index 1
- parent node : index $\lceil \frac{n}{2} \rceil$
- left child node : index $2i$
- right child node : index $2i+1$

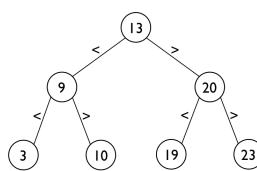
Traversal

- visiting every node of the given tree
- preorder traversal : root-left-right
- inorder traversal : left-root-right
- postorder traversal : left-right-root \rightarrow postfix를 알 수 있다

Binary Search Trees

Binary Search

3 9 10 13 19 20 23



* 구현시 주의할점

1. delete할 때와 insert할 때 parent노드 찾는 방법이 달라야 함!
2. 만약 right subtree의 min이 delete노드인 경우 예외 처리필요!

- binary search tree is either

- an empty tree
- the keys of the left tree are smaller than its of the root.
- the keys of the right tree are bigger than its of the root.
- both subtrees are binary search tree
- key 값은 모두 distinct.

Operations

① insertion

- i) 원래 insert하고자 하는 값이 이미 있으면? \rightarrow searching 수행
 - ii) 없다면 마지막에 삽입해야 하는가?
- 삽입위치 찾으면 pointer 연결
 \rightarrow height of the tree
- time complexity : search $O(h)$ + insert $O(1) \Rightarrow O(h)$

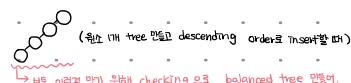
② deletion

- i) delete할 노드가 tree에 있는가?
 - ii) 있다면 위치는 어떤가?
- 1) children 수 = 0인 경우 : 그냥 delete
 - 2) children 수 = 1인 경우 : child를 delete하는 자리에 옮기고 delete
 - 3) children 수 = 2인 경우 : 원쪽 subtree의 max 또는 오른쪽 subtree의 min을 delete할 노드 자리로 옮기고 연결.
- time complexity : search $O(h)$ + $\begin{cases} 1) 2) delete O(1) \\ 3) delete O(h) \end{cases} \Rightarrow O(h)$

* h와 n의 관계

- 평균적으로 : $h = \log_2 n$

- worst case : $h = n$



③ Joining two trees

two binary search trees A,B \rightarrow keys : A < B

- $\left[\begin{array}{c} \text{left subtree} \leq \text{max} \\ \text{right subtree} \leq \text{min} \end{array} \right] \rightarrow \text{rootnode} \text{로 하는 new-tree 생성}$

- time complexity : search $O(h)$ + joining $O(1) \Rightarrow O(h)$

주 퀴즈 주의할 점

④ splitting

null을 연산하고 있는 것은 아닌지 확인해보기 / parent가 정상이 되어 연산 진행

- 주어진 key 값을 기준으로 2개의 binary search tree로 split. (elt: bigger than given key, llt: smaller than given key)

[주어진 key보다 cur key가 크면 → 그 노드의 right subtree로 second tree]

[주어진 key 보다 cur key가 작으면 → 그 노드의 left subtree로 first tree]

- time complexity

Iterators

Binary Tree

① postorder iterator

```
class PostIter {
    Stack s;
    public PostIter(Node r) {
        s = new Stack();
        pushNextDesc(r);
    }
    public void pushNextDesc(Node n) {
        while (n != null) {
            s.push(n);
            if (n.left == null) n = n.right; else n = n.left;
        }
    }
    public int getData() {
        if (atEnd()) return -1;
        return s.peek().data;
    }
    public void next() {
        if (atEnd()) return;
        Node prev = s.pop();
        if (!s.isEmpty()) {
            if (s.peek().left == prev) pushNextDesc(s.peek().right);
        }
    }
    public boolean atEnd() {
        return s.isEmpty();
    }
}
```

② inorder iterator

```
class InIter {
    Stack s;
    public int getData() {
        if (atEnd()) return -1;
        return s.peek().data;
    }
    public boolean atEnd() {
        return s.isEmpty();
    }
    public void pushLeft(Node n) {
        while (n != null) {
            s.push(n);
            n = n.left;
        }
    }
    public InIter(Node r) {
        s = new Stack();
        pushLeft(r);
    }
    public void next() {
        if (atEnd()) return;
        Node n = s.pop();
        if (n.right != null) pushLeft(n.right);
    }
}
```

③ preorder iterator

```
class PreIter {
    Stack s;
    public int getData() {
        if (atEnd()) return -1;
        return s.peek().data;
    }
    public boolean atEnd() {
        return s.isEmpty();
    }
    public PreIter(Node r) {
        s = new Stack();
        if (r != null) s.push(r);
    }
    public void next() {
        Node cur = s.pop();
        if (cur.right != null) s.push(cur.right);
        if (cur.left != null) s.push(cur.left);
    }
}
```

Level order traversal

: 각 노드를 레벨 순으로 순회.

→ 이전의 traversal 과의 차이: queue 사용. (이전 방법들은 stack 사용)

pseudo code

```
initialize queue
queue.enqueue(root);
while queue != empty
    n ← queue.dequeue();
    if (n!=NULL)
        print n.data
        queue.enqueue(leftchild);
        queue.enqueue(rightchild);
```

HEAPS

Curve of forgetting study planner

1HOUR 1DAY 2DAY 1WEEK 2WEEK 1MONTH 3MONTH

date :

Priority Queues

- operation : insert, query max, delete max, query min, delete min

Heaps

- complete binary tree
- children의 key 값보다 parent의 key 값이 더 크다.
- array representation 사용.
 - : complete 힙 아니면 array를 할당해놓고 맨쓰는 공간이 필요

Basic heap operations

① Create a heap : allocating a new array

(가정 : max # of array가 정해짐)

- time complexity : $O(1)$

② inserting a new item

- 제일 마지막에 삽입하고 힙의 성질을 만족하기 위해 성립될 때까지 bottom-up 실행

- time complexity : insert $O(1)$ + bottom-up $O(n) \Rightarrow O(n)$

③ Querying the maximum root of the heap

- time complexity : finding $O(1)$

④ deleting the maximum

- 제일 마지막 노드를 가져와서 root에 넣은 뒤 top-down Heapify 진행

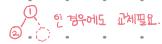
* Heapify?

한 노드의 두 subtree가 모두 Heap일 때 root node도 힙의 성질을 지킬 수 있도록 swap

How? children 둘 다 값이 작다면 힙 성질은 유지된다.

- time complexity : insert $O(1)$ + heapify $O(h) \Rightarrow O(h)$

* 구현할 때 주의!!



인 경우에도 교체 필요.

④ index 틀어가는지 확인

⑤ union : 두 힙을 하나의 힙으로 합침

→ 배열을 초기화하고 buildHeap 연산 수행
→ 자료구조가 힙으로 변환되는 과정에서 힙 속성 위반

⑥ delete

[~~first~~ 원하는 element와 비교하는 property 있는는 번선 필요 (heapify / bottom-up)]

decreaseKey to -∞ delete minimum

Heap Sort

① Heapify from the bottom

- Heapify를 적용시키려면 subtree Heap이라 하므로 bottom부터 실행

- leaf node는 실행해도 바로 리턴되므로 마지막 노드의 parent 브터 Heapify 진행

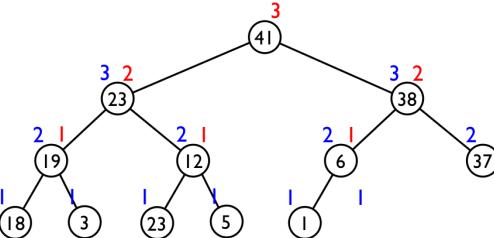
② repeatedly delete the maximum

- Heap의 맨巅에 하나의 delete maximum 수행 후 array에 넣는 과정은 정렬되어 있음.

- time complexity : $\frac{n}{2} \lceil \log_2 n \rceil = O(n \log n) + \text{heapify } O(n) \Rightarrow O(n \log n)$

* Heapify는 왜 $O(n)$?

$S = \text{Heapify}$ 연산 수행 시간이라 하면



- 파란색 숫자들의 합 = S

- 빨간색 숫자들의 합 = 25

- $S = 25 - S = (\text{root노드의 } 2S - S) + (\text{다른 노드들의 } 2S - S)$

$\downarrow \text{트리의 높이}$

$$= \log_2 n + 2 \cdot \lceil \frac{n}{2} \rceil \Rightarrow O(n)$$

GRAPH

Curve of forgetting study planner

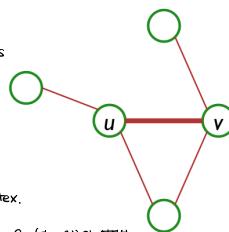
1HOUR 1DAY 2DAY 1WEEK 2WEEK 1MONTH 3MONTH

date :

Graphs

Terminology

- Graph $G = (V, E)$
- V : Vertices (singular vertex) $\rightarrow V(G)$: set of vertices
- E : Edges $\rightarrow E(G)$: set of edges
- 그래프에서 edge로 연결되어 있으면 adjacent하다.
- ex) u 와 v adjacent. edge (u, v) 로 연결되어 있다.
- cycle : a path starts and ends at the same vertex.
- path : length n from u to $v \Rightarrow e_1 = (u, x_1), e_2 = (x_1, x_2) \dots e_n = (x_{n-1}, v)$ 의 연결선.
- simple path : vertex 중복되지 않는 path.
- simple cycle : 길이 2 이상 vertex 중복이 아닌 cycle.
- acyclic : simple cycle이 없는 graph.
- degree of vertex : incident한 edge #
- Subgraph : $G' = (V', E')$ 일 때 $V' \subseteq V$ 이고, E' 는 E 의 subset일 때 G' 은 G 의 subgraph



- vertex-induced subgraph : specify vertex first \rightarrow 연결된 all edges 가져오기
- edge-induced subgraph : specify edge first \rightarrow 연결된 all vertices 가져오기
- weighed graph : each edge가 numerical weight로 annotated 되어 있는 graph
- connected graph : every pair of vertices on path이 있는 graph.
 \hookrightarrow unconnected 일 때 그 subgraph들은 각각 connected component로.

Tree & Graph

- tree : an acyclic connected graph.

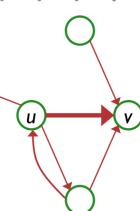
- [rooted tree]
- [unrooted tree]

X tree / graph의 차이 (degree)

[tree : children 수]

[graph : vertex 와 연결된 edge 수]

- spanning tree : Graph G의 모든 vertices를 모두 포함하는 subgraph.

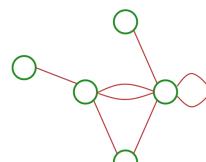


Directed Graphs

- directed graph (digraphs) : edge는 ordered pair of vertices
- adjacent from/to : $<u, v>$ (u to v), $<v, u>$ (v to u) \rightarrow head to tail
- degree
 - [in-degree : vertex로 들어오는邊으로의 edge 수]
 - [out-degree : vertex로 흘러나가는 边으로의 edge 수]
- strongly-connected : $u \rightarrow v, v \rightarrow u$ 의 direct path이 존재 하는 경우.
- underlying graph : directed graph의 orientation을 무시한 undirected graph.

Multigraph & pseudograph

- multigraph : 같은 pair of vertices 사이로 edge가 여러개인 경우.
- loop : an edge to itself
- pseudograph : loop를 사용할 수 있는 multigraph.



Internal representation

Adjacency matrix (인접행렬)

각 행렬의 (i, j) element가 존재하면 $A[i][j] = 1$, 존재하지 않으면 $A[i][j] = 0$

weighted graph의 경우 edge에 special value 있으면 weight 저장

Adjacency List (인접리스트)

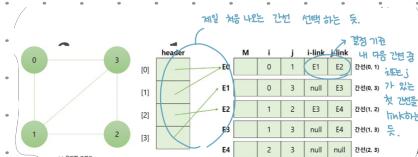
각 vertex마다 해당 vertex를 전부 노드에 저장하는 linked list로 연결하고 이를 하나의 배열에 저장

weighted graph의 경우 weight도 같이 노드에 저장하는지

inverse adjacency list : store edges to vertex i

예제 : duplicate copies of each edge

Adjacency multilist



간선을 노드에 표기하는 linked list로 만들 / 숨겨지거나 몇 줄

Graph Search

Depth first search (DFS)

adjacent vertex에 recursive한 traversal

- ① vertex 0 \rightarrow vertex 1
- ② vertex 1 \rightarrow vertex 3
- ③ vertex 3 \rightarrow vertex 4
- ④ vertex 1 \rightarrow vertex 4 (already visited)

DFS를 수행하면 all connected component 찾을 수 있음

Breadth first search (BFS)

queue 사용

enqueue를 할 때 visiting vertex 표기

visit vertices in nondecreasing order of shortest path lengths from v

running time : $O(|V|^2)$, $O(|V|+|E|)$

Minimum Spanning Trees

Weighted graph의 total edge cost를 최소화하는 spanning tree

* G is connected, acyclic 이면 $|E|=|V|-1$

Key Lemma

Graph G의 nonempty SubSet인 S가 있을 때, edge e를 S와 S의 endpoint, S밖에 다른 endpoints를 가진 edge 중 minimum cost edge이면 every MST는 e를 포함한다.

증명) e를 포함하지 않는 다음 정리는 MST가 필요로 하는 MST / fx 20인 MST (T')

\hookrightarrow MST 비교를 통해 증명

Kruskal's algorithm

empty graph T를 초기화

graph G의 edges를 nondecreasing order로 정렬한 edge를 포함한 이중 사이클이 있는다면 T에 추가.

$|V|-1$ 개의 edge가 되면 종료.

모든 iteration을 끝나니 edge 수가 $|V|-1$ 보다 적다면 MST는 MST에 존재하지 않는 것을 알수 있다. (edge not distinct)

i) cycle 체크를 dfs/bfs로 한다면 : $O(|E|log|E| + |E||V|) = O(|E||V|)$

ii) cycle 체크를 union-find로 한다면 : endpoint 2개의 Find연산으로 same subset이면 cycle 형성

$\rightarrow O(|E|log|E|) + |E||V| = O(|E||V|)$: no improvement

iii) cycle 체크를 weighted union-find로 한다면

$\rightarrow O(|V| + (|V| + |E|) + |E|log|E| + |E|log|V|)$

$\rightarrow O(|V| + |E|log|V|)$

$\rightarrow O(|E|log|V|)$ if G is connected

GRAPH

Curve of forgetting study planner

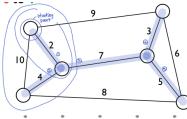
1HOUR 1DAY 2DAY 1WEEK 2WEEK 1MONTH 3MONTH

date :

Prim's algorithm

$V(T)$ 와 $V(G) \setminus V(T)$ 사이의 edge \geq minimum cost edge 를 T에 추가.

이용 만족하는 edge 경로에서 일련 순서로 disconnected graph.



T의 edge 수가 $|V|-1$ 이 될 때까지 반복.

(= Key Lemma 그대로)

- running time : $O(|V| + |E| \cdot |V|)$ $\Rightarrow O(|V| \cdot |E|)$

[Implementation]

- using a heap (of edges) : min heap of edges cost

: 선택된 vertex 주제로 새롭게 (candidate) 되면 adjacent edges heap에 insert하고 그에 minimum cost edge를 찾아서

Algorithm

여기서 문제는?

만약 candidate가 되었지만, 다른 candidate 중에 저렴한게 있는 경우 heap에 minimum 관계를 두었으므로 이를 delete 할 때 문제가 생김.

\Rightarrow ghost edge로 남겨두고 minimum에 맞을 때 candidate가 아니면 그때 지워주는거임.

- running time : $O(|V|(|V|+|E| \log |E|)) \Rightarrow O(|E| \log |V|)$: connected

Shortest Path

Dijkstra's algorithm

Weighted (dir) graph 이고 모든 edges weight이 nonnegative일 때 vertex set의 shortest path를 찾는다.

[Key observation]

vertex s 에서 path의 second-to-last vertex 를 a 로 하면 shortest path의 $a \rightarrow t$ 를 하면 shortest path의 $s \rightarrow t$ 를 찾았을 때

* Dijkstra's algorithm 풀이

Set S : shortest paths of vertices의 집합

$d[v]$: shortest path from s to v ($d[s] = 0$) // 실제 shortest path는 s 와 s 와 같은 vertex 중 2nd-to-last를 찾았을 때

의 shortest path // direct path가 있는 경우 바로 초기화

$prev[v]$: shortest path의 2nd-to-last vertex

$d[v]$ 의 처음 작은 vertex를 찾으면 s 에 포함된 거리 (St singleton 시작) 진행. 이제 새로운 vertex 포함하면서 $d[v]$ 를 업데이트

update 사용법 (s 는 add된 vertex) > 2nd-to-end vertex의 경우와 함께 $d[v]$ 갱신 내용)

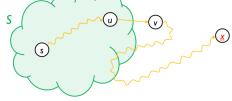
> If $d[x] = \min_{v \in S} d[v]$, $d[x]$ is shortest length of an $s-x$ path

> Let P be an arbitrary path from s to x

> P has to "cross" the boundary from S to $V \setminus S$:

> let $s \sim v$ be the first arc on the boundary

> (length of P) \geq (length of the $s-v$ prefix of P) $\geq d[v] \geq d[x]$



infinity boolean flag은 사용하는데 safest한 것으로 대체하여 사용

- running time : $O(|V|^2)$

path는 $prev$ 를 따라다니면서 찾을 수 있음

Min-Heap

decreasekey operation 추가.

key값을 조건문으로 바꾸고 property 만족할 때까지 parent와 swap 진행 / root 끌때까지

- running time : $O(\log n)$

Improvement

▷ Prim : min heap of edges : cost x change

▷ dijkstra : min heap of vertices, $d[v]$: value can change (decrease only.)

$d[v]$ 를 minheap에 만들 때, $d[v]$ 가 다른 vertex를 친구가면서 간접

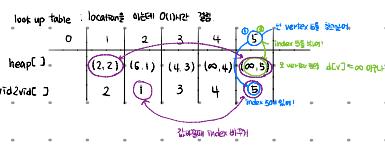
decreasekey operation으로 $d[v]$ 를 update 진행

$d[v]$ 의 최솟값 찾는 연산 : deleteMin 연산 $|V|$ 번

$d[v]$ update : decreasekey 연산 (TIE 번) $\rightarrow O(|V| + |E| \log |V|)$ $\rightarrow O(|E| \log |V|)$: weakly connected

\Rightarrow 풀이 : decreasekey 를 하면 vertex의 index를 찾고 \rightarrow search entire heap 했음

: need to maintain index of vertex in heap



\Rightarrow running time : $O((|V|+|E|) \log |V|)$, $O(|E| \log |V|)$: weakly connected

Topological Sort

directed graph에서 전체 vertex를 방문하는 순서를 정해줌.

\rightarrow pre-registered 모든 vertex를 찾아서 미리나감 / empty인 경우 디버깅

But : acyclic인 Order를 찾을수 없음

Directed Acyclic Graphs (DAGs) 때문에 topological order가 존재함

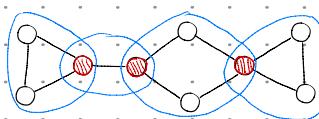
Biconnected graphs

- articulation point (cut vertex) : graph를 2개로 나누는 vertex.

- biconnected graph : articulation point이 없는 graph

- biconnected component : maximal biconnected subgraph

additional vertex 추가 시 still biconnected인 vertices가 있음



DFS Tree

- Tree edges : DFS 속도를 가속화하는 edge

- Back edge : DFS에서 사용하지 않는 edge

- Cross edge : directed graph에서만 드래그

Articulation Point 찾기

- $dfn[v]$: vertex v 의 DFS 번호

- $low[v]$: $\min\{dfn[v], low[z] \mid z \in v \text{ child}, dfn[z] : (v,z) \text{ is back edge}\}$

root node ($mild$ 2nd) : $low[v] = v \text{ child}$, non-root nodes ($low[v] \geq dfn[v]$) : $v \text{ child } \neq \text{cut vertex}$

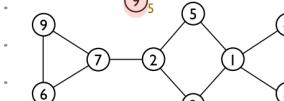
Finding biconnected components

The main routine calls $findBC(root, dummy)$, with $dfn[]$ initialized as zero

procedure $findBC(v, parent)$

```

    set  $dfn[v] = v$ 의 DFS 번호
     $low[v] \leftarrow dfn[v]$  : initial value
    for each incident edge  $(v, x)$ 
        if  $x = \text{parent}$  or  $dfn[v] \leq dfn[x]$  then continue
        push  $(v, x)$  : back edge queue
        if  $x$  has been visited then
            update  $low[v]$  to  $low[x] \cup ?$ 
        else
            findBC( $x, v$ )
            update  $low[v]$  to  $low[x] \cup ?$ 
            if  $low[x] \geq dfn[v]$  then
                repeat
                    pop and output an edge
                until the edge is  $(v, x)$ 
    
```



UNION-FIND

Curve of forgetting study planner

1HOUR 1DAY 2DAY 1WEEK 2WEEK 1MONTH 3MONTH

date :

Union-Find

maintains family of disjoint sets S_i

Operations

- Union (i, j) : S_i, S_j 를 합치는 연산 (전제조건: i, j 의 값은 알고 있어야 함: root node value)
- find(x) : x 를 포함하는 set의 root를 찾아서 return (전제조건: x 를 포함하는 set은 현재도 유일함)

Representation (tree는 pointer로 구현)

Union : 다른 tree를 끌어 tree의 subtree로 만듬

- $O(1)$ time

- i, j 를 모르는 경우, 그들 내부 component를 증명한 Find연산 후 실행해야 함

Find

- 주어진 값이 포함된 tree의 root 리턴

- $O(n)$ time

Internal representation

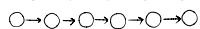
각자 자신의 parent를 가지고 있어야 함 (→비결선인)

parent가 없으면 -1로 선언. Initialization도 모두 -1로 선언

Initialization : $O(n)$ time

Weighted Union

Worst case of Union-Find



Representation

- Union : 원 union연산과 동일하지만, 더 작은 cardinality의 tree를 subtree로 만든다.

Cardinality update 필요함

- Find : 위연산과 동일

Internal representation

- parent 저장 array

- cardinality 저장 array: initialize by -1

Improvement

maximum height of a tree : $\lceil \log_2 k \rceil$ (k : cardinality)

proof) ① $k=1$ 일 때 $h=0$: $\lceil \log_2 1 \rceil = 0$

② $k \geq 2$ 일 때 union 전의 상태를 보면 $k=k_1+k_2$, $k_1 < k_2$

$$\begin{aligned} \lceil \log_2 k \rceil &\geq \left\lceil \begin{array}{c} \text{---} \\ k_1 \\ \text{---} \end{array} \right\rceil + \left\lceil \begin{array}{c} \text{---} \\ k_2 \\ \text{---} \end{array} \right\rceil \leq \lceil \log_2 k_1 \rceil + \lceil \log_2 k_2 \rceil \end{aligned}$$

$$\lceil \log_2 k \rceil - 1 = \text{overall union 이후 } k_2 \text{에 의해 height 증가. : } \lceil \log_2 k_2 \rceil - 1 + 1 = \lceil \log_2 k \rceil$$

$$\text{union 이후 } k_2 \text{에 의해 height 증가. : } \lceil \log_2 k_2 \rceil$$

$$(\text{maximum height of subtree which cardinality is } k) = \lceil \log_2 k_2 \rceil$$

\Rightarrow Find연산 수행시간: $O(\log n)$

Improvement

path compression

find연산을 반복적으로 호출해 하면 비효율적. $O(n \log n)$ 빙글

remember하자 [① storing : examine 시간 더 걸릴지도

[② change the node to the child of its root : 자녀는 모든 node에 대해서

같은 연산 수행

- find연산 진행하면서 O(1) 연산 수행: $O(n \log n)$ 빙글 X

RED-BLACK TREE

Red-black trees

* leaf node : null, 3. 단일 노드, 4. 노드가 2개의 자식 노드, 5. 2개의 자식 노드, 6. 2개의 자식 노드.

tree property

- p1) Every node is colored either red or black
- p2) The root is black
- p3) Every leaf/nil pointer is black
- p4) If a node is red, both its children are black.

- p5) Every simple path from a node to a descendant leaf contains same # black nodes. (height)

⇒ Lemma : A red-black tree with n (internal) nodes has height $\leq 2\log_2(n+1) - 2$

Claim : (#internal nodes in a subtree rooted at x) $\geq 2^{\text{height } x} - 1$

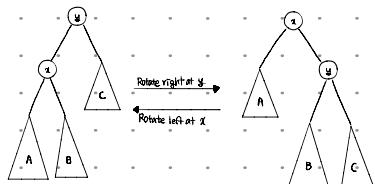
Proof) $n = 0 \Rightarrow 0 = 2^0 - 1 = 1$

$$n+1 \Rightarrow 2^{\text{height } x} \geq \text{black-height} \geq \text{height } x - 1 \quad \text{#nodes} \geq 2^{(\text{height } x) - 1} + 1 = 2^{\text{height } x} - 1 \quad \text{#leaf}$$

$$\rightarrow \text{height } x = \text{height } \text{black}(x) \geq n/2 + 1 \quad (x \neq \text{root})$$

$$n \geq 2^{n/2} + 1 - 1 \rightarrow n \geq 2\log_2(n+1) - 2$$

Rotations



Insertion

Binary Search Tree에서 Insert하는 경우 insert한 노드는 red로 색칠 이후 property를 유지해라

<property>

newly inserted node(x) : root(white), parent(red), grandparent(white)

i) uncle red : parent, uncle black, grandfather red. \rightarrow grandfather

ii) sibling

1) right child : left rotate at parent \rightarrow parent

parent black, grandparent red

right rotate at grandparent

\rightarrow root node

Root black

(p3) initially empty tree, red node

(p4) parent's sibling node

⇒ running time $O(\log n)$

Deletion

Binary Search Tree의 deletion을 진행한 후 property 유지하기

disappearing node : 삭제 노드를 지우고 어떤

[disappearing node : red black : no problem]

disappearing node : black black : one extra black node 더 있으면

X : root node(white), black(white)(white)

1) sibling red : sibling black, parent red, left rotate at the parent

2) sibling black

i) nephew black : sibling red, \rightarrow parent

ii) else

right nephew black : left nephew black, sibling red, right rotate at sibling

Sibling parent의 swap

right nephew black

left rotate at the parent

\rightarrow root

black

⇒ running time : $O(\log n)$

FIBONACCI HEAPS

date :

Fibonacci Heaps

각 노드의 key는 children보다 작음.

각 노드는 children, parent, prev, next를 가지고 있음.

Sibling은 circular doubly linked list로 연결.

Root는 circular doubly linked lists로 연결되어있고, 그중 minimum을 가리키는 조인터가 존재한다.

child를 접근하면 mark를 시켜, parent가 아니라면 child를 더 이상 접근할 수 없다.

account : 방문시간을 저장하고 필요할 때 쓰는 힙

(balance)

potential : lower bound depending only on current state.

$$(\#roots) + 2 (\#\text{marked nodes})$$

Operations

① creating a new heap : root가 empty, list를 initialize

- running time: O(1)

② insert a new value : new root는 O(1)의 minimum check.

- running time: O(1) : potential increase : O(1)

③ querying the minimum

- running time: O(1)

④ uniting two heaps : root를 하나의 list로 만들고 min update

- running time: O(1) : potential = (1st potential) + (2nd potential)

⑤ deleting the minimum : minimum root를 가장 children을 new root로 만든다 → consolidating.

- consolidating : degree를 가진 root이 합드록

- 1) delete → root연결 : running time: O($\alpha(n)$) : potential increase: O($\alpha(m)$)

2) consolidation : running time: potential decrease: paid.

3) build new list : running time: O($\alpha(n)$)

$$\Rightarrow O(\alpha(m))$$

⑥ decrease key : decrease key property: 연결된 모든 root를 만들고 각 노드의 parent가 marked라면 parents의 자식

- 1) decrease value : running time: O(1) : potential increase: O(1)

2) parent losing 2nd child : running time: O(1) : potential decrease: O(1) + O($\alpha(n)$) : node marking: O($\alpha(n)$) : marked nodes: O($\alpha(n)$)

$$\Rightarrow O(1)$$

⑦ delete : - O(1) decrease key 및 deleteMin.

SEGMENT TREE

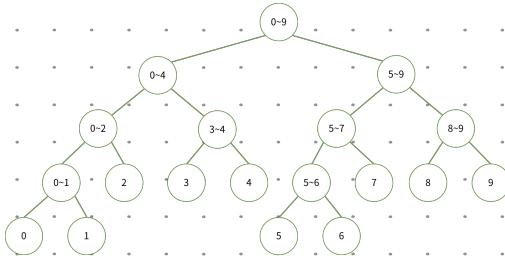
Curve of forgetting study planner

1HOUR 1DAY 2DAY 1WEEK 2WEEK 1MONTH 3MONTH

date :

Segment Tree

배열을 저장하는 도구로서 배열의 값이 바뀔 때 합계를 저장함으로서 합계를 빠르게 구할 수 있게 하는 자료구조.



leaf node : 배열의 수 저장

internal node : left child + right child 값 저장

구현

left child node 2^n

right child node $2^n + 1$

→ 배열 구현 (vector)

✿ initialization : 배열을 모두 받은 후 tree init

저장할 배열 a, 트리 tree, 노드 번호(벡터 인덱스) node, 담당하는 범위 start & end

- 재귀
 + 배열 index

```
// a: 배열 a
// tree: 세그먼트 트리
// node: 세그먼트 트리 노드 번호
// node가 담당하는 합의 범위가 start ~ end
long long init(vector<long long> &a, vector<long long> &tree, int node, int start, int end) {
    if (start == end) {
        return tree[node] = a[start];
    } else {
        return tree[node] = init(a, tree, node*2, start, (start+end)/2) + init(a, tree,
node*2+1, (start+end)/2+1, end);
    }
}
```

✿ calculating sum

- node가 담당하고 있는 구간 [start, end], 합을 구하는 구간 [left, right]

① [left, right]와 [start, end]가 겹치지 않는 경우

② [left, right]가 [start, end]를 완전히 포함하는 경우

③ [start, end]가 [left, right]를 완전히 포함하는 경우

④ [left, right]와 [start, end]가 겹쳐져 있는 경우 (1, 2, 3 차례 나누기 경우)

```
// node가 담당하는 구간이 start~end이고, 구해야하는 합의 범위는 left~right
long long sum(vector<long long> &tree, int node, int start, int end, int left, int right) {
    if (left > end || right < start) {
        return 0;
    }
    if (left <= start && end <= right) {
        return tree[node];
    }
    return sum(tree, node*2, start, (start+end)/2, left, right) + sum(tree, node*2+1, (start+end)/2+1, end, left, right);
}
```

✿ 값 change

```
void update(vector<long long> &tree, int node, int start, int end, int index, long long diff) {
    if (index < start || index > end) return;
    tree[node] = tree[node] + diff;
    if (start != end) {
        update(tree, node*2, start, (start+end)/2, index, diff);
        update(tree, node*2+1, (start+end)/2+1, end, index, diff);
    }
}
```

STANDARD LIBRARY

Curve of forgetting study planner

1HOUR 1DAY 2DAY 1WEEK 2WEEK 1MONTH 3MONTH

date :

1. vector

```
#include <vector>
vector<int> v;
v.push_back(10);
v.push_front(10);
v.size();
v.clear();
v.empty();
v.at(1); v[1];
v.pop_back();
v.pop_front();
v.insert(v.end(), 5); // end가 가리키게 되는 곳에 5의 값을 넣음

v.capacity();
v.reserve(100); // 100개의 용량을 만들어줌
v.shrink_to_fit(19);
v.resize(10);

vector<int> v1(5); // 0을 5개 초기화
vector<int> v2(5, 2); // 2를 5개 초기화
```

2. iterator

```
vector<int>::iterator p;
p = v.begin();
p != v.end();
*p // iterator의 내부값
++, --로 iterator가 가리키는 칸을 옮김

const_iterator // const인 iterator
    v.cbegin();
    v.cend();

reverse_iterator // ++을 하면 반대방향으로 움직이는 iterator
    v.rbegin();
    v.rend();

const_reverse_iterator
    v.crbegin();
    v.crend();
```

3. list

```
#include <list> // doubly linked list만 있음 (singly linked list X)
list<int> List;
List.push_back(3);
List.push_front(3);
List.size();

list<int>::iterator iter;
```

4. deque : list + vector / 랜덤 액세스, push_front, pop_front, vector의 기능 모두 가능

```
#include <deque>
deque<int> dq;
```

5. stack

```
#include <stack>
stack<int> Stack;
Stack.push(10);
Stack.pop();
Stack.size();
Stack.empty();
Stack.top();
```

6. queue

```
#include <queue>
queue<int> Queue;
Queue.front();
Queue.back();
Queue.push(4);
Queue.pop();
Queue.empty();
```

priority_queue

```
priority_queue<int> PQueue;
push하고 pop을 하면 큰 숫자부터 나옴 -> heapSort
// 작은 숫자부터 나오게 하려면?
#include <functional>
priority_queue<int, vector<int>, greater<int> > PQueue; // 어렵네
// top에 있는 값만 정렬되어 있음
```

7. set

```
#include <set>
set<int> s;
s.insert(10); // 중복된 값은 2번들어가지 않음
s.find(10); // find할 대상을 찾은 경우 그 값을 가리키는 iterator가, 없으면 end()위치의 iterator를 리턴
s.end()
s.erase(s.find(1)); // 1이 없는 경우 end()를 지우려고 하므로 문제가 될 수 있음
s.erase(1);
// 정렬되어 저장되어있음
```

```
#include <cctype>
s.count(1); // 1이 존재하면 true, 아니면 false 리턴
```

```
// iterator 대신에 순회를 쉽게 하는 방법
for(int current:s){cout << current << endl;}
for(auto current:s){cout << current << endl;}
```

8. multiset

```
#include <set>
multiset<int> s;
s.erase(3); // 3의 중복된 값 모두 삭제
s.erase(find(3)); // 해당 iterator한개만 삭제 (숫자 하나만 삭제됨)
s.count(3); // 3의 갯수 출력
```

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 넣어라

7) fill(범위의 시작점, 끝점, 채울 값)
 ex) fill(a+2, a+5, 0) : a+2부터 a+5까지를 0으로 채워 넣어라

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 하라는대로 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 true, 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){

other.math.other.english;
} // 총합이 낮은 학생부터 정렬

date :

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, 두번재 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, 두번재 범위의 시작, 결과를 저장할 범위의 시작점)` : 합집합 연산

32) `set_intersection(첫번재 범위의 시작, 끝, 두번재 범위의 시작, 결과를 저장할 범위의 시작점)` : 교집합 연산

33) `set_difference(첫번재 범위의 시작, 끝, 두번재 범위의 시작, 결과를 저장할 범위의 시작점)` : 차집합 연산

34) `set_symmetric_difference(첫번재 범위의 시작, 끝, 두번재 범위의 시작, 결과를 저장할 범위의 시작점)` : 대칭 차집합 연산

www.cplusplus.com

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());
    }
}
```

date :

```

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->rkhf = 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;
}

```

date :

```

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> ans;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (gra[m * i + j].size() == 1) {
                ans.push_back(gra[m * i + j].back());
            }
        }
    }
    return ans;
}

int graph::doDFS(vector<int>& visited, int point, int& space) {
    visited[point] = 1;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (gra[m * i + j].size() == 1) {
                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)); }
            }
        }
    }
    return space;
}

```

date :

```

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;
}

```