2017 SCSC Workshop

Sogang univ.
System modeling & Optimization Lab
Sangdurk Han

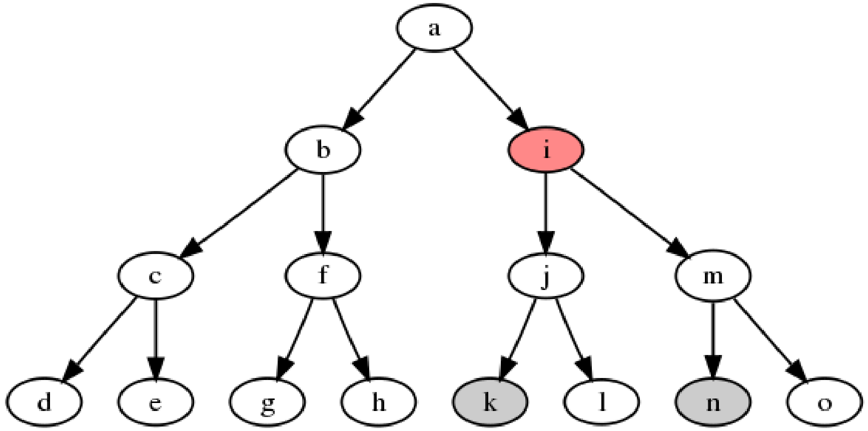
01 Data Structure & STL

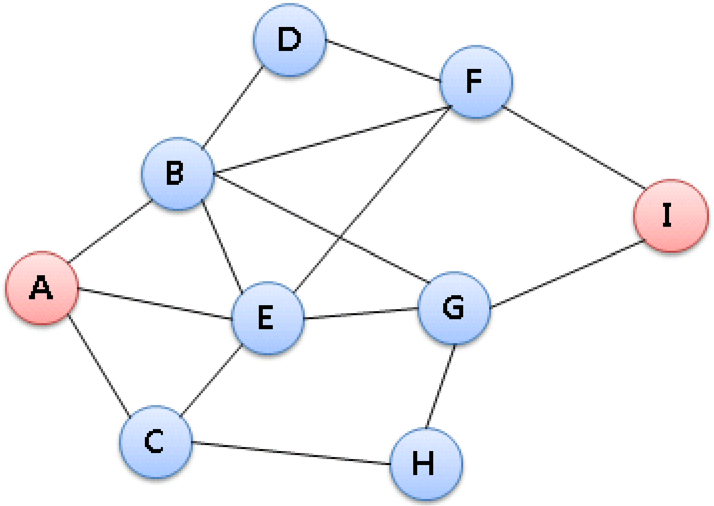
Data Structure & STL

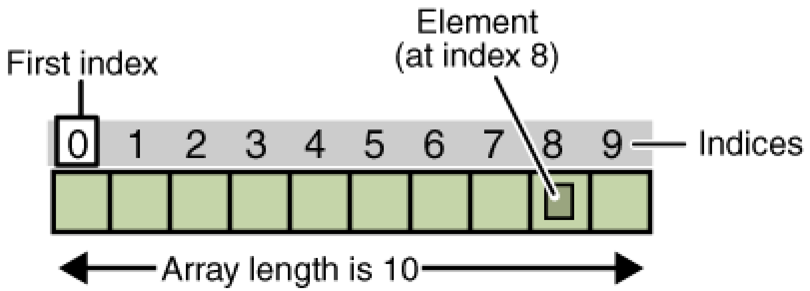
Data

Data를 표현하는 방법

배열, 트리, 그래프

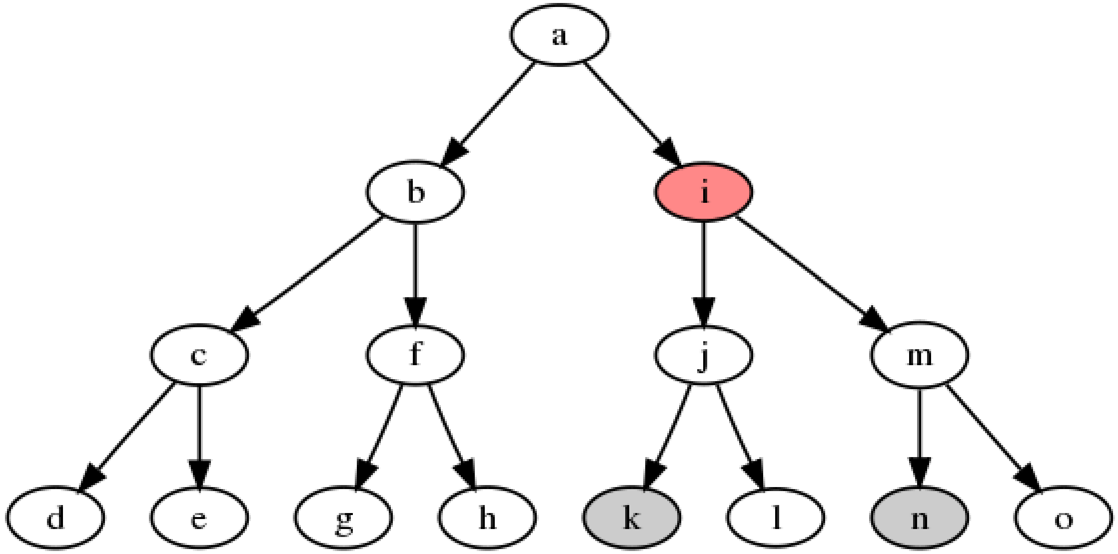






Definition

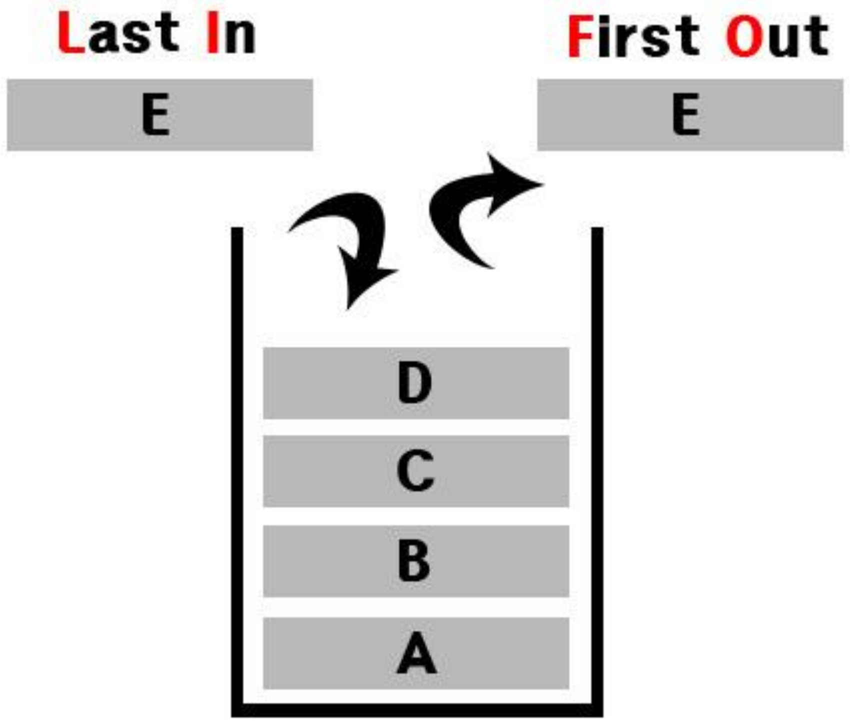
자료구조(Data Structure)는 자료(Data)를 효율적으로 이용할 수 있도록 저장할 수 있는 방법



Data Structure & STL

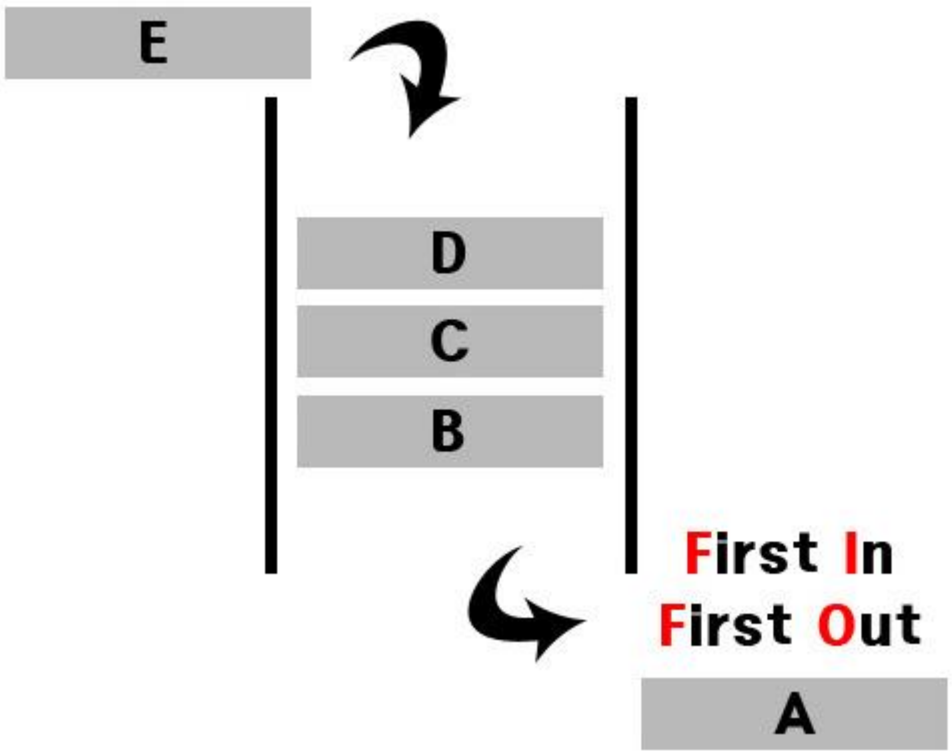
Stack(DFS)

Last In First Out 방식

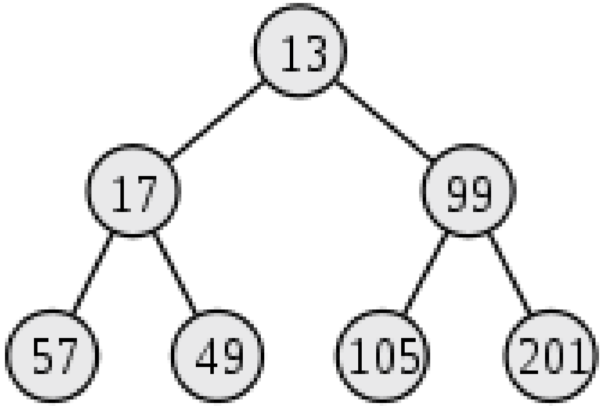


Queue(BFS)

First In First Out 방식



Heap(prioirity_queue)



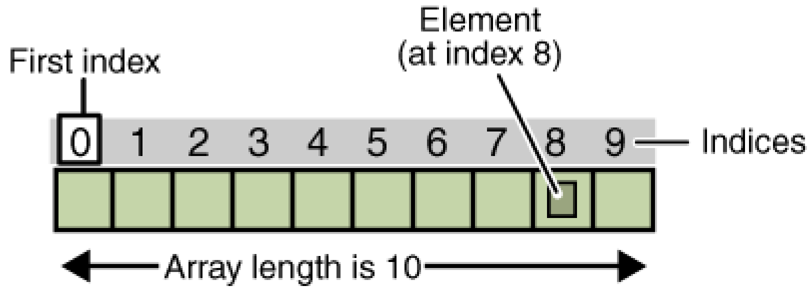
가장 최솟값이나 최댓값을 제공

추가 $\boldsymbol{O(logN)}$, 삭제 $\boldsymbol{O(logN)}$

주로 그래프에서 최단 경로를 구하는 다익스트라 알고리즘에서 이용되거나 그리디 알고리즘에서 이용된다.

컨테이너 (vector)

배열과 거의 동일



추가 $O(1)$, 접근 $O(1)$, 삭제 $O(N)$

삭제는 거의 없고 추가만 하는 경우에 많이 사용됨
대부분 배열 대체용으로 많이 씀
(메모리가 필요한 부분만큼만 사용하기 때문에)

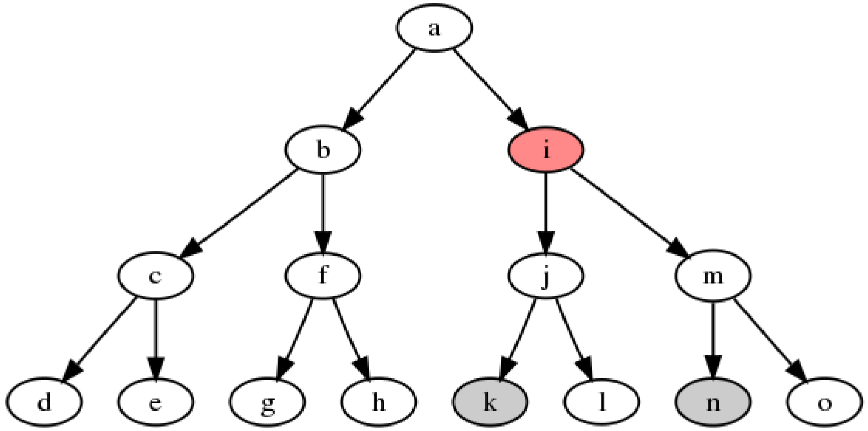
용도 – 그래프 정점, 간선 저장

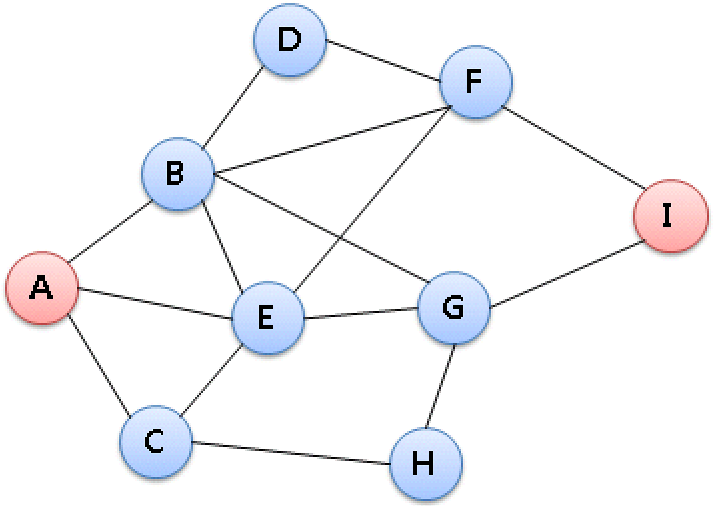
Data Structure & STL

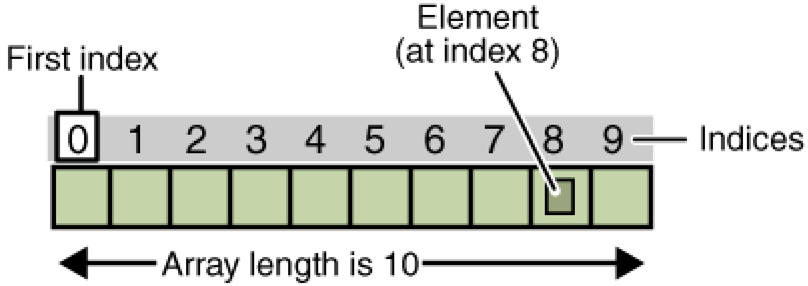
Data

Data를 표현하는 방법

배열, 트리, 그래프

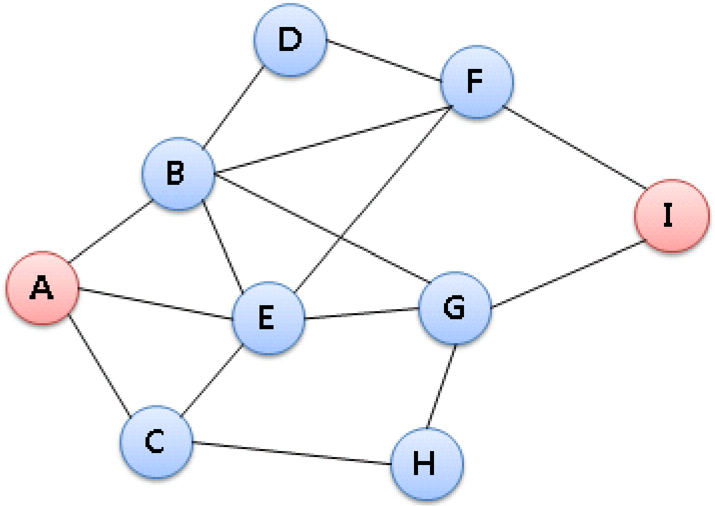






Data

Data를 표현하는 방법



그래프

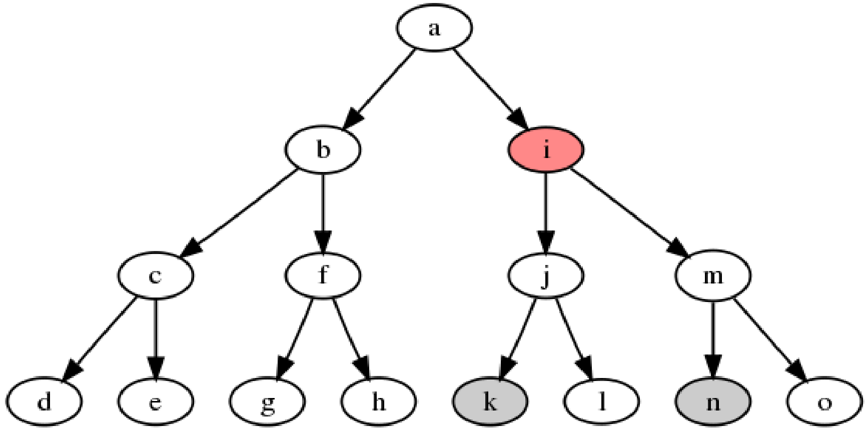
정점(vertex) - A, B, C, ..., I
간선(edge) – (A-B), (A-C), (A-E), ..., (F-I)

ex) 여러 도시들을 연결하는 도로망,
사람들 간의 지인 관계, 웹사이트 간의 링크 관계

Data

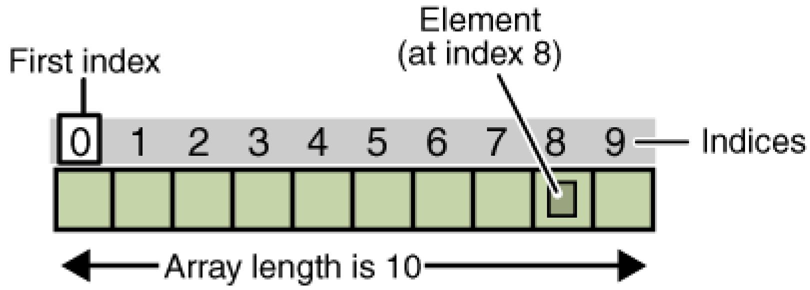
Data를 표현하는 방법

트리



1. Cycle이 없는 그래프
2. 모든 임의의 정점에서 다른 정점으로 이동 가능
3. N개의 정점과 총 N-1개의 간선을 가짐

컨테이너 (vector)



배열과 거의 동일

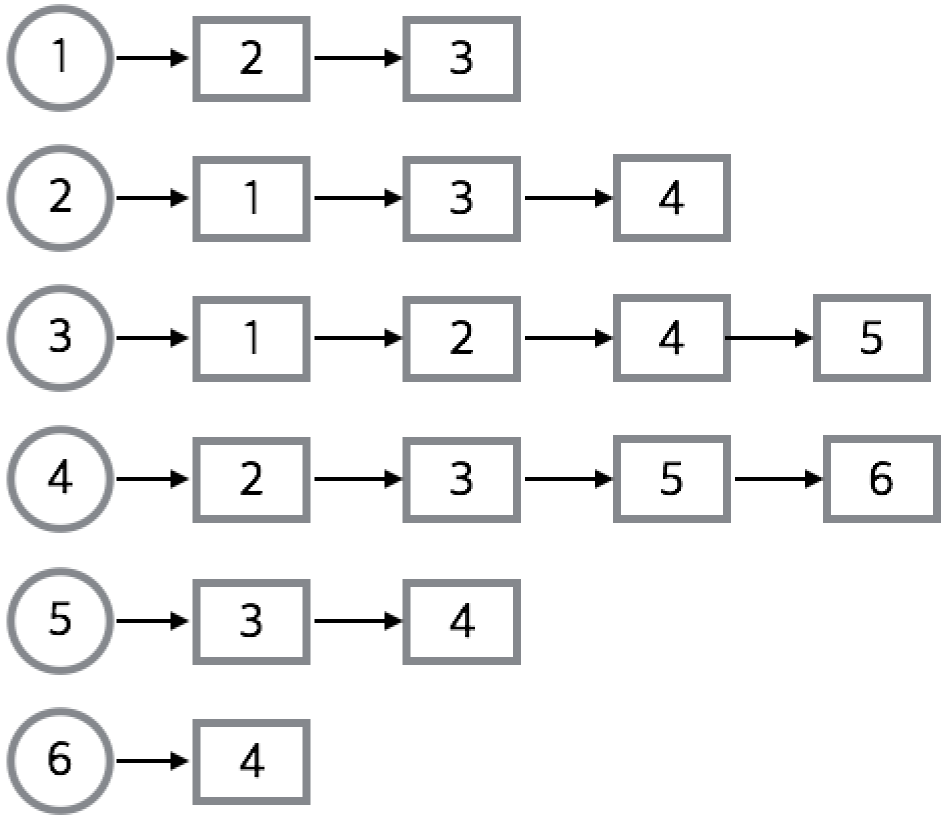
추가 $O(1)$, 접근 $O(1)$, 삭제 $O(N)$

삭제는 거의 없고 추가만 하는 경우에 많이 사용됨
대부분 배열 대체용으로 많이 씀
(메모리가 필요한 부분만큼만 사용하기 때문에)

용도 – 그래프 정점, 간선 저장

컨테이너 (vector)

용도 - 그래프 정점, 간선 저장




```cpp
#include <vector>

using namespace std;

const int N = 1e5 + 5;
vector <int> e[N];

int main() {
    int n, m;
    scanf("%d %d", &n, &m); // 정점, 간선

    for (int i = 0; i < m; i++) {
        int a, b;
        scanf("%d %d", &a, &b); // a to b
        e[a].push_back(b);
        //e[b].push_back(a); //양방향일 경우
    }
}
```

컨테이너 (vector)

접근

```cpp
void go(int cur) {
    /* cur과 인접한 정점 모두 탐색 */
    for (int i = 0; i < e[cur].size(); i++) {
        int nxt = e[cur][i];
        /* your code */
    }
}
```

vector <자료형(ex: int, double...> v;

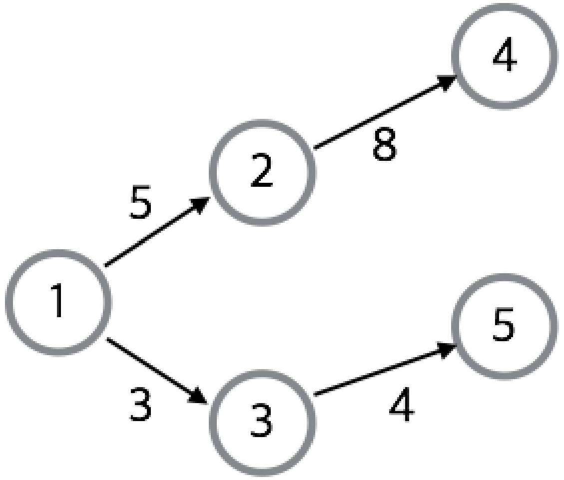
vector에 넣을 자료의 형태 지정

Data Structure & STL

Data

그래프
(가중치)

→



그럴 경우 vector에 저장할 데이터가
(자신이 가리키고 있는 정점, 가중치) 2가지를 저장해야 됨

Pair

사용자 지정 자료형
int, double, float, ... 등은
C언어가 기본적으로 제공하는 자료형

우리는 (int, int)의 쌍의 정보를 저장하고 싶음
그럴 때 사용하는 STL이 pair

Pair (Algorithm 헤더)

```
typedef pair <int, int> ii; //(int, int)
typedef pair <double, int> di; //(double, int)
typedef pair <ii, di> pp; //(ii, di)
const int N = 55;
vector <ii> e[N];

int main() {
    ii a = ii(1, 3);
    di b = di(3.5, 3);
    pp c = pp(a, di(3.4, 4));
    int n, m;
    scanf("%d %d", &n, &m);

    for (int i = 0; i < m; i++) {
        int a, b, c;
        scanf("%d %d %d", &a, &b, &c); //a에서 b로 c만큼의 거리
        e[a].push_back(ii(b, c)); // 목적지 b와 거리 c 저장
    }
}
```

Pair

장점 – 이후 sort함수의 compare 함수를 구현할 필요 없음

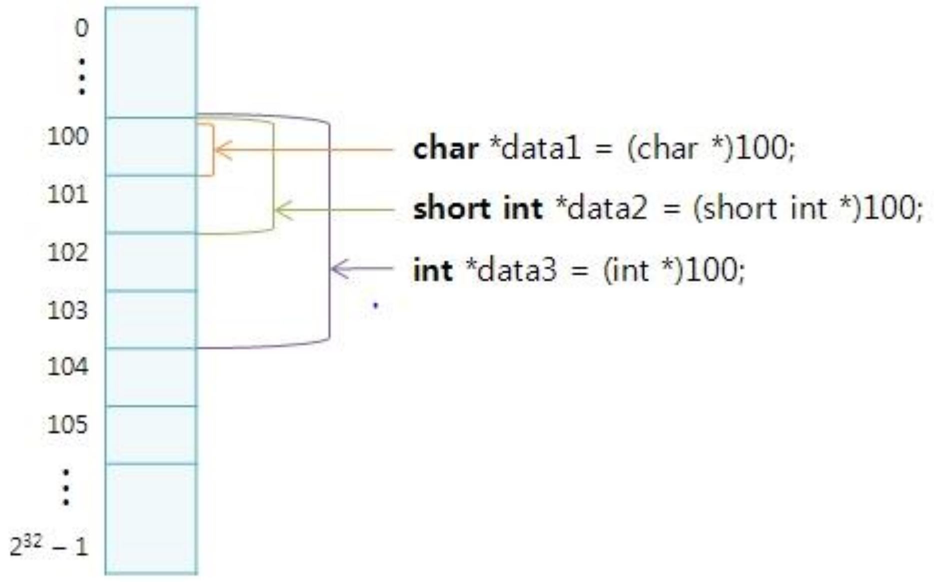
Example

```
ii a = ii(1, 3), ii b = ii(2, 2);
if( a < b ) puts("OK");

ii a = ii( 1, 3), ii b = ii( 1, 4);
if( a < b ) puts("OK");
```

Iterator

STL의 위치를 가리키는 포인터
(주소값 뿐만이 아니라 그 이외의 정보를 추가적으로 가지고 있음)



Example

vector <int> v;
v.begin(); // v의 시작 위치의 주소값
v.end(); // v의 끝 위치 다음 주소값

Iterator

```
v.size() = (int)(v.end() – v.end())

(*(v.begin())) // = v[0];
(*(++v.begin())) // = v[1];
(*(--v.end()) // = v[v.size()-1];

vector <ii>::iterator it = v.begin();
(*it) // = v[0];
it++; (*it) // = v[1];
```

Algorithm

sort()함수 : 배열, vector를 정렬

```
sort( v.begin(), v.end() ); // vector일 경우
sort( v, v+n ) // 배열일 경우
```

a~b인덱스 정렬 (ex 0~n-1)

```
sort( v.begin()+a, v.end()+b+1); // vector일 경우
sort( v+a, v+b+1 ); // 배열일 경우
```

Algorithm

sort()함수 : 배열, vector를 정렬

비교 연산이 일반적인 비교 방식과 다를 경우

```
bool comp(int a, int b) { // a가 b보다 작은가?
    if (a < 3) return 1;
    else if (b> 4) return 1;
    else return 0;
}
vector <int> v;

int main() {
    sort(v.begin(), v.end(), comp);
}
```

Algorithm

sort()함수 : 배열, vector를 정렬

비교 연산이 일반적인 비교 방식과 다를 경우

```
bool comp(int a, int b) { // a가 b보다 작은가?
    if (a < 3) return 1;
    else if (b> 4) return 1;
    else return 0;
}
vector <int> v;

int main() {
    sort(v.begin(), v.end(), comp);
}
```

Algorithm

abs(자료형)함수 : 절대값을 return해주는 함수

max(자료형, 자료형) 함수 : 최댓값
min(자료형, 자료형) 함수 : 최솟값

모두 pair도 됨

Algorithm

binary_search() : 함수 (O(logN))

```
#include <algorithm>
#include <vector>

using namespace std;

vector <int> e;

int main() {

    e = { 1, 2, 5, 4, 3};

    sort(e.begin(), e.end());

    binary_search(e.begin(), e.end(), 6); // 0 return
    binary_search(e.begin(), e.end(), 5); // 1 return
    /* 찾고자하는 자료가 있을 경우 1 return*/
}
```

Algorithm

lower_bound() : 함수 (O(logN))

```
vector <int> e;

int main() {

    e = { 1, 1, 2, 5, 6};

    sort(e.begin(), e.end());

    lower_bound(e.begin(), e.end(), 1);
    /* 1보다 크거나 같은 수 중 가장 앞의 iterator를 return */
    /* e = { 1 <- 여기를 가리킴 , 1, 2, 5, 6}; */
    lower_bound(e.begin(), e.end(), 3);
    /* 3보다 크거나 같은 수 중 가장 앞의 iterator를 return */
    /* e = { 1, 1, 2, 5  <- 여기를 가리킴 , 6}; */
    binary_search(e.begin(), e.end(), 10);
    /* 찾고자하는 수가 없을 경우 v.end()를 return */
}
```

Algorithm

upper_bound() : 함수 (O(logN))

```
vector <int> e;

int main() {

    e = { 1, 1, 2, 5, 6};

    sort(e.begin(), e.end());

    upper_bound(e.begin(), e.end(), 1);
    /* 1보다 큰 수 중 가장 앞의 iterator를 return */
    /* e = { 1, 1, 2 <- 여기를 가리킴, 5, 6}; */
    upper_bound(e.begin(), e.end(), 3);
    /* 3보다 큰 수 중 가장 앞의 iterator를 return */
    /* e = { 1, 1, 2, 5  <- 여기를 가리킴 , 6}; */
    upper_bound(e.begin(), e.end(), 10);
    /* 찾고자하는 수가 없을 경우 v.end()를 return */
}
```

Map & Set

Iterator를 통해
map, set 전체 탐색

```
map <int, int> mp;
set <int> s;

int main() {
    for (map<int, int>::iterator it = mp.begin(); it != mp.end(); it++) {
        // (it->first) : map의 id
        // (it->second) : map의 id에 해당하는 값
    }

    for (set<int>::iterator it = s.begin(); it != s.end(); it++) {
        // (*it) 해당하는 set의 값
    }
    // 주의 set, map에 insert를 할 경우 iterator가 변경 됨
}
```

02 Dynamic programing

Dynamic Programing

Definition

복잡한 문제를 간단한 여러 개의 문제로 나누어 푸는 방법을 말한다.
이것은 부분 문제 반복과 최적 기본 구조를 가지고 있는 알고리즘을 일반적인 방법에 비해 더욱 적은 시간 내에 풀 때 사용한다.

기본적으로 수학적 귀납법과 크게 다르지 않다.

정의 - $f[i]$ = i번 째 피보나치 수
초기화 – $f[0] = 1$
수식 - $f[i] = f[i-1] + f[i-2]$

Dynamic Programing

Notice

기본적으로 수학적 귀납법과 크게 다르지 않다.

정의 - f[i] = i번 째 피보나치 수
초기화 – f[0] = 1
수식 - f[i] = f[i-1] + f[i-2]

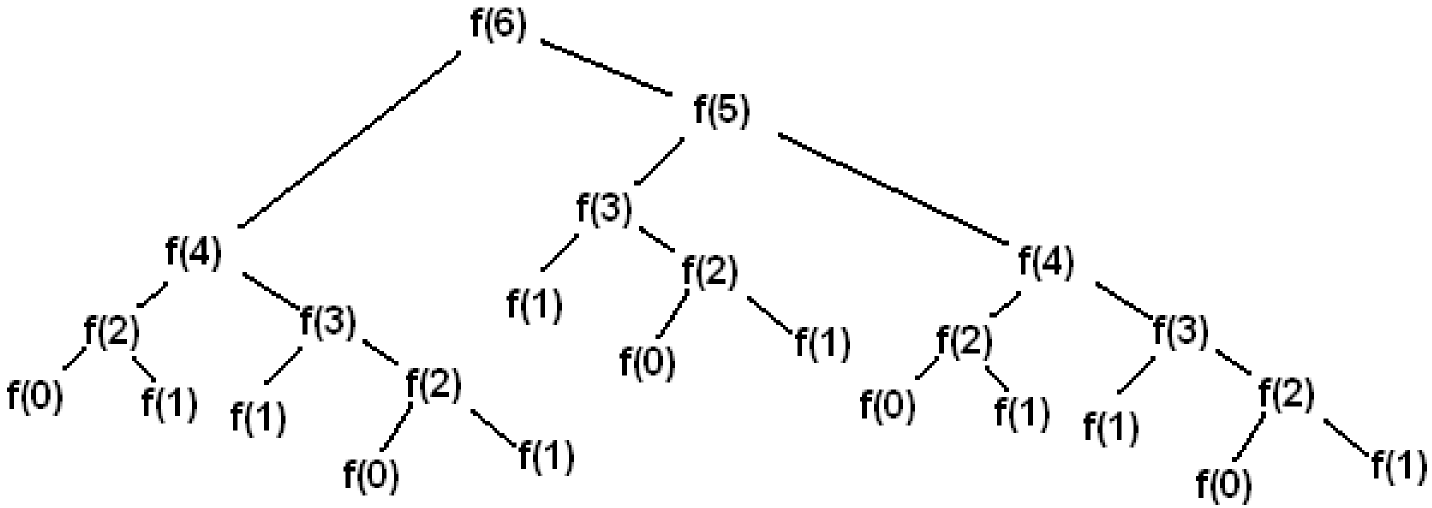
그렇기 때문에 수식을 유도하는 방법은 초기값이 성립하는 것과
K번째 까지 성립할 때 K+1이 성립함을 초기화와 수식 단계에서
이루어지기 때문에 증명이 필요가 없음

Dynamic Programing

피보나치 수

```c
int fibo( int n ){
    if( n <= 1 ) return n;
    else return fibo( n-1 ) + fibo( n-2 );
}

int main(){
    int n;
    scanf("%d", &n);
    printf("%d₩n", fibo( n ) );
}
```



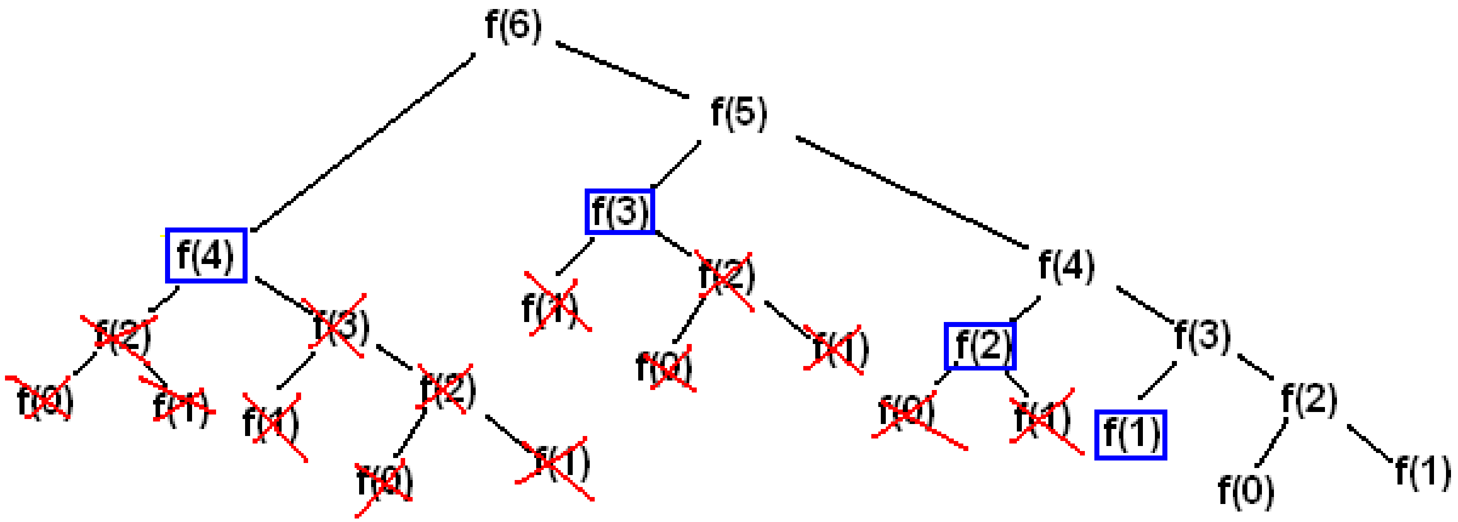
Dynamic Programing

피보나치 수

```c
int f[N];

int fibo( int n ){
    if( n <= 1 ) return n;
    else if( f[n] > 0 ) return f[n];
    else return f[n] = fibo( n-1 ) + fibo( n-2 );
}

int main(){
    int n;
    scanf("%d", &n);
    printf("%d\n", fibo( n ) );
}
```



포도주 시식

1. 포도주 잔을 선택하면 그 잔에 들어있는 포도주는
2. 모두 마셔야 하고, 마신 후에는 원래 위치에 다시 놓아야 한다.
3. 연속으로 놓여 있는 3잔을 모두 마실 수는 없다.

목표 시식하는 포도주의 양을 최대화

1 <= n <= 10000

포도주 시식

정의 d[i][3]

d[i][0] = i번째 포도주를 마시지 않을 경우
d[i][1] = i번째 포도주를 시식하고 연속으로 1잔의 포도주를 시식한 상태
d[i][2] = i번째 포도주를 시식하고 연속으로 2잔의 포도주를 시식한 상태

Dynamic Programing

포도주 시식

초기화

d[1][0]=0
d[1][1]=a[0]
d[1][2]=0

Dynamic Programing

포도주 시식

Dp식

```
d[i][0]=max(d[i-1][0], d[i-1][1], d[i-1][2]); // 마시지 않는 경우
d[i][1]=d[i-1][0]+a[i]; // 마시는 경우
d[i][2]=d[i-1][1]+a[i]; // 마시는 경우
```

시간복잡도

$O(cn) = O(n)$

Dynamic Programing

포도주 시식

코드

```
#include <cstdio>
#include <algorithm>
using namespace std;
int a[10001], d[10001][3];

int main(){
    int n;
    scanf("%d", &n);
    for(int i=0; i<n; i++)
        scanf("%d", a+i);
    d[0][1]=a[0];
    for(int i=1; i<n; i++){
        d[i][0]=max(d[i-1][0], max(d[i-1][1], d[i-1][2]));
        d[i][1]=d[i-1][0]+a[i];
        d[i][2]=d[i-1][1]+a[i];
    }
    printf("%d", max(d[n-1][0], max(d[n-1][1], d[n-1][2])));
    return 0;
}
```

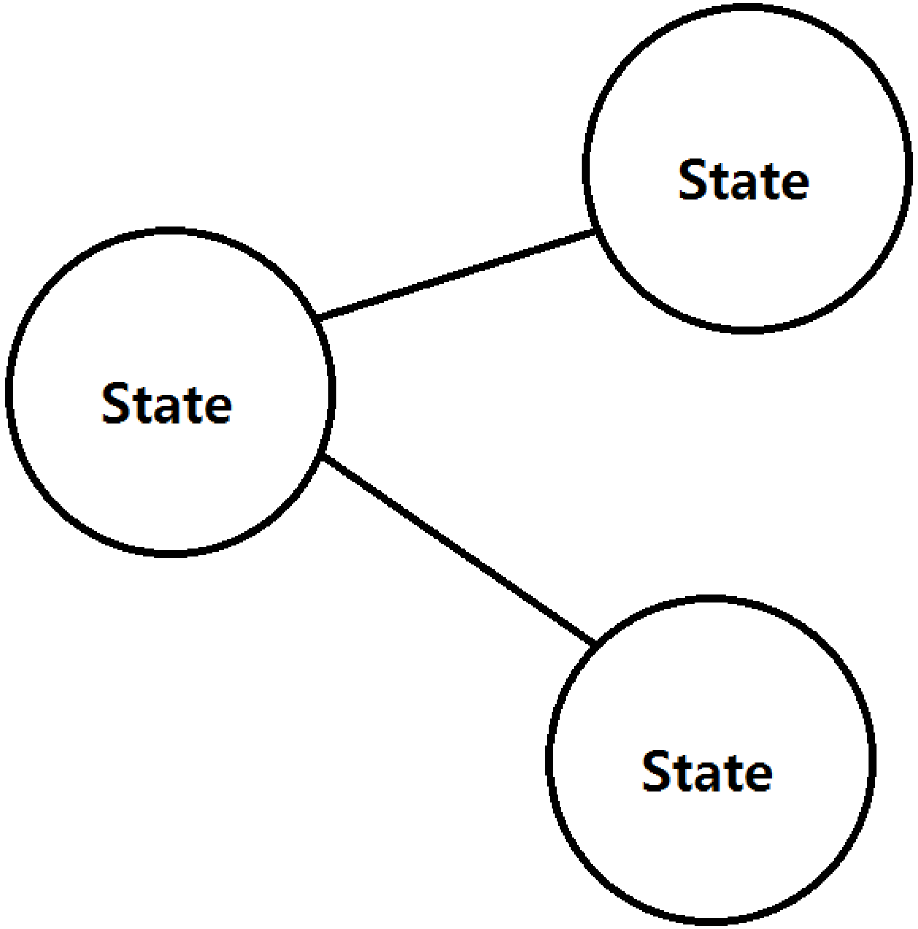
Dynamic Programing

Notice

결국은 어떤 상태 집합을 정점

이전 상태에서 다음 상태로 넘어가는 것을 간선

결국에는 중복되는 경로 들의 결과값을 배열에 저장하여 중복 된 경로를 방문하는 것을 방지

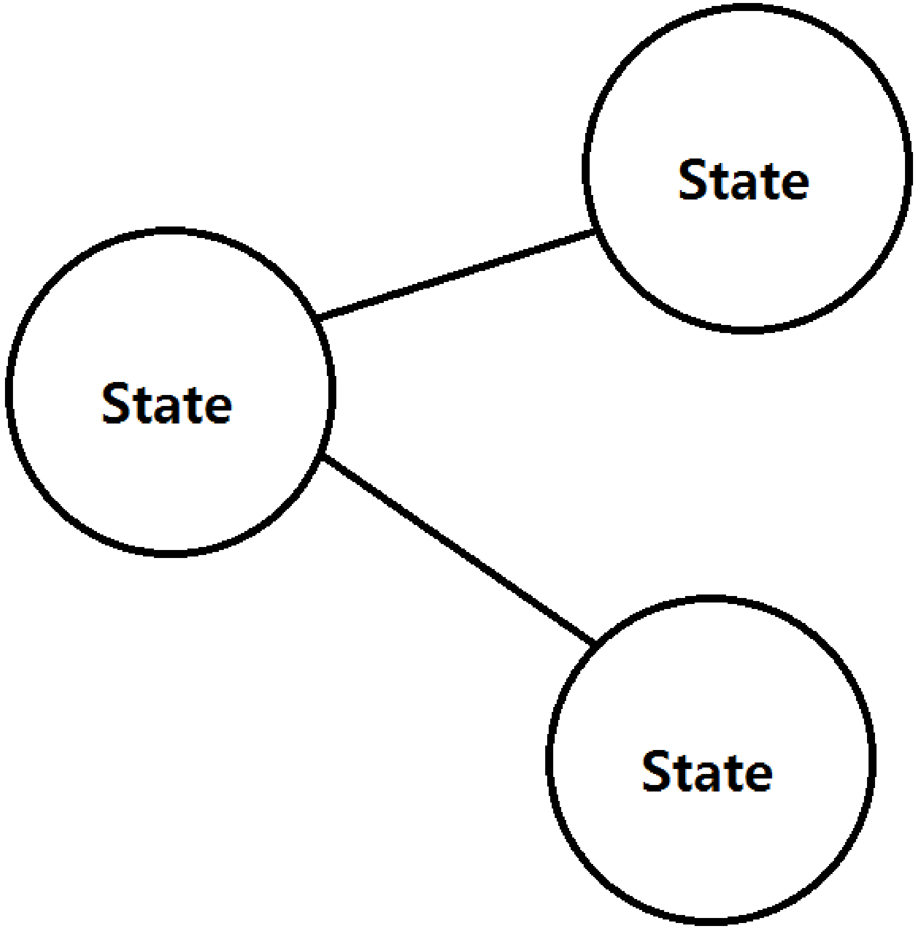


Dynamic Programing

Notice

이것을 응용하면

```
int go( prev state 정보 ){
        /* 연산 */
        prev state로 부터
                cur state 정보 획득
        return go( cur state 정보 )
}
```



8퍼즐

3*3 표에 다음과 같이 수가 채워져 있다. 오른쪽 아래 가장 끝 칸은 비어 있는 칸이다.

1	2	3
4	5	6
7	8	

어떤 수와 인접해 있는 네 개의 칸 중에 하나가 비어 있으면, 수를 그 칸으로 이동시킬 수가 있다. 물론 표 바깥으로 나가는 경우는 불가능하다. 우리의 목표는 초기 상태가 주어졌을 때, 최소의 이동으로 위와 같은 정리된 상태를 만드는 것이다. 다음의 예를 보자.

Dynamic Programing

8퍼즐

1		3
4	2	5
7	8	6

state 0

1	2	3
4		5
7	8	6

state 1

1	2	3
4	5	
7	8	6

state 2

1	2	3
4	5	6
7	8	

state 3

가장 윗 상태에서 세 번의 이동을 통해 정리된 상태를 만들 수 있다. 이와 같이 최소 이동 횟수를 구하는 프로그램을 작성하시오.

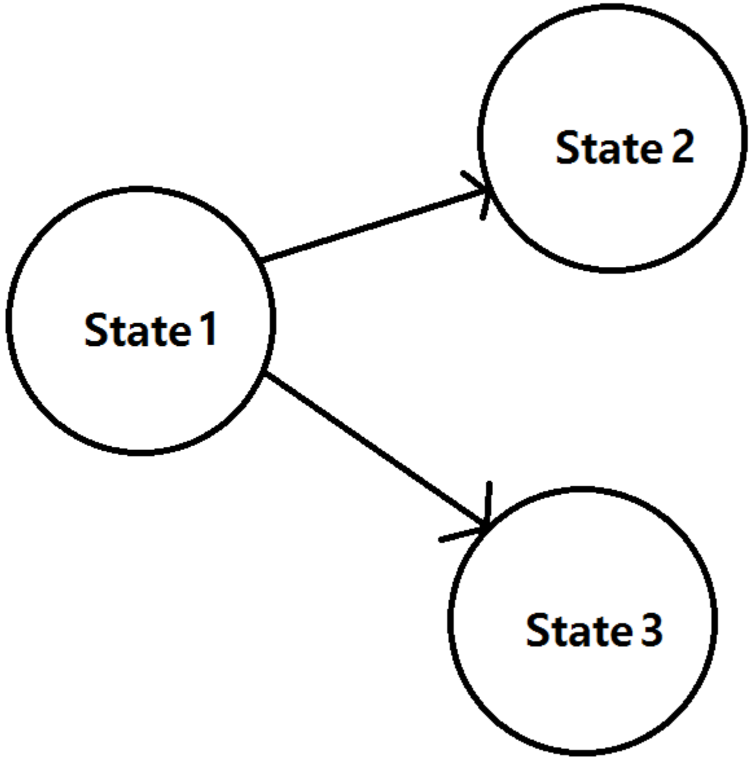
Dynamic Programing

8퍼즐

State로 표현하면
State 0에서 목적지 state까지
가야하는 최단 경로를 구하면 됨

중복되는 경로는 이미 방문한 경로
이므로 방문하지 않음

D[state i] = state0에서 state i까지 가는
최단 경로



Longest Common Subsequence (LCS)

문자열 s, t
s[0], … , s[n-1]
t[1], … , t[n-1]

공통 부분 문자열 길이의 최대값

1<= n, m <= 1000

Dynamic Programing

Longest Common Subsequence (LCS)

정의

D[i][j] = S문자열은 i번째 까지 선택했고
T문자열은 j번째 까지 선택

초기화

d[i][j] = 0

1 <= i <= n

1 <= j <= m

Longest Common Subsequence (LCS)

Dp식

```
if(S[i]==T[i])
  d[i][j] = d[i-1][j-1] + 1
If(S[i]!=T[i])
  d[i][j] = max(d[i-1][j], d[i][j-1])
```

Longest Common Subsequence (LCS)

```
int n, m;
cahr s[MAx_N], t[MAX_M];

int d[MAX_N+1][MAX_M+1];

for(int i=0; i<n; i++){
    for(int j=0; j<m; j++){
        if(s[i]==t[i])
            d[i+1][j+1] = d[i][j] +1;
        else
            d[i+1][j+1] = max( d[i][j+1], d[i+1][j] );
    }
}

printf("%d\n", d[n][m]);
```

Dynamic Programing

Longest Increasing Subsequence. (LIS)

길이 n 수열 a0,a1, ..., an-1
증가 부분열 중 최장의 것의 길이

1 <= n <= 10^3
1 <= ai <= 10^6

10 20 40 30 70 50 60
10 20 40 30 70 50 60

최대 길이는 5

Dynamic Programing

Longest Increasing Subsequence. (LIS)

정의

D[i] = i번째까지 선택했을 때 최대부분
증가수열 길이

초기화

D[i] = 1
1 <= i <= n

Dp식

D[i] = d[j] + 1
1 <= j < I

Longest Increasing Subsequence. (LIS)

```
for(int i=0; i<n; i++)
    d[i]=1;
for(int i=0; i<n; i++)
    for(int j=0; j<i; j++)
        if(a[j]<a[i])
            d[i]=max(d[i], d[j]+1);
int ans=0;
for(int i=0; i<n; i++)
    ans=max(ans, d[i]);
```

Longest Increasing Subsequence. (LIS)

다른방법
O(NlogN), Binary_search를 이용

```
fill(d, d+n, INF);
for(int i=0; i<n; i++)
    (*lower_bound(d, d+n, a[i])) = a[i];

printf("%d\n", n-(int)(lower_bound(d, d+n, INF)-d));
```

Longest Increasing Subsequence. (LIS)

다른방법

lower_bound()
정렬된 배열로부터 이진탐색에 의해 특정값이 들어 갈 수 있는 가장 작은 위치의 주소를 반환함.

```
fill(d, d+n, INF);
for(int i=0; i<n; i++)
    (*lower_bound(d, d+n, a[i])) = a[i];

printf("%d\n", n-(int)(lower_bound(d, d+n, INF)-d));
```

Longest Increasing Subsequence. (LIS)

다른방법

lower_bound()
정렬된 배열로부터 이진탐색에 의해 특정값이 들어 갈 수 있는 가장 작은 위치의 주소를 반환함.

```
fill(d, d+n, INF);
for(int i=0; i<n; i++)
    (*lower_bound(d, d+n, a[i])) = a[i];

printf("%d\n", n-(int)(lower_bound(d, d+n, INF)-d));
```

Table

- **10 20 40 30 70 50 60**
- **10 20 40 30 70 50 60**

i	0	1	2	3	4	5	6
D[i]	INF	INF	INF	INF	INF	INF	INF

Table

- **10 20 40 30 70 50 60**
- **10 20 40 30 70 50 60**

i	0	1	2	3	4	5	6
D[i]	10	INF	INF	INF	INF	INF	INF

Table

- **10 20 40 30 70 50 60**
- **10 20 40 30 70 50 60**

i	0	1	2	3	4	5	6
D[i]	10	20	INF	INF	INF	INF	INF

Table

- **10 20 40 30 70 50 60**
- **10 20 40 30 70 50 60**

i	0	1	2	3	4	5	6
D[i]	10	20	40	INF	INF	INF	INF

Table

- **10 20 40 30 70 50 60**
- **10 20 40 30 70 50 60**

i	0	1	2	3	4	5	6
D[i]	10	20	30	INF	INF	INF	INF

Table

- **10 20 40 30 70 50 60**
- **10 20 40 30 70 50 60**

i	0	1	2	3	4	5	6
D[i]	10	20	30	70	INF	INF	INF

Table

- **10 20 40 30 70 50 60**
- **10 20 40 30 70 50 60**

i	0	1	2	3	4	5	6
D[i]	10	20	30	50	INF	INF	INF

Table

- **10 20 40 30 70 50 60**
- **10 20 40 30 70 50 60**

i	0	1	2	3	4	5	6
D[i]	10	20	30	50	60	INF	INF

Table

- **10 20 40 30 70 50 60**
- **10 20 40 30 70 50 60**

i	0	1	2	3	4	5	6
D[i]	10	20	30	50	60	INF	INF

- (int)((lower_bound(d, d+n, INF)-d) => 5

풀어 볼만한 예선 문제

https://www.acmicpc.net/problem/11066
https://www.acmicpc.net/problem/11062
https://www.acmicpc.net/problem/10251
https://www.acmicpc.net/problem/9015
https://www.acmicpc.net/problem/9011
https://www.acmicpc.net/problem/9012
https://www.acmicpc.net/problem/9009
https://www.acmicpc.net/problem/7662
https://www.acmicpc.net/problem/3758
https://www.acmicpc.net/problem/9007

감사합니다.