# 알고리즘과 코딩테스트 준비

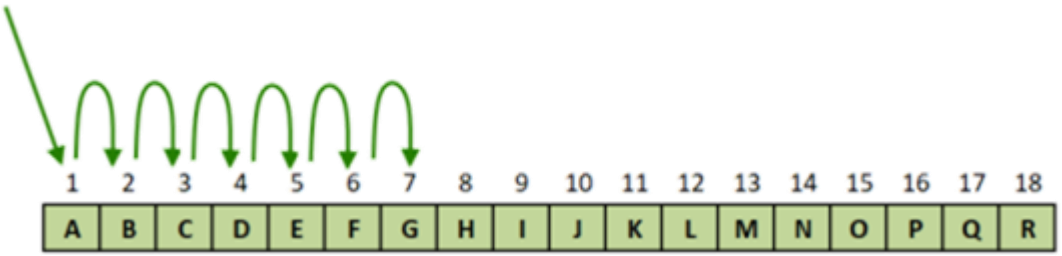
2022/1/13~1/19

윤형기 (hky@openwith.net)

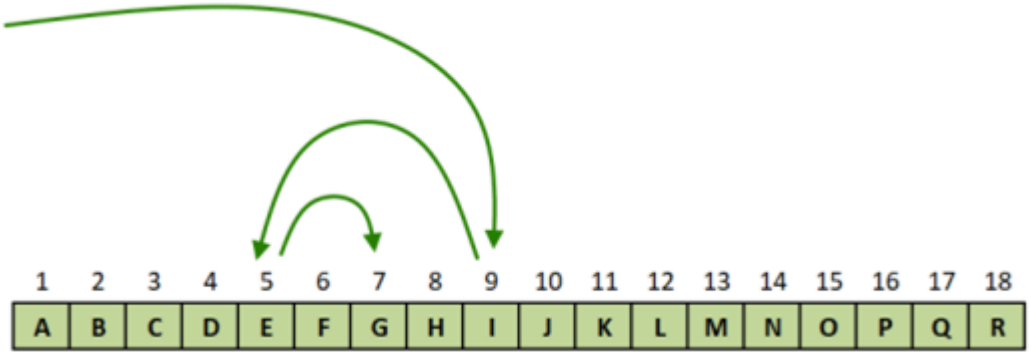
# CHAPTER 07 이진탐색

# 개요

- Linear search vs. Binary Search
  - linear search = sequential search




  - 이진탐색

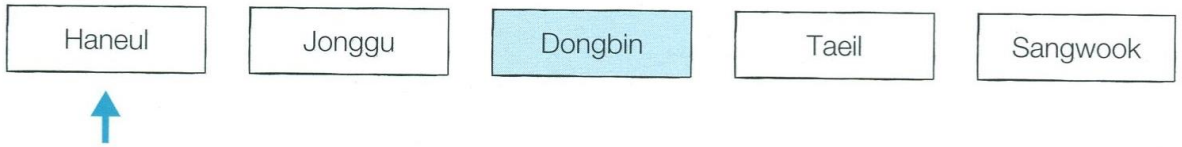


# 1 범위를 반씩 좁혀가는 탐색

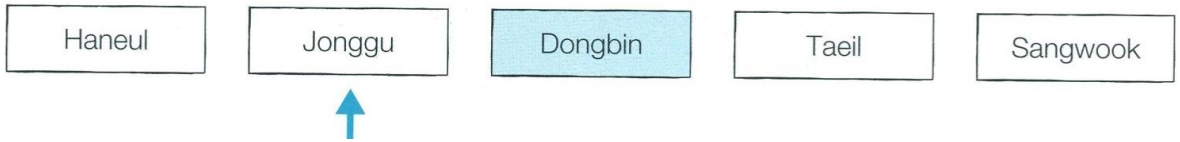
- 순차 탐색 (Sequential Search)

step 0 초기 단계

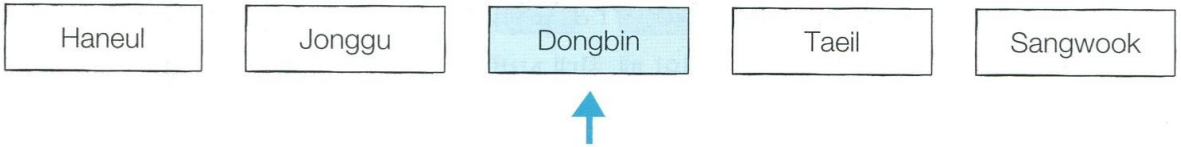

step 1 가장 먼저 첫 번째 데이터를 확인한다. Haneul은 찾고자 하는 문자열과 같지않다. 따라서 다음 데이터로 이동한다.




step 2 두 번째 데이터를 확인한다. Jonggu는 찾고자 하는 문자열과 같지 않다. 다음 데이터로 이동한다.




step 3 세 번째 데이터를 확인한다. Dongbin은 찾고자 하는 문자열과 같으므로 탐색을 마친다.



- 7-1.py 순차탐색 소스코드

```
# 순차 탐색 소스코드 구현
def sequential_search(n, target, array):
    # 각 원소를 하나씩 확인하며
    for i in range(n):
        # 현재의 원소가 찾고자 하는 원소와 동일한 경우
        if array[i] == target:
            return i + 1 # 현재의 위치 반환 (인덱스는 0부터 시작하므로 1 더하기)
    return -1 # 원소를 찾지 못한 경우 -1 반환

print("생성할 원소 개수를 입력한 다음 한 칸 띄고 찾을 문자열을 입력하세요.")
input_data = input().split()
n = int(input_data[0]) # 원소의 개수
target = input_data[1] # 찾고자 하는 문자열

print("앞서 적은 원소 개수만큼 문자열을 입력하세요. 구분은 띄어쓰기 한 칸으로 합니다.")
array = input().split()

# 순차 탐색 수행 결과 출력
print(sequential_search(n, target, array))
```

- 7-1.cpp

```
#include <bits/stdc++.h>

using namespace std;

// 순차 탐색 소스코드 구현
int sequantialSearch(int n, string target, vector<string> arr) {
    // 각 원소를 하나씩 확인하며
    for (int i = 0; i < n; i++) {
        // 현재의 원소가 찾고자 하는 원소와 동일한 경우
        if (arr[i] == target) {
            return i + 1; // 현재의 위치 반환 (인덱스는 0부터 시작하므로 1 더하기)
        }
    }
    return -1; // 원소를 찾지 못한 경우 -1 반환
}

int n; // 원소의 개수
string target; // 찾고자 하는 문자열
vector<string> arr;
```

```
int main(void) {
    cout << "생성할 원소 개수를 입력한 다음 한 칸 띄고 찾을 문자열을 입력하세요." << '₩n';
    cin >> n >> target;

    cout << "앞서 적은 원소 개수만큼 문자열을 입력하세요. 구분은 띄어쓰기 한 칸으로 합니다." << '₩n';
    for (int i = 0; i < n; i++) {
        string x;
        cin >> x;
        arr.push_back(x);
    }

    // 순차 탐색 수행 결과 출력
    cout << sequantialSearch(n, target, arr) << '₩n';
}
```

- 7-2.py 재귀함수로 구현한 이진탐색 소스코드

```python
# 이진 탐색 소스코드 구현 (반복문)
def binary_search(array, target, start, end):
    while start <= end:
        mid = (start + end) // 2
        # 찾은 경우 중간점 인덱스 반환
        if array[mid] == target:
            return mid
        # 중간점의 값보다 찾고자 하는 값이 작은 경우 왼쪽 확인
        elif array[mid] > target:
            end = mid - 1
        # 중간점의 값보다 찾고자 하는 값이 큰 경우 오른쪽 확인
        else:
            start = mid + 1
    return None

# n(원소의 개수)과 target(찾고자 하는 값)을 입력 받기
n, target = list(map(int, input().split()))
# 전체 원소 입력 받기
array = list(map(int, input().split()))

# 이진 탐색 수행 결과 출력
result = binary_search(array, target, 0, n - 1)
if result == None:
    print("원소가 존재하지 않습니다.")
else:
    print(result + 1)
```

- 7-3.py 반복문으로 구현한 이진탐색 소스코드

```python
# 이진 탐색 소스코드 구현 (반복문)
def binary_search(array, target, start, end):
    while start <= end:
        mid = (start + end) // 2
        # 찾은 경우 중간점 인덱스 반환
        if array[mid] == target:
            return mid
        # 중간점의 값보다 찾고자 하는 값이 작은 경우 왼쪽 확인
        elif array[mid] > target:
            end = mid - 1
        # 중간점의 값보다 찾고자 하는 값이 큰 경우 오른쪽 확인
        else:
            start = mid + 1
    return None

# n(원소의 개수)과 target(찾고자 하는 값)을 입력 받기
n, target = list(map(int, input().split()))
# 전체 원소 입력 받기
array = list(map(int, input().split()))

# 이진 탐색 수행 결과 출력
result = binary_search(array, target, 0, n - 1)
if result == None:
    print("원소가 존재하지 않습니다.")
else:
    print(result + 1)
```

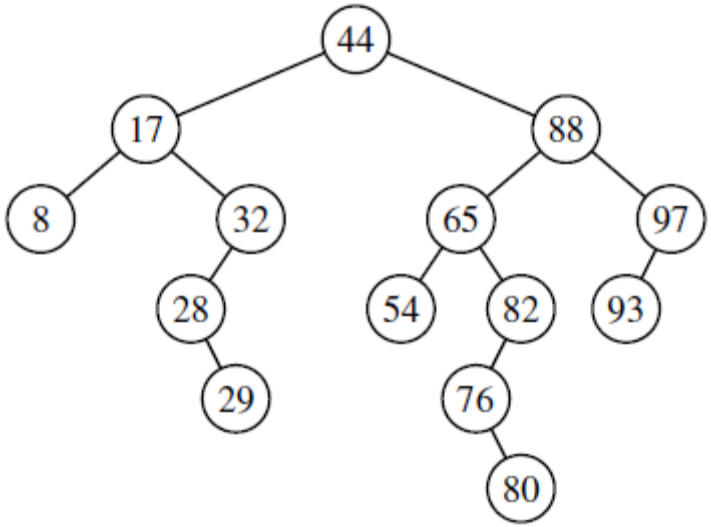
- 7-3.cpp

```
#include <bits/stdc++.h>

using namespace std;

// 이진 탐색 소스코드 구현(반복문)
int binarySearch(vector<int>& arr, int target, int start, int end) {
    while (start <= end) {
        int mid = (start + end) / 2;
        // 찾은 경우 중간점 인덱스 반환
        if (arr[mid] == target) return mid;
        // 중간점의 값보다 찾고자 하는 값이 작은 경우 왼쪽 확인
        else if (arr[mid] > target) end = mid - 1;
        // 중간점의 값보다 찾고자 하는 값이 큰 경우 오른쪽 확인
        else start = mid + 1;
    }
    return -1;
}

int n, target;
vector<int> arr;
```

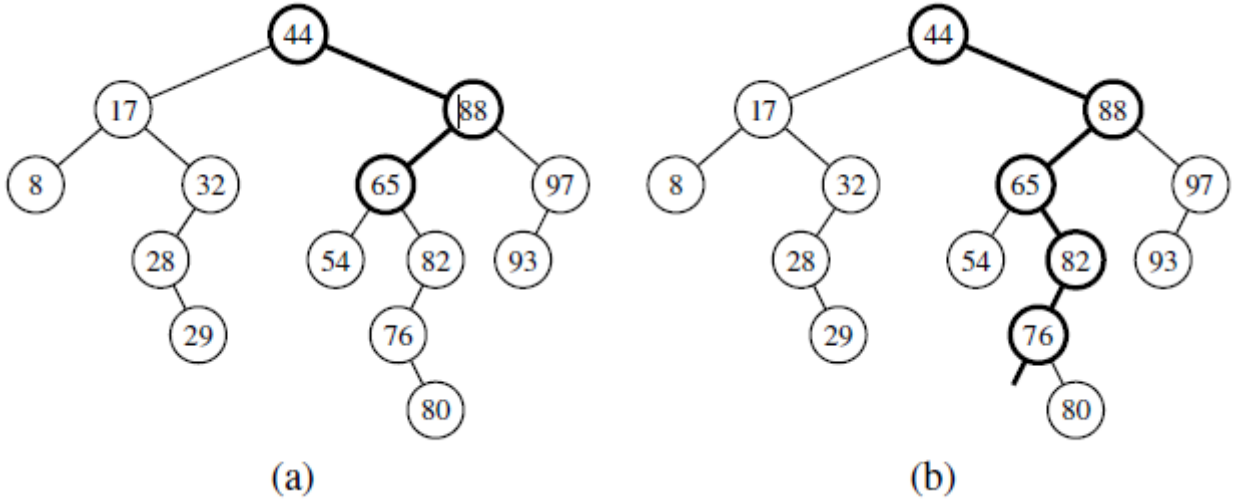
```
int main(void) {
    // n(원소의 개수)와 target(찾고자 하는 값)을 입력 받기
    cin >> n >> target;
    // 전체 원소 입력 받기
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        arr.push_back(x);
    }
    // 이진 탐색 수행 결과 출력
    int result = binarySearch(arr, target, 0, n - 1);
    if (result == -1) {
        cout << "원소가 존재하지 않습니다." << '₩n';
    }
    else {
        cout << result + 1 << '₩n';
    }
}
```

# 이진탐색트리

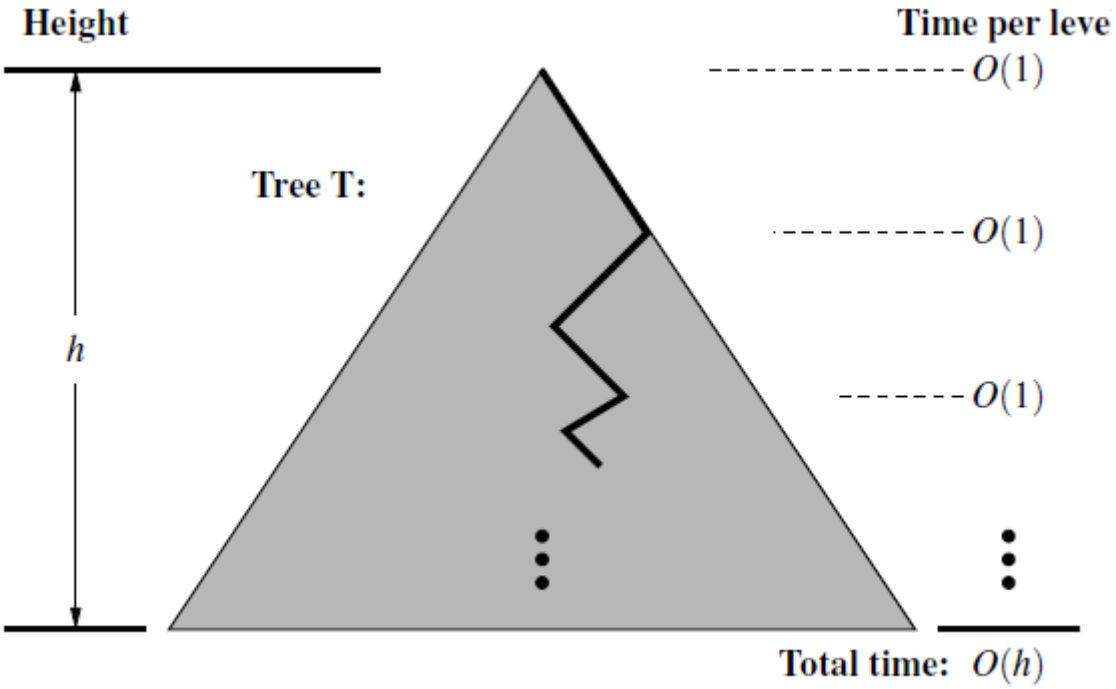
- BST



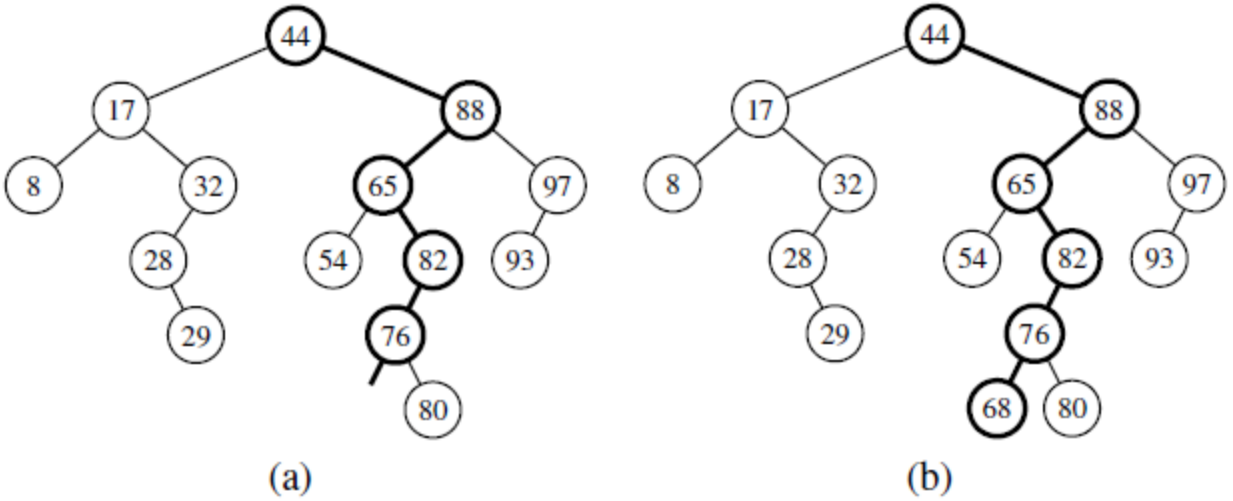
- BST 검색




(a) 검색: key 65  성공 (b) 검색: key 68 - unsuccessful


Height
Time per leve
O(1)
Tree T:
O(1)
h
O(1)
Total time: O(h)

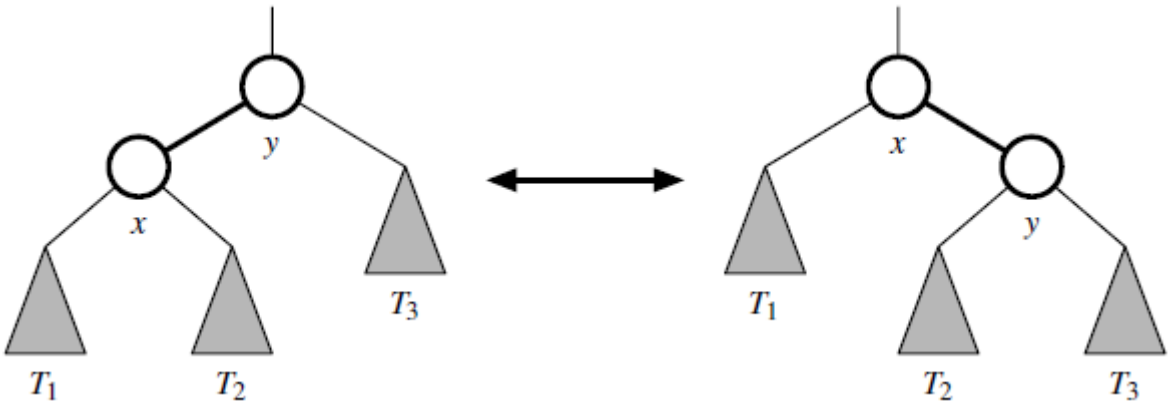
- 삽입과 삭제




삽입 - key 68

(a) 위치 확인 (b) 삽입완료

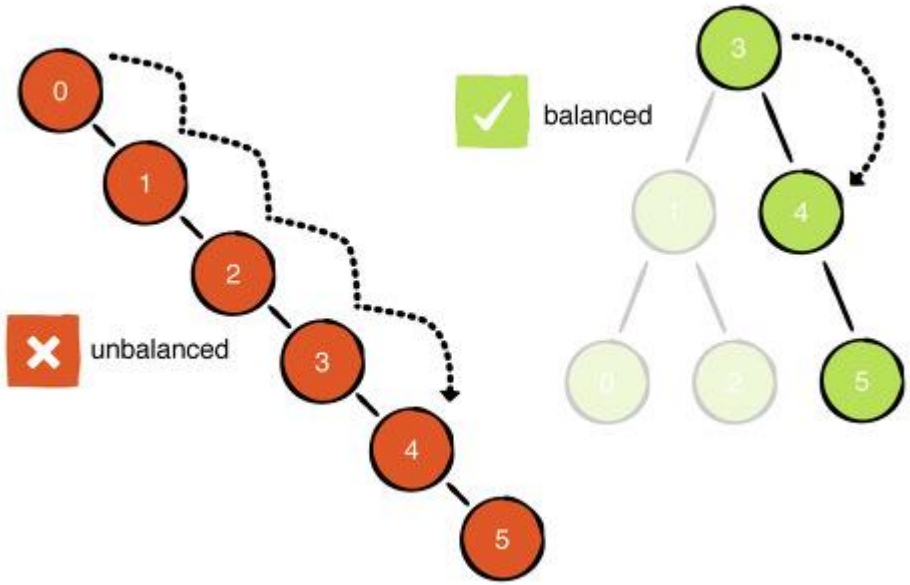
- 자가균형트리 (Self-balancing Binary Search Tree)
  - 편향성을 해결하면 BST에서 삽입, 검색, 삭제가 모두 O(log n)이 됨.




rotation operation

  - 예:
    - AVL tree
    - Red Black tree
    - Splay tree

– AVL Tree의 예



- C++의 경우 STL의 set, multiset, map + iterator 이용
  - hash = unordered_set, unordered_multiset, unordered_map
  - BST = set, multiset, map
  - (단, set, map이 unordered_set, unordered_multiset, unordered_map 보다 성능이 안정적 즉, O(log n)

```
#include <cstdio>
#include <set>
#include <map>
#include <string>
#include <functional>
#include <iostream>

using namespace std;
```

```
int main() {
    set<int, greater<int> > s;
    map<int, string, less<int> > m;

    // --- set example
    s.insert(4), s.insert(99), s.insert(0);
    s.insert(24), s.insert(9), s.insert(10);
    auto s_iter = s.find(4);
    if (s_iter != s.end()) s.erase(s_iter);
    s.erase(s.find(0));
    for (const auto &i : s) printf("%d ", i);puts("");
    cout << "lower bound of 11 : " << *(s.lower_bound(11)) << endl;
    cout << "upper bound of 11 : " << *(s.upper_bound(11)) << endl;

    // --- map example
    m.insert(make_pair(6, "six")), m.insert(make_pair(5, "five"));
    m.emplace(1, "one"), m.emplace(2, "two"), m.emplace(10, "ten");
    m[3] = "three", m[4] = "four";
    m.erase(4);
    for (const auto &i : m) printf("%d: %s, ", i.first, i.second.c_str());puts("");
    cout << "lower bound of 4 : " << (*m.lower_bound(4)).second << endl;

    return 0;
}
```

실습과제

# 2 [실전 문제] 부품 찾기

난이도 ●◐○ | 풀이 시간 30분 | 시간 제한 1초 | 메모리 제한 128MB

동빈이네 전자 매장에는 부품이 N개 있다. 각 부품은 정수 형태의 고유한 번호가 있다. 어느 날 손님이 M개 종류의 부품을 대량으로 구매하겠다며 당일 날 견적서를 요청했다. 동빈이는 때를 놓치지 않고 손님이 문의한 부품 M개 종류를 모두 확인해서 견적서를 작성해야 한다. 이때 가게 안에 부품이 모두 있는지 확인하는 프로그램을 작성해보자.

예를 들어 가게의 부품이 총 5개일 때 부품 번호가 다음과 같다고 하자.

```
N = 5
[8, 3, 7, 9, 2]
```

손님은 총 3개의 부품이 있는지 확인 요청했는데 부품 번호는 다음과 같다.

```
M = 3
[5, 7, 9]
```

이때 손님이 요청한 부품 번호의 순서대로 부품을 확인해 부품이 있으면 yes를, 없으면 no를 출력한다. 구분은 공백으로 한다.

**입력 조건**

- 첫째 줄에 정수 N이 주어진다. ($1 \le N \le 1{,}000{,}000$)
- 둘째 줄에는 공백으로 구분하여 N개의 정수가 주어진다. 이때 정수는 1보다 크고 1,000,000 이하이다.
- 셋째 줄에는 정수 M이 주어진다. ($1 \le M \le 100{,}000$)
- 넷째 줄에는 공백으로 구분하여 M개의 정수가 주어진다. 이때 정수는 1보다 크고 1,000,000 이하이다.

**출력 조건**

- 첫째 줄에 공백으로 구분하여 각 부품이 존재하면 yes를, 없으면 no를 출력한다.

**입력 예시**

```
5
8 3 7 9 2
3
5 7 9
```

**출력 예시**

```
no yes yes
```

- Python
- C++

실습과제

# 3 [실전 문제] 떡볶이 떡 만들기

난이도 ●●○ | 풀이 시간 40분 | 시간 제한 2초 | 메모리 제한 128MB

오늘 동빈이는 여행 가신 부모님을 대신해서 떡집 일을 하기로 했다. 오늘은 떡볶이 떡을 만드는 날이다. 동빈이네 떡볶이 떡은 재밌게도 떡볶이 떡의 길이가 일정하지 않다. 대신에 한 봉지 안에 들어가는 떡의 총 길이는 절단기로 잘라서 맞춰준다.

절단기에 높이(H)를 지정하면 줄지어진 떡을 한 번에 절단한다. 높이가 H보다 긴 떡은 H 위의 부분이 잘릴 것이고, 낮은 떡은 잘리지 않는다.

예를 들어 높이가 19, 14, 10, 17cm인 떡이 나란히 있고 절단기 높이를 15cm로 지정하면 자른 뒤 떡의 높이는 15, 14, 10, 15cm가 될 것이다. 잘린 떡의 길이는 차례대로 4, 0, 0, 2cm이다. 손님은 6cm만큼의 길이를 가져간다.

손님이 왔을 때 요청한 총 길이가 M일 때 적어도 M만큼의 떡을 얻기 위해 절단기에 설정할 수 있는 높이의 최댓값을 구하는 프로그램을 작성하시오.

**입력 조건**
- 첫째 줄에 떡의 개수 N과 요청한 떡의 길이 M이 주어진다. ($1 \le N \le 1{,}000{,}000$, $1 \le M \le 2{,}000{,}000{,}000$)
- 둘째 줄에는 떡의 개별 높이가 주어진다. 떡 높이의 총합은 항상 M 이상이므로, 손님은 필요한 양만큼 떡을 사갈 수 있다. 높이는 10억보다 작거나 같은 양의 정수 또는 0이다.

**출력 조건**
- 적어도 M만큼의 떡을 집에 가져가기 위해 절단기에 설정할 수 있는 높이의 최댓값을 출력한다.

**입력 예시**

```
4 6
19 15 10 17
```

**출력 예시**

```
15
```

- Python

- C++

# Chapter 05 DFS/BFS

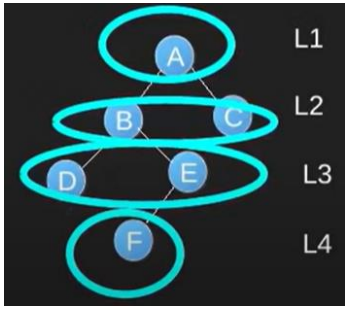
- 1 꼭 필요한 자료구조 기초
- 2 탐색 알고리즘 DFS/BFS
- 3 [실전 문제] 음료수 얼려 먹기
- 4 [실전 문제] 미로 탈출

# 1 꼭 필요한 자료구조 기초

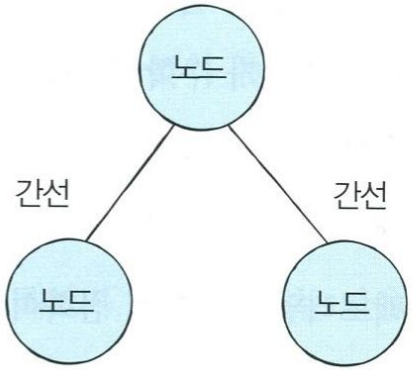
- 스택
- 큐
- 재귀함수

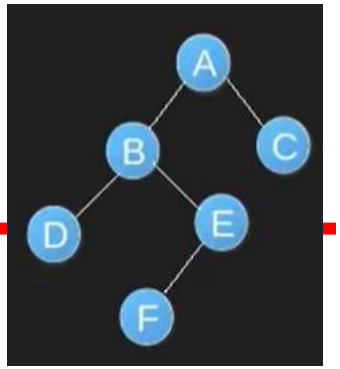
# 탐색 알고리즘 DFS/BFS

- 트리 순회 (Tree traversal)
  - 트리 내의 모든 노드(node)를 오직 한 번씩 방문하
- 종류:
  - Breadth First Traversal (level order)
    - Layer별로 순회하는 것



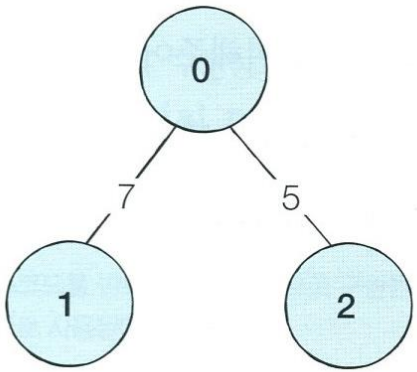

  - Depth First Traversal
    - Preorder Traversal (전위 순회)
    - Inorder Traversal (중위 순회)
    - Postorder Traversal (후위 순회)
- 시간복잡도는 모두 O(n)  단, n = node의 수



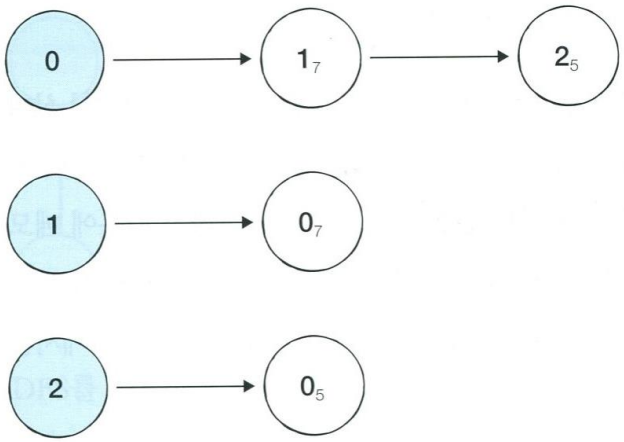



- Depth First Traversal
  - Preorder Traversal (전위 순회) <root><Left><Right>
    - A B D E F C
  - Inorder Traversal (중위 순회) <Left><root><Right>
    - D B F E A C
  - Postorder Traversal (후위 순회) <Left><Right><root>
    - D F E B C A

# DFS




| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0 | 7 | 5 |
| 1 | 7 | 0 | 무한 |
| 2 | 5 | 무한 | 0 |



- 5-6.py 인접행렬 방식 예제

```
INF = 999999999 # 무한의 비용 선언

# 2차원 리스트를 이용해 인접 행렬 표현
graph = [
    [0, 7, 5],
    [7, 0, INF],
    [5, INF, 0]
]

print(graph)
```

- 5-6.cpp

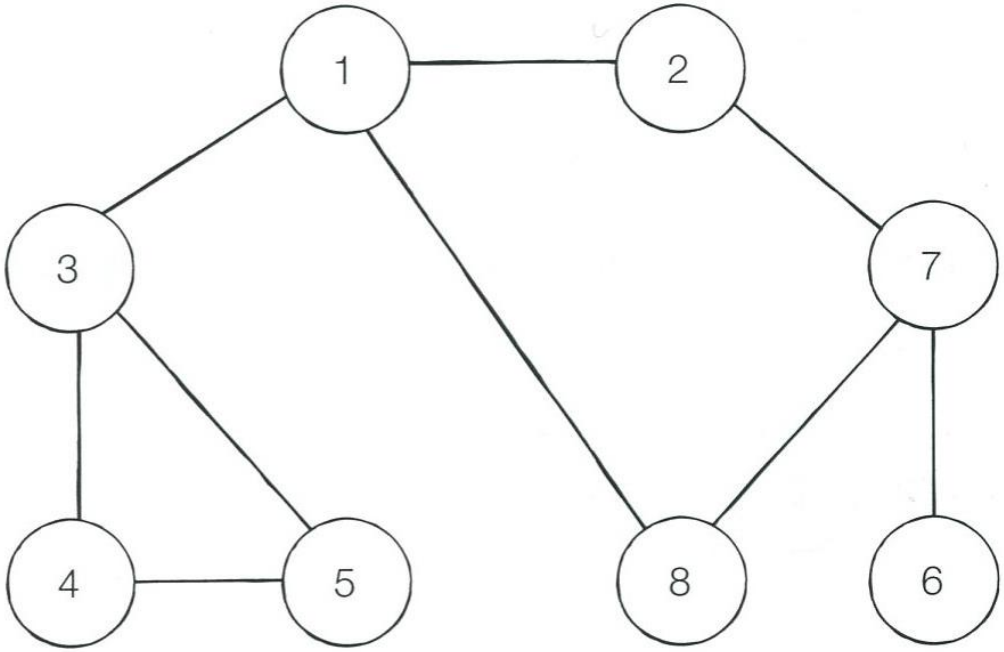
```
#include <bits/stdc++.h>
#define INF 999999999 // 무한의 비용 선언

using namespace std;

// 2차원 리스트를 이용해 인접 행렬 표현
int graph[3][3] = {
    {0, 7, 5},
    {7, 0, INF},
    {5, INF, 0}
};

int main(void) {
    // 그래프 출력
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            cout << graph[i][j] << ' ';
        }
        cout << '₩n';
    }
}
```

- 5-7.py 인접 리스트 방식 예제

```
# 행(Row)이 3개인 2차원 리스트로 인접 리스트 표현
graph = [[] for _ in range(3)]

# 노드 0에 연결된 노드 정보 저장 (노드, 거리)
graph[0].append((1, 7))
graph[0].append((2, 5))

# 노드 1에 연결된 노드 정보 저장 (노드, 거리)
graph[1].append((0, 7))

# 노드 2에 연결된 노드 정보 저장 (노드, 거리)
graph[2].append((0, 5))

print(graph)
```

- 5-7.cpp 인접 리스트 방식 예제

```cpp
#include <bits/stdc++.h>

using namespace std;

// 행(Row)이 3개인 인접 리스트 표현
vector<pair<int, int> > graph[3];

int main(void) {
    // 노드 0에 연결된 노드 정보 저장 {노드, 거리}
    graph[0].push_back({1, 7});
    graph[0].push_back({2, 5});

    // 노드 1에 연결된 노드 정보 저장 {노드, 거리}
    graph[1].push_back({0, 7});

    // 노드 2에 연결된 노드 정보 저장 {노드, 거리}
    graph[2].push_back({0, 5});

    // 그래프 출력
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < graph[i].size(); j++) {
            cout << '(' << graph[i][j].first << ',' << graph[i][j].second << ')' << ' ';
        }
        cout << '₩n';
    }
}
```


1
2
3
7
4
5
8
6

- 5-8.py DFS 예제

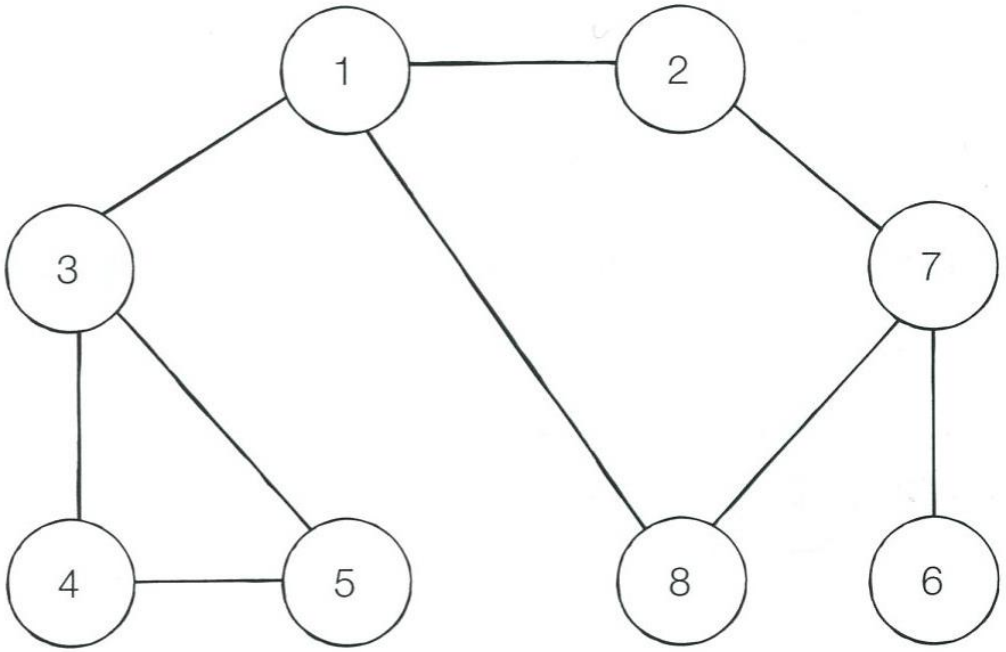
```
# DFS 함수 정의
def dfs(graph, v, visited):
    # 현재 노드를 방문 처리
    visited[v] = True
    print(v, end=' ')
    # 현재 노드와 연결된 다른 노드를 재귀적으로 방문
    for i in graph[v]:
        if not visited[i]:
            dfs(graph, i, visited)

# 각 노드가 연결된 정보를 리스트 자료형으로 표현(2차원 리스트)
graph = [
  [],
  [2, 3, 8],    [1, 7],    [1, 4, 5],    [3, 5],
  [3, 4],    [7],    [2, 6, 8],    [1, 7]
]
# 각 노드가 방문된 정보를 리스트 자료형으로 표현(1차원 리스트)
visited = [False] * 9
# 정의된 DFS 함수 호출
dfs(graph, 1, visited)
```

- 5-8.cpp DFS 예제

```
#include <bits/stdc++.h>

using namespace std;

bool visited[9];
vector<int> graph[9];

// DFS 함수 정의
void dfs(int x) {
    // 현재 노드를 방문 처리
    visited[x] = true;
    cout << x << ' ';
    // 현재 노드와 연결된 다른 노드를 재귀적으로 방문
    for (int i = 0; i < graph[x].size(); i++) {
        int y = graph[x][i];
        if (!visited[y]) dfs(y);
    }
}
```

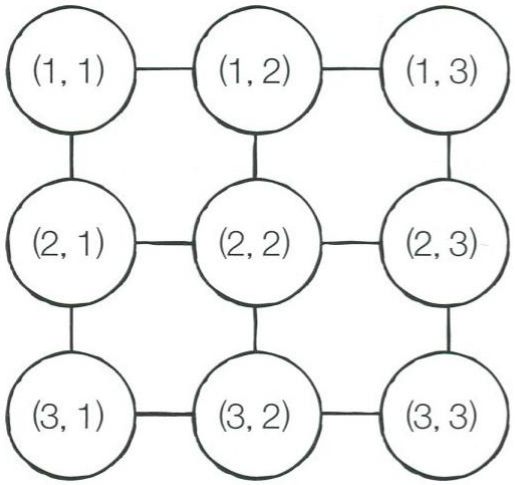
```
int main(void) {
    // 노드 1에 연결된 노드 정보 저장
    graph[1].push_back(2);
    graph[1].push_back(3);
    graph[1].push_back(8);

    // 노드 2에 연결된 노드 정보 저장
    graph[2].push_back(1);
    graph[2].push_back(7);

    // 노드 3에 연결된 노드 정보 저장
    graph[3].push_back(1);
    graph[3].push_back(4);
    graph[3].push_back(5);

    // 노드 4에 연결된 노드 정보 저장
    graph[4].push_back(3);
    graph[4].push_back(5);

    // 노드 5에 연결된 노드 정보 저장
    graph[5].push_back(3);
    graph[5].push_back(4);

    // 노드 6에 연결된 노드 정보 저장
    graph[6].push_back(7);

    // 노드 7에 연결된 노드 정보 저장
    graph[7].push_back(2);
    graph[7].push_back(6);
    graph[7].push_back(8);

    // 노드 8에 연결된 노드 정보 저장
```

- BFS (너비우선 탐색)
  - 그래프에서 가까운 노드부터 우선적으로 탐색



| | DFS | BFS |
|---|---|---|
| 동작 원리 | 스택 | 큐 |
| 구현 방법 | 재귀 함수 이용 | 큐 자료구조 이용 |

| | | |
|---|---|---|
| (1, 1) | (1, 2) | (1, 3) |
| (2, 1) | (2, 2) | (2, 3) |
| (3, 1) | (3, 2) | (3, 3) |



- 5-9.py BFS 예제

```
from collections import deque

# BFS 함수 정의
def bfs(graph, start, visited):
    # 큐(Queue) 구현을 위해 deque 라이브러리 사용
    queue = deque([start])
    # 현재 노드를 방문 처리
    visited[start] = True
    # 큐가 빌 때까지 반복
    while queue:
        # 큐에서 하나의 원소를 뽑아 출력
        v = queue.popleft()
        print(v, end=' ')
        # 해당 원소와 연결된, 아직 방문하지 않은 원소들을 큐에 삽입
        for i in graph[v]:
            if not visited[i]:
                queue.append(i)
                visited[i] = True
```

```
# 각 노드가 연결된 정보를 리스트 자료형으로 표현(2차원 리스트)
graph = [
  [],
  [2, 3, 8],
  [1, 7],
  [1, 4, 5],
  [3, 5],
  [3, 4],
  [7],
  [2, 6, 8],
  [1, 7]
]

# 각 노드가 방문된 정보를 리스트 자료형으로 표현(1차원 리스트)
visited = [False] * 9

# 정의된 BFS 함수 호출
bfs(graph, 1, visited)
```

- 5-9.cpp

```cpp
#include <bits/stdc++.h>

using namespace std;

bool visited[9];
vector<int> graph[9];

// BFS 함수 정의
void bfs(int start) {
    queue<int> q;
    q.push(start);
    // 현재 노드를 방문 처리
    visited[start] = true;
    // 큐가 빌 때까지 반복
    while(!q.empty()) {
            // 큐에서 하나의 원소를 뽑아 출력
        int x = q.front();
        q.pop();
        cout << x << ' ';
        // 해당 원소와 연결된, 아직 방문하지 않은 원소들을 큐에 삽입
        for(int i = 0; i < graph[x].size(); i++) {
            int y = graph[x][i];
            if(!visited[y]) {
                q.push(y);
                visited[y] = true;
            }
        }
    }
}
```

```
int main(void) {
    // 노드 1에 연결된 노드 정보 저장
    graph[1].push_back(2);
    graph[1].push_back(3);
    graph[1].push_back(8);

    // 노드 2에 연결된 노드 정보 저장
    graph[2].push_back(1);
    graph[2].push_back(7);

    // 노드 3에 연결된 노드 정보 저장
    graph[3].push_back(1);
    graph[3].push_back(4);
    graph[3].push_back(5);

    // 노드 4에 연결된 노드 정보 저장
    graph[4].push_back(3);
    graph[4].push_back(5);

    // 노드 5에 연결된 노드 정보 저장
    graph[5].push_back(3);
    graph[5].push_back(4);

    // 노드 6에 연결된 노드 정보 저장
    graph[6].push_back(7);

    // 노드 7에 연결된 노드 정보 저장
    graph[7].push_back(2);
    graph[7].push_back(6);
    graph[7].push_back(8);
```

# 3 [실전 문제] 음료수 얼려 먹기

난이도 ●◐○ | 풀이 시간 30분 | 시간 제한 1초 | 메모리 제한 128MB

N × M 크기의 얼음 틀이 있다. 구멍이 뚫려 있는 부분은 0, 칸막이가 존재하는 부분은 1로 표시된다. 구멍이 뚫려 있는 부분끼리 상, 하, 좌, 우로 붙어 있는 경우 서로 연결되어 있는 것으로 간주한다. 이때 얼음 틀의 모양이 주어졌을 때 생성되는 총 아이스크림의 개수를 구하는 프로그램을 작성하시오. 다음의 4 × 5 얼음 틀 예시에서는 아이스크림이 총 3개 생성된다.

```
00110
00011
11111
00000
```

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 |

**입력 조건**
- 첫 번째 줄에 얼음 틀의 세로 길이 N과 가로 길이 M이 주어진다. (1 ≤ N, M ≤ 1,000)
- 두 번째 줄부터 N + 1번째 줄까지 얼음 틀의 형태가 주어진다.
- 이때 구멍이 뚫려있는 부분은 0, 그렇지 않은 부분은 1이다.

**출력 조건**
- 한 번에 만들 수 있는 아이스크림의 개수를 출력한다.

- Python

- C++

실습과제

# 4 [실전 문제] 미로 탈출

난이도 ●◐○ | 풀이 시간 30분 | 시간 제한 1초 | 메모리 제한 128MB

동빈이는 N × M 크기의 직사각형 형태의 미로에 갇혀 있다. 미로에는 여러 마리의 괴물이 있어 이를 피해 탈출해야 한다. 동빈이의 위치는 (1, 1)이고 미로의 출구는 (N, M)의 위치에 존재하며 한 번에 한 칸씩 이동할 수 있다. 이때 괴물이 있는 부분은 0으로, 괴물이 없는 부분은 1로 표시되어 있다. 미로는 반드시 탈출할 수 있는 형태로 제시된다. 이때 동빈이가 탈출하기 위해 움직여야 하는 최소 칸의 개수를 구하시오. 칸을 셀 때는 시작 칸과 마지막 칸을 모두 포함해서 계산한다.

**입력 조건**
- 첫째 줄에 두 정수 N, M(4 ≤ N, M ≤ 200)이 주어집니다. 다음 N개의 줄에는 각각 M개의 정수(0 혹은 1)로 미로의 정보가 주어진다. 각각의 수들은 공백 없이 붙어서 입력으로 제시된다. 또한 시작 칸과 마지막 칸은 항상 1이다.

**출력 조건**
- 첫째 줄에 최소 이동 칸의 개수를 출력한다.

**입력 예시**

```
5 6
101010
111111
000001
111111
111111
```

**출력 예시**

```
10
```

- Python

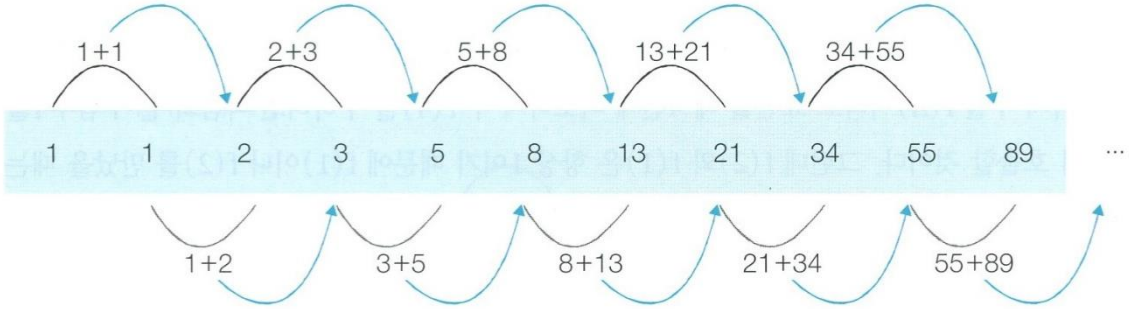
- C++

# Chapter 8 다이나믹 프로그래밍 (1)

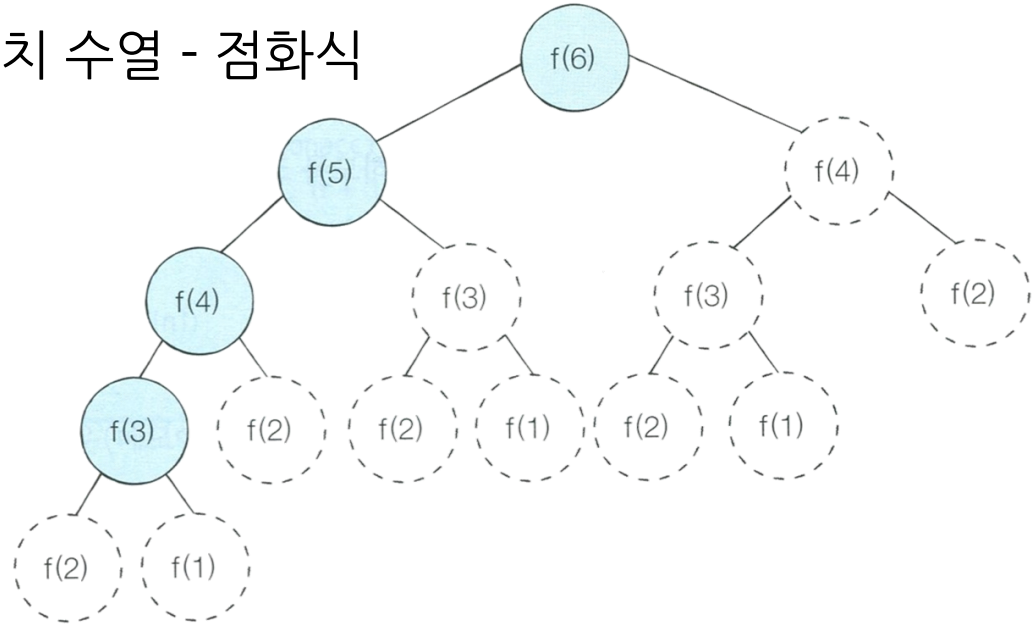
# 1 다이나믹 프로그래밍

- 개념
  - Dynamic Programming starts by solving subproblems and builds up to solving the big problem
  - 다음 조건을 만족할 때 사용
    - (1) 최적 부분구조 (Optimal Substructure)
      - 큰 문제를 작은 문제로 나눌 수 있으며 작은 문제의 답을 모아서 큰 문제를 해결하는 것
    - (2) 중복되는 부분문제 (Overlapping Subproblem)
      - 동일한 작은 문제를 반복적으로 해결
  - 메모리를 적절히 사용하여 시간 효율성을 향상
  - 종류
    - Top-down
    - Bottom-up

- 중복되는 연산을 줄이자




  - 피보나치 수열 - 점화식



- 8-1.py 피보나치 함수 소스코드

```
# 피보나치 함수(Fibonacci Function)을 재귀함수로 구현
def fibo(x):
    if x == 1 or x == 2:
        return 1
    return fibo(x - 1) + fibo(x - 2)

print(fibo(4))
```

- 8-2.py 피보나치 함수 소스코드 (재귀적)

```
# 한 번 계산된 결과를 메모이제이션(Memoization)하기 위한 리스트 초기화
d = [0] * 100

# 피보나치 함수(Fibonacci Function)를 재귀함수로 구현 (탑다운 다이나믹 프로그래밍)
def fibo(x):
    # 종료 조건(1 혹은 2일 때 1을 반환)
    if x == 1 or x == 2:
        return 1
    # 이미 계산한 적 있는 문제라면 그대로 반환
    if d[x] != 0:
        return d[x]
    # 아직 계산하지 않은 문제라면 점화식에 따라서 피보나치 결과 반환
    d[x] = fibo(x - 1) + fibo(x - 2)
    return d[x]

print(fibo(99))
```

- 8-3.py 호출되는 함수 확인

```
d = [0] * 100

def fibo(x):
    print('f(' + str(x) + ')', end=' ')
    if x == 1 or x == 2:
        return 1
    if d[x] != 0:
        return d[x]
    d[x] = fibo(x - 1) + fibo(x - 2)
    return d[x]

fibo(6)
```

- 8-4.py 피보나치 수열 소스코드 (반복적)

```
# 앞서 계산된 결과를 저장하기 위한 DP 테이블 초기화
d = [0] * 100

# 첫 번째 피보나치 수와 두 번째 피보나치 수는 1
d[1] = 1
d[2] = 1
n = 99

# 피보나치 함수(Fibonacci Function) 반복문으로 구현
(보텀업 다이나믹 프로그래밍)
for i in range(3, n + 1):
    d[i] = d[i - 1] + d[i - 2]

print(d[n])
```

- 8-1.cpp

```
#include <bits/stdc++.h>

using namespace std;

// 피보나치 함수(Fibonacci Function)을 재귀함수로 구현
int fibo(int x) {
    if (x == 1 || x == 2) {
        return 1;
    }
    return fibo(x - 1) + fibo(x - 2);
}

int main(void) {
    cout << fibo(4) << '₩n';
}
```

- 8-2.cpp (피보나치 수열 - 재귀적)

```
#include <bits/stdc++.h>
using namespace std;
// 한 번 계산된 결과를 메모이제이션(Memoization)하기 위한 배열 초기화
long long d[100];

// 피보나치 함수(Fibonacci Function)를 재귀함수로 구현 (탑다운 다이나믹 프로그래밍)
long long fibo(int x) {
    // 종료 조건(1 혹은 2일 때 1을 반환)
    if (x == 1 || x == 2) {
        return 1;
    }
    // 이미 계산한 적 있는 문제라면 그대로 반환
    if (d[x] != 0) {
        return d[x];
    }
    // 아직 계산하지 않은 문제라면 점화식에 따라서 피보나치 결과 반환
    d[x] = fibo(x - 1) + fibo(x - 2);
    return d[x];
}

int main(void) {
    cout << fibo(50) << '₩n';
}
```

- 8-3.cpp (호출되는 함수 확인)

```cpp
#include <bits/stdc++.h>

using namespace std;

long long d[100];

long long fibo(int x) {
    cout << "f(" << x << ") ";
    if (x == 1 || x == 2) {
        return 1;
    }
    if (d[x] != 0) {
        return d[x];
    }
    d[x] = fibo(x - 1) + fibo(x - 2);
    return d[x];
}

int main(void) {
    fibo(6);
}
```

- 8-4.cpp 피보나치 수열 (반복적)

```
#include <bits/stdc++.h>

using namespace std;

// 앞서 계산된 결과를 저장하기 위한 DP 테이블 초기화
long long d[100];

int main(void) {
    // 첫 번째 피보나치 수와 두 번째 피보나치 수는 1
    d[1] = 1;
    d[2] = 1;
    int n = 50; // 50번째 피보나치 수를 계산

    // 피보나치 함수(Fibonacci Function) 반복문으로 구현(보텀업 다이나믹 프로그래밍)
    for (int i = 3; i <= n; i++) {
        d[i] = d[i - 1] + d[i - 2];
    }
    cout << d[n] << '₩n';
}
```

- Bottom-up
  - DP의 전형적 형태
  - DP table = 결과저장용 리스트
- Top-down
  - Memoization
    - 한번 계산한 결과를 메모리 공간에 메모하는 것
  - 일종의 caching

- DP vs. 분할정복
  - 부분문제의 중복

# 2 [실전 문제] 1로 만들기

난이도 ●◐○ | 풀이 시간 20분 | 시간 제한 1초 | 메모리 제한 128MB

정수 X가 주어질 때 정수 X에 사용할 수 있는 연산은 다음과 같이 4가지이다.

ⓐ X가 5로 나누어떨어지면, 5로 나눈다.

ⓑ X가 3으로 나누어떨어지면, 3으로 나눈다.

ⓒ X가 2로 나누어떨어지면, 2로 나눈다.

ⓓ X에서 1을 뺀다.

정수 X가 주어졌을 때, 연산 4개를 적절히 사용해서 1을 만들려고 한다. 연산을 사용하는 횟수의 최솟값을 출력하시오.

예를 들어 정수가 26이면 다음과 같이 계산해서 3번의 연산이 최솟값이다.

1. 26 − 1 = 25 (ⓓ)
2. 25 / 5 = 5 (ⓐ)
3. 5 / 5 = 1 (ⓐ)

**입력 조건** • 첫째 줄에 정수 X가 주어진다. ($1 \le X \le 30,000$)

**출력 조건** • 첫째 줄에 연산을 하는 횟수의 최솟값을 출력한다.

**입력 예시**

```
26
```

**출력 예시**

```
3
```

- Python

- C++

# 3 [실전 문제] 개미 전사

난이도 ●●○ | 풀이 시간 30분 | 시간 제한 1초 | 메모리 제한 128MB

개미 전사는 부족한 식량을 충당하고자 메뚜기 마을의 식량창고를 몰래 공격하려고 한다. 메뚜기 마을에는 여러 개의 식량창고가 있는데 식량창고는 일직선으로 이어져 있다. 각 식량창고에는 정해진 수의 식량을 저장하고 있으며 개미 전사는 식량창고를 선택적으로 약탈하여 식량을 빼앗을 예정이다. 이때 메뚜기 정찰병들은 일직선상에 존재하는 식량창고 중에서 서로 인접한 식량창고가 공격받으면 바로 알아챌 수 있다. 따라서 개미 전사가 정찰병에게 들키지 않고 식량창고를 약탈하기 위해서는 최소한 한 칸 이상 떨어진 식량창고를 약탈해야 한다. 예를 들어 식량창고 4개가 다음과 같이 존재한다고 가정하자.

```
{1, 3, 1, 5}
```

| 1 | 3 | 1 | 5 |
|---|---|---|---|

이때 개미 전사는 두 번째 식량창고와 네 번째 식량창고를 선택했을 때 최댓값인 총 8개의 식량을 빼앗을 수 있다. 개미 전사는 식량창고가 이렇게 일직선상일 때 최대한 많은 식량을 얻기를 원한다.

개미 전사를 위해 식량창고 N개에 대한 정보가 주어졌을 때 얻을 수 있는 식량의 최댓값을 구하는 프로그램을 작성하시오.

**입력 조건**
- 첫째 줄에 식량창고의 개수 N이 주어진다. ($3 \le N \le 100$)
- 둘째 줄에 공백으로 구분되어 각 식량창고에 저장된 식량의 개수 K가 주어진다. ($0 \le K \le 1,000$)

**출력 조건**
- 첫째 줄에 개미 전사가 얻을 수 있는 식량의 최댓값을 출력하시오.

**입력 예시**

```
4
1 3 1 5
```

**출력 예시**

```
8
```

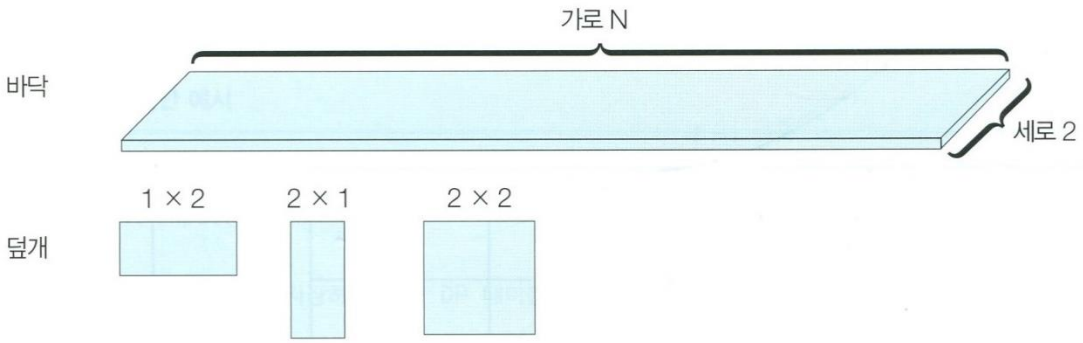
- Python

- C++

# 4 [실전 문제] 바닥 공사

난이도 ●◐○ | 풀이 시간 20분 | 시간 제한 1초 | 메모리 제한 128MB

가로의 길이가 N, 세로의 길이가 2인 직사각형 형태의 얇은 바닥이 있다. 태일이는 이 얇은 바닥을 1 × 2의 덮개, 2 × 1의 덮개, 2 × 2의 덮개를 이용해 채우고자 한다.




이때 바닥을 채우는 모든 경우의 수를 구하는 프로그램을 작성하시오. 예를 들어 2 × 3 크기의 바닥을 채우는 경우의 수는 5가지이다.

**입력 조건** • 첫째 줄에 N이 주어진다. (1 ≤ N ≤ 1,000)

**출력 조건** • 첫째 줄에 2 × N 크기의 바닥을 채우는 방법의 수를 796,796으로 나눈 나머지를 출력한다.

**입력 예시**

```
3
```

**출력 예시**

```
5
```

- Python

- C++

# 5 [실전 문제] 효율적인 화폐 구성

난이도 ●●○ | 풀이 시간 30분 | 시간 제한 1초 | 메모리 제한 128MB

N가지 종류의 화폐가 있다. 이 화폐들의 개수를 최소한으로 이용해서 그 가치의 합이 M원이 되도록 하려고 한다. 이때 각 화폐는 몇 개라도 사용할 수 있으며, 사용한 화폐의 구성은 같지만 순서만 다른 것은 같은 경우로 구분한다. 예를 들어 2원, 3원 단위의 화폐가 있을 때는 15원을 만들기 위해 3원을 5개 사용하는 것이 가장 최소한의 화폐 개수이다.

**입력 조건**

- 첫째 줄에 N, M이 주어진다. ($1 \le N \le 100$, $1 \le M \le 10,000$)
- 이후 N개의 줄에는 각 화폐의 가치가 주어진다. 화폐 가치는 10,000보다 작거나 같은 자연수이다.

**출력 조건**

- 첫째 줄에 M원을 만들기 위한 최소한의 화폐 개수를 출력한다.
- 불가능할 때는 −1을 출력한다.

**입력 예시 1**

```
2 15
2
3
```

**출력 예시 1**

```
5
```

**입력 예시 2**

```
3 4
3
5
7
```

**출력 예시 2**

```
-1
```

- Python

- C++

# 과제 실습