

OT

25.03.17 (월) 오후 5시 ~ 7시

목차

- 스터디 소개
- PS를 위한 C++
- 재귀
- 비트마스킹
- 백트래킹

스터디 소개

스터디 소개 | 01 개요

- 스터디 내용
 - ps에서 사용하는 기초적인 개념과 실제 문제에 적용하는 방법
- 권장 대상
 - Silver 5 ~ Gold 3 (solved.ac 클래스 2 ~ 4) 정도에서 다루는 알고리즘을 공부하고 싶은 분들
- 선수 조건
 - c언어 문법: 입출력, 조건문, 반복문, 배열, 함수
- 여러 명을 대상으로 강의한 경험이 부족므로, 피드백을 주시면 개선해 나가겠습니다.
- 강의 중간에 이해가 되지 않는 부분이나 설명이 부족한 부분이 있으면 편하게 말해주세요.

스터디 소개 | 01 개요

- 강의 주제는 진행자의 판단에 따라 변동될 수 있습니다.
- 5차시는 공휴일이라 온라인으로 진행될 수 있습니다.
- 월요일 오후 5시 ~ 7시에 정보과학관 모든 강의실이 예약이 불가능합니다.
- 최대한 정보과학관 근처 강의실을 예약할 예정이니 양해 부탁드립니다.

차시	날짜	주제	강의 자료
1차시	25.03.17(월)	OT : PS를 위한 C++, 재귀, 백트래킹	링크
2차시	25.03.24(월)	알고리즘 기초 : 시간 복잡도, 정렬, 이분 탐색, 분할 정복	링크
3차시	25.03.31(월)	정수론 기초: 거듭제곱, 소인수 분해, 에라토스테네스의 체, 유클리드 호제법	링크
중간고사 시간(6~8주차)	-	-	-
4차시	25.04.28(월)	동적 계획법	링크
5차시	25.05.05(월)	그리디, 비트마스킹, 서로소 집합	링크
6차시	25.05.12(월)	자료구조 : 스택, 큐, 덱	링크
7차시	25.05.19(월)	그래프 기초: 그래프, 깊이 우선 탐색, 너비 우선 탐색	링크
8차시	25.05.26(월)	최단 경로 : 다익스트라, 벨만 포드, 플로이드 워셜	링크
기말고사 시간(14~16주차)	-	-	-

강의 자료 : github.com/seonghwan7694/SCCC-Begginer

스터디 소개 | 02 solved.ac 란?

- 백준 문제를 난이도별로 정리하고, 티어 시스템을 적용한 랭킹 사이트

레벨	미해결	해결	전체	진행도
모든 문제	모두 보기	모두 보기	모두 보기	
Unrated	5,915	3	5,918	<div><div></div></div>
Bronze V	72	92	164	<div><div></div></div>
Bronze IV	270	59	329	<div><div></div></div>
Bronze III	683	74	757	<div><div></div></div>
Bronze II	897	72	969	<div><div></div></div>
Bronze I	764	51	815	<div><div></div></div>
Silver V	844	60	904	<div><div></div></div>
Silver IV	855	65	920	<div><div></div></div>
Silver III	969	70	1,039	<div><div></div></div>
Silver II	909	84	993	<div><div></div></div>
Silver I	977	61	1,038	<div><div></div></div>
Gold V	1,152	74	1,226	<div><div></div></div>
Gold IV	1,583	84	1,667	<div><div></div></div>
Gold III	1,455	52	1,507	<div><div></div></div>
Gold II	1,330	42	1,372	<div><div></div></div>
Gold I	1,101	32	1,133	<div><div></div></div>
Platinum V	1,314	28	1,342	<div><div></div></div>
Platinum IV	1,284	17	1,301	<div><div></div></div>
Platinum III	1,377	8	1,385	<div><div></div></div>

스터디 소개 | 02 solved.ac 란?

- 문제에 알고리즘 태그가 붙어 있어서, 특정 유형의 문제를 연습 가능


#그래프 탐색 graph_traversal

정렬 ID 레벨 제목 푼 사람 수 ↓ 평균 시도 랜덤

#		제목		푼 사람 수	평균 시도
2 1260	🗨	DFS와 BFS STANDARD	🔖	75,453	2.60
1 2178	🗨	미로 탐색	🔖	67,199	2.22
3 2606	🗨	바이러스 STANDARD	🔖	64,429	2.17
1 2667	🗨	단지번호붙이기	🔖	60,168	2.31
2 1012	🗨	유기농 배추 STANDARD	🔖	59,166	2.56
5 7576	🗨	토마토 STANDARD	🔖	54,048	2.66

스터디 소개 | 02 solved.ac 란?

- 검색을 통해 풀고 싶은 문제를 찾을 수 있음.

 #graphs s#1000.. *s5..g3 -@\$me

문제

사용자

태그

정렬

ID






레벨

제목

풀 사람 수 ↓

평균 시도

랜덤

#	제목
 1389	케빈 베이컨의 6단계 법칙
 17070	파이프 옮기기 1
 10451	순열 사이클
 17471	게리맨더링
 11559	Puyo Puyo

#graphs : 그래프 태그가 붙여진 문제

s#1000.. : 1000명 이상 푼 문제

*s5..g3 : 실버 5와 골드 3 사이의 문제

-@\$me : 내가 푼 문제를 제외하고

스터디 소개 | 02 solved.ac 란?

- CLASS : PS에서 마주치게 되는 주제와 트릭을 단계별로 모아둔 학습 로드맵
 - 클래스를 달성하면 레이팅을 올릴 수 있음.



스터디 소개 | 03 알고리즘 문제 푸는 과정

1. 문제를 정확히 이해하기.

- 입력과 출력 형식 확인 : 어떤 데이터가 주어지고, 어떤 형식으로 결과를 출력해야 하는지 확인.
- 제약 사항 확인 : 데이터의 크기, 특정한 조건(메모리 제한, 시간 제한) 등을 확인.
- 예제 분석 : 주어진 예제를 직접 손으로 풀어보면서 문제의 패턴을 파악.

2. 풀이를 생각하기

- 시간 복잡도와 공간 복잡도를 계산해서 대략 어느 정도 성능을 가진 알고리즘을 생각해내야 하는지 판단하기.
- 컴퓨터는 간단한 동작을 1초에 1억 번 할 수 있음.
 - $n = 5000$ 이면 $O(N^2)$ 정도의 알고리즘을 작성하면 되겠구나.
 - $n = 200,000$ 이면 $O(n \log n)$ 정도의 알고리즘을 작성하면 되겠구나.
- 수 하나는 4B ~ 8B이다
 - `int a[10'000'000]`은 $40,000,000B = 40MB$
 - `int a[5'000][5'000]`은 $100,000,000B = 100MB$

스터디 소개 | 03 알고리즘 문제 푸는 과정

3. 코드로 옮기기
4. 온라인 저지에 제출하기
5. 왜 틀렸는지 고민하기

맞왜틀 팁 : 엣지 케이스(가장 작은 케이스, 경계에 있는 케이스, 정말 큰 케이스)를 직접 테스트해보면 좋음.

(반복)

(반복)

(반복)

※ 안 풀린다고, 2~3시간 이상 너무 오래 고민하지 말기.

※ 다른 사람의 코드를 참고해서 공부하고 넘어가거나, 다음을 약속하기.

PS를 위한 C++

PS를 위한 C++ | 01 개요

- C로도 PS(Problem Solving : 문제 해결)를 할 수 있지만, C++를 추천!


- C는 표준 라이브러리가 부족함.

우선순위 큐를 구현하려면 힙 자료구조를 직접 만들어야 한다.

- 백준 1753번 문제는 다익스트라 알고리즘을 사용하는 대표적인 문제.

C로 풀려면 C++로 6배만큼 타이핑해야한다.

4 1753	맞았습니다!!	11044 KB	152 ms	C99	6621 B
4 1753	맞았습니다!!	9184 KB	88 ms	C++17	1097 B



PS를 위한 C++ | 02 입출력 (I/O) 방식

- C에서는 scanf/printf 를 사용하지만, C++에서는 cin/cout을 사용한다.
 - bits/stdc++.h 는 C++에서 PS할 때 자주 사용되는 모든 표준 라이브러리를 포함하는 헤더 파일이다.
 - C++에서는 표준 라이브러리(STL)를 사용할 때, std::cout 처럼 네임스페이스를 앞에 붙여줘야 한다.
 - using namespace std; 는 "std::"를 생략할 수 있도록 해주는 선언문이다.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5      ios::sync_with_stdio(false); // C와 C++의 입출력 동기화를 끊어 속도 향상
6      cin.tie(nullptr); // 입력과 출력을 묶지 않아 속도 향상
7
8      int a, b;
9      cin >> a >> b; // scanf("%d %d", &a, &b); 와 같은 역할
10     cout << a + b << "\n"; // printf("%d\n", a + b); 와 같은 역할
11
12     return 0;
13 }
```

PS를 위한 C++ | 03 C++ STL

- C++ STL(Standard Template Library)은 자료구조와 알고리즘을 쉽게 사용할 수 있도록 제공하는 표준 라이브러리이다.
 - 데이터를 저장하고 관리하는 자료구조를 제공한다. **컨테이너**라고 불린다.
 - vector, deque, list, array / set, map, unordered_set, unordered_map / stack, queue, priority_queue
 - 컨테이너에서 데이터를 조작하는 다양한 **알고리즘**을 제공한다.
 - sort, binary_search, max_element, min_element, next_permutation 등이 있다.
 - 컨테이너의 요소를 순회할 수 있는 객체를 제공한다. **이터레이터**라고 불린다.
 - vector<int>::iterator it 을 예시로 들 수 있다.
 - it은 vector<int>을 순회할 때 사용할 수 있는 반복자(이터레이터) 객체이다.

PS를 위한 C++ | 03 C++ STL (1) : vector

- vector는 동적 배열 기능(크기가 자동으로 조절되는 기능)을 제공하는 STL 컨테이너이다.

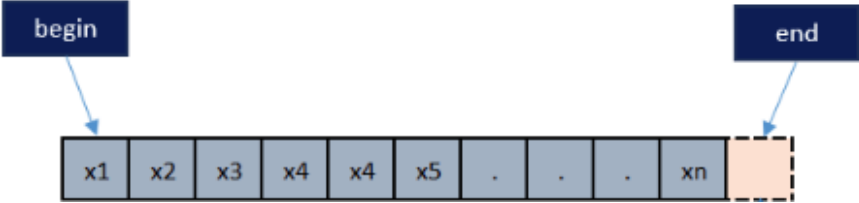
- **사용법**

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main(){
6      vector<int> v1; // 빈 벡터
7      vector<int> v2(5); // 크기 5, 모든 요소는 0으로 초기화
8      vector<int> v3(5, 10); // 크기 5, 모든 요소는 10으로 초기화
9      vector<int> v4 = {1, 2, 3, 4, 5}; // 리스트 초기화
10     vector<vector<int>> v5(5, vector<int>(6, 1)); // 5 x 6 크기, 모든 요소는 1로 초기화
11
12     vector<int> v;
13     v.push_back(1); // {1}
14     v.push_back(2); // {1, 2}
15     v.pop_back(); // {1} (마지막 요소 제거)
16
17     cout << v.size(); // 1 (현재 크기)
18     cout << v.capacity(); // 동적 할당된 크기
19
20     cout << v[0]; // 일반 배열처럼 사용 가능 (0(1))
```


PS를 위한 C++ | 03 C++ STL (1) : vector

- vector는 동적 배열 기능(크기가 자동으로 조절되는 기능)을 제공하는 STL 컨테이너이다.
- **사용법**

```
22 // 반복자 사용 (C++ STL에서 컨테이너의 원소를 순회하는 데 사용하는 객체. 배열에서 포인터처럼 동작하는 개념.)
23 v = {1, 2, 3, 4, 5};
24 for(vector<int>::iterator it = v.begin(); it != v.end(); it++){
25     cout << *it << " ";
26 } // 1 2 3 4 5
27
28 // 범위 기반 for문 (C++11 이상)
29 for(int x : v){
30     cout << x << " ";
31 }
32 }
```



PS를 위한 C++ | 03 C++ STL (2) : sort

- sort함수는 배열이나 컨테이너(vector 등)의 원소를 정렬해준다.
 - 기본 동작은 오름차순 정렬을 한다.
 - STL에서 제공하는 `greater<T>()` 와 같은 함수 객체를 사용해서 내림차순 정렬을 할 수 있다.

- **사용법**

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5      vector<int> v = {3, 1, 5, 4, 2};
6
7      sort(v.begin(), v.end()); // 오름차순 정렬
8      sort(v.begin(), v.end(), greater<int>()); // 내림차순 정렬
9
10 }
```

PS를 위한 C++ | 03 C++ STL (2) : sort

- 사용자 정의 정렬 (람다 표현식을 활용한 예제)
 - 람다 표현식은 이름 없는 함수(익명 함수)를 만드는 문법으로 짧고 간결하게 함수 기능을 구현할 수 있다.
[캡처](매개 변수){함수 본문} 형태인데, 저렇게 쓸 수 있구나 정도로 형태만 간단히 기억하면 된다.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5      vector<int> v = {3, 1, 5, 4, 2};
6
7      sort(v.begin(), v.end(), [](int a, int b){
8          return a > b; // 내림차순 정렬
9      });
10     return 0;
11 }
```

PS를 위한 C++ | 03 C++ STL (3) : binary_search

- binary_search는 이분 탐색 기능을 제공하는 함수이다.
 - 찾는 값의 존재 유무를 리턴한다. true / false만 반환한다.
 - 값의 위치를 찾으려면 lower_bound를 사용하면 된다. 다음 차시에 다룰 예정이다.

- **사용법**

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  using namespace std;
5
6  int main() {
7      vector<int> arr = {1, 3, 5, 7, 9};
8      cout << binary_search(arr.begin(), arr.end(), 3); // 1 (true)
9  }
```

PS를 위한 C++ | 03 C++ STL (4) : priority_queue

- C++에서는 힙을 기반으로 구현된 priority_queue가 기본 제공된다.
 - C에서는 힙을 직접 구현해야 한다.
 - 힙은 최댓값 및 최솟값을 찾아내는 연산을 빠르게 하기 위해 고안된 자료구조이다.

- **사용법**

```
1  √ #include <iostream>
2    #include <queue>
3    using namespace std;
4
5  √ int main() {
6      priority_queue<int> pq; // 기본적으로 최대 힙
7
8      pq.push(3);
9      pq.push(1);
10     pq.push(4);
11
12     cout << pq.top() << "\n"; // 4
13     pq.pop();                 // 가장 큰 값 제거
14
15     return 0;
16 }
```

PS를 위한 C++ | 03 C++ STL (5) : 순열

- next_permutation()는 사전순으로 다음 순열을 생성하는 함수이다.
- prev_permutation()는 사전순으로 이전 순열을 생성하는 함수이다.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5      ios::sync_with_stdio(false);
6
7      vector<int> a = {1, 2, 3, 4, 5};
8      next_permutation(a.begin(), a.end());
9      for(auto x : a) cout << x << " "; // 1 2 3 5 4
10     prev_permutation(a.begin(), a.end());
11     for(auto x : a) cout << x << " "; // 1 2 3 4 5
12     return 0;
13 }
```

PS를 위한 C++ | 03 C++ STL (5) : 순열

- next_permutation()는 사전순으로 다음 순열을 생성하는 함수이다.
- **사용법** 예) 모든 순열을 출력하기

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    ios::sync_with_stdio(false);

    vector<int> a = {1, 2, 3};

    do{
        for(auto x : a) cout << x << " ";
        cout << "\n";
    }while(next_permutation(a.begin(), a.end()));

    return 0;
}
```

출력값

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

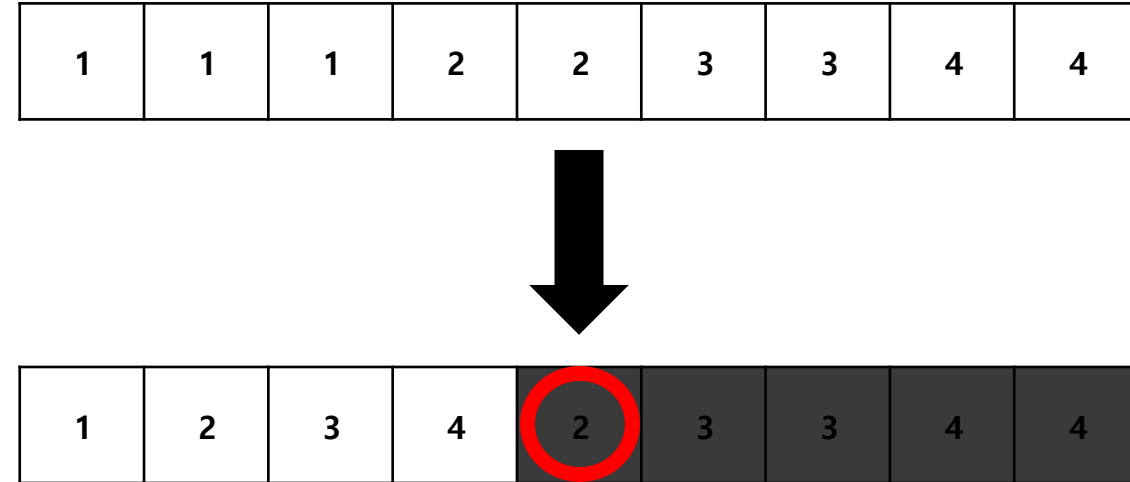
PS를 위한 C++ | 03 C++ STL (6) : 중복 제거

- 배열에서 중복된 값을 제거하기 위해 algorithm 헤더에 정의된 unique 함수와 vector 헤더에 정의된 erase 를 활용할 수 있다.
- unique 함수를 사용하기 전에 정렬을 먼저 수행해야한다.
- unique 함수는 연속된 중복을 하나의 값으로 압축하고, 새로운 끝을 반환한다.

PS를 위한 C++ | 03 C++ STL (6) : 중복 제거

- **사용법**

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5      ios::sync_with_stdio(false);
6
7      vector<int> v = {1, 1, 1, 2, 2, 3, 3, 4, 4};
8      v.erase(unique(v.begin(), v.end()), v.end());
9      for(auto x : v) cout << x << " "; // 1 2 3 4 출력함
10     cout << "\n";
11     return 0;
12 }
```



PS를 위한 C++ | 03 C++ STL (7) : 최대값, 최소값

- C 표준 라이브러리인 <stdlib.h>, <math.h> 에는 max 함수와 min 함수가 포함되어 있지 않다.
- C++ STL 에서는 max, min / max_element, min_element 함수를 제공한다.
- max, min 함수는 각각 최댓값과 최솟값을 반환한다.
- max_element, min_element 함수는 각각 최댓값과 최솟값의 이터레이터를 반환한다.
- **사용법**

```
7  ✓ int main() {  
8      vector<int> v = {3, 1, 4, 1, 5, 9, 2};  
9  
10     auto maxIt = max_element(v.begin(), v.end()); // 최댓값 위치 찾기  
11     auto minIt = min_element(v.begin(), v.end()); // 최솟값 위치 찾기  
12  
13     cout << "최댓값: " << *maxIt << "\n"; // *을 사용해 값 얻기  
14     cout << "최솟값: " << *minIt << "\n";  
15  
16     return 0;  
17 }
```

PS를 위한 C++ | 03 C++ STL (8) : pair, tuple

- C에서 구조체는 서로 다른 타입의 데이터를 하나로 묶어 사용하기 위해 사용된다.
- 이와 유사하게 C++에서는 pair, tuple를 제공한다.
 - pair는 두 개의 값만 저장 가능하고 first, second로 접근할 수 있다.
 - tuple은 세 개 이상의 값을 저장할 수 있고 get<index>(튜플 이름)으로 접근할 수 있다.

- **사용법**

```
1  #include <iostream>
2  #include <tuple>
3  using namespace std;
4
5  int main() {
6      pair<int, string> p = {1, "apple"};
7      cout << p.first << " " << p.second << "\n"; // 1 apple
8
9      tuple<int, double, string> t = {1, 2.5, "hello"};
10     cout << get<0>(t) << " " << get<1>(t) << " " << get<2>(t) << "\n"; // 1 2.5 hello
11 }
```

PS를 위한 C++ | 03 C++ STL (8) : pair, tuple

- 구조적 바인딩(Structured Binding, C++ 17 이상)을 통해 tuple, pair 등의 값을 쉽게 분해하여 변수에 할당할 수 있다.

```
8   tuple<int, double, string> t = {1, 2.5, "hello"};
9   auto [a, b, c] = t;
10  cout << a << " " << b << " " << c << "\n"; // 1 2.5 hello
11 }
```

PS를 위한 C++ | 03 C++ STL (8) : pair, tuple

- PS에서 단독으로 사용하는 경우는 거의 없고 주로 컨테이너(vector, queue, set, map 등)와 같이 사용한다.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5      queue<pair<int, int>> q;
6
7      q.push({1, 2});
8      q.push({3, 4});
9      while(!q.empty()){
10         auto [a, b] = q.front();
11         q.pop();
12         cout << a << " " << b << "\n";
13     }
14 }
```

PS를 위한 C++ | 03 STL C++ (9) : 스택, 큐, 덱

- 스택, 큐, 덱은 데이터를 저장하고 관리하는 방식을 정의하는 자료구조이다.
 - 스택 : 가장 나중에 삽입된 데이터가 가장 먼저 제거되는 자료구조 (LIFO : Last In First Out)
 - 큐 : 먼저 삽입된 데이터가 먼저 제거되는 자료구조 (FIFO : First In First Out)
 - 덱 : 스택과 큐의 특징을 모두 포함하는 자료구조
- C에서는 직접 구현해야하지만, C++에서는 이미 구현되어있다.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5      stack<tuple<string, string>> s;
6      s.push(make_tuple("안녕", "하세요"));
7      s.pop();
8  }
```

PS를 위한 C++ | 03 C++ STL (10) BST, Hash Table

- C++ STL에서는 이진 탐색 트리(BST, Red-Black Tree)와 해시 테이블(Hash Table)이 이미 구현되어 있다.
 - 이진 탐색 트리 : 정렬된 데이터를 빠르게 탐색할 수 있는 자료구조
 - 해시 테이블 : 키와 값의 쌍을 저장하는 자료구조로, 데이터를 빠르게 검색할 수 있도록 설계된 구조
- map: (key, value) 저장, 자동 정렬 ($O(\log N)$) // BST 기반
- unordered_map: (key, value) 저장, 정렬 없음, 빠른 탐색 ($O(1)$ 평균) // 해시 테이블 기반
- set: 중복 없는 원소 저장, 자동 정렬 ($O(\log N)$) // BST 기반
- unordered_set: 중복 없는 원소 저장, 정렬 없음, 빠른 탐색 ($O(1)$ 평균) // 해시 테이블 기반

재귀

재귀 | 01 재귀의 개념과 원리

- 재귀 함수란?

자기 자신을 호출하는 함수!!!

- 예) 피보나치 수열 : $F(n) = F(n - 1) + F(n - 2)$
- 예) 하노이 탑 : $T(n) = 2T(n - 1) + 1$
- 예) 이진 탐색 : $T(n) = T(n/2) + O(1)$
- 예) 팩토리얼 : $F(n) = F(n - 1)$

```
int fibo(int n){  
    if(n == 0 or n == 1) return n;  
    return fibo(n - 1) + fibo(n - 2);  
}
```

자기 자신을 호출함



매우 중요

재귀 | 01 재귀의 개념과 원리

매우 중요



- **재귀 호출(Recursive Call)**과 **종료 조건(Base Case)**

재귀 함수는 재귀 호출과 종료 조건이 있어야 한다.

- 재귀 호출

정의 : 함수 내부에서 자기 자신을 다시 호출하는 프로그래밍 기법.

팁 : 주어진 문제를 작은 문제로 나누고, 이를 반복적으로 해결하는 과정에서 자연스럽게 등장한다.

- 종료 조건

정의 : 재귀 호출이 멈추는 조건.

팁 : 보통 가장 작은 입력에 대해 직접 값을 반환한다.

```
int factorial(int n){  
    if(n == 1) 종료 조건 return 1;  
    return n * factorial(n - 1); 재귀 호출  
}
```

재귀 | 01 재귀의 개념과 원리

- 재귀와 반복문의 차이

- 재귀를 알아야 하는 이유는..

일반적인 for 문이나 while 문으로 해결할 수 없는 문제들이 존재. 재귀를 사용하면 간결하게 해결 가능.

예) 트리 탐색, 그래프 탐색(DFS), 백트래킹 등은 재귀 없이는 비효율적이거나 구현이 복잡해질 수 있음

```
void preorder_traversal(TreeNode *root){
    if(!root) return;
    stack<TreeNode*> s;
    s.push(root);

    while(!s.empty()){
        TreeNode *node = s.top();
        s.pop();
        cout << node -> val << " ";

        if(node->right) s.push(node->right);
        if(node->left) s.push(node->left);
    }
}
```



```
void Preorder_traversal(TreeNode *root){
    if(!root) return;
    cout << root->val << " ";
    Preorder_traversal(root->left);
    Preorder_traversal(root->right);
}
```

재귀 | 01 재귀의 개념과 원리

- 재귀와 반복문의 차이

- 재귀를 알아야 하는 이유는..

분할 정복(Divide & Conquer) 알고리즘의 핵심이기 때문에 알아야 한다.

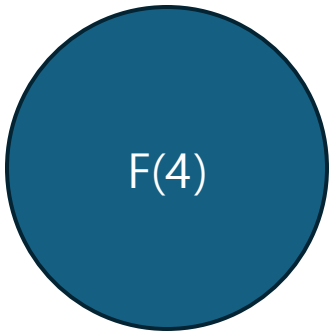
예) 머지 소트, 퀵 소트, 이진 탐색, 평면상의 가장 가까운 두 점, 빠른 거듭제곱

```
// 구간 [left, right] 을 정렬하는 함수
void mergeSort(vector<int> &arr, int left, int right){
    if(left >= right) return; // Base Case
    int mid = (left + right) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}
```

재귀 | 01 재귀의 개념과 원리

```
int fibo(int n){  
    if(n == 0 or n == 1) return n;  
    return fibo(n - 1) + fibo(n - 2);  
}
```

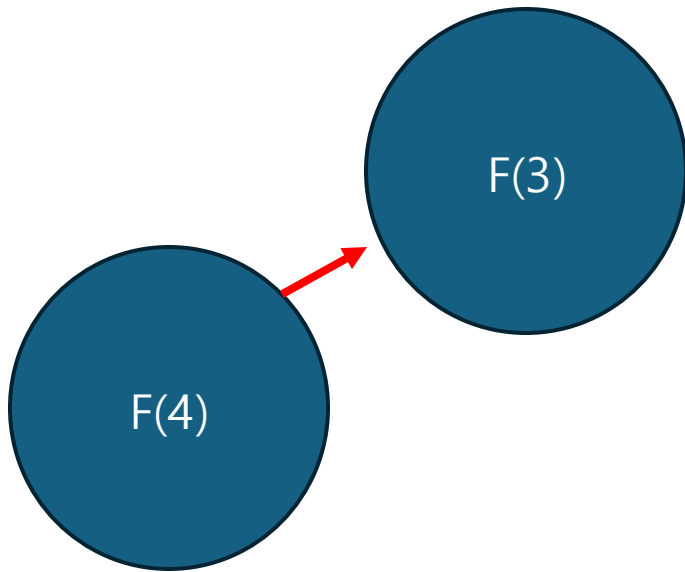
- 재귀 함수의 동작 방식 (예시, 피보나치 수열 : $F(n) = F(n - 1) + F(n - 2)$)
- $F(4)$ 를 호출하면, 종료 조건을 만족하지 못하므로 $F(n-1)$ 을 호출하게 된다.



재귀 | 01 재귀의 개념과 원리

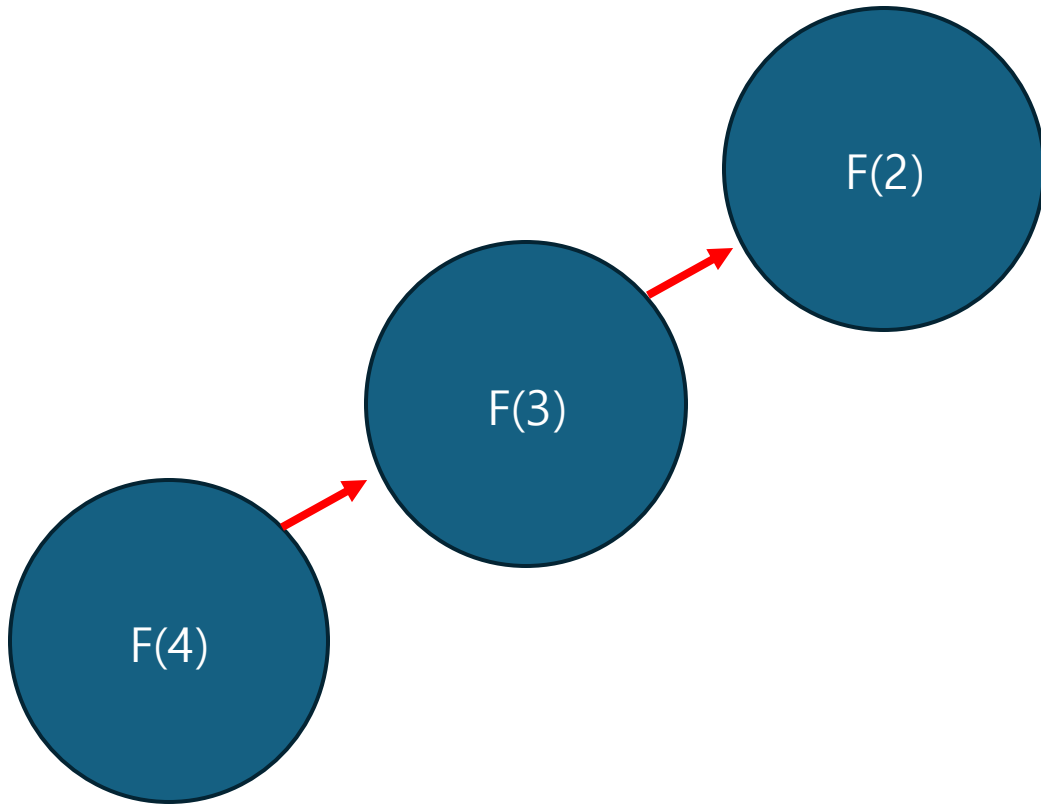
```
int fibo(int n){  
    if(n == 0 or n == 1) return n;  
    return fibo(n - 1) + fibo(n - 2);  
}
```

- 재귀 함수의 동작 방식 (예시, 피보나치 수열 : $F(n) = F(n - 1) + F(n - 2)$)
- $F(3)$ 역시 종료 조건을 만족하지 못하기에 $F(2)$ 를 호출한다.



재귀 | 01 재귀의 개념과 원리

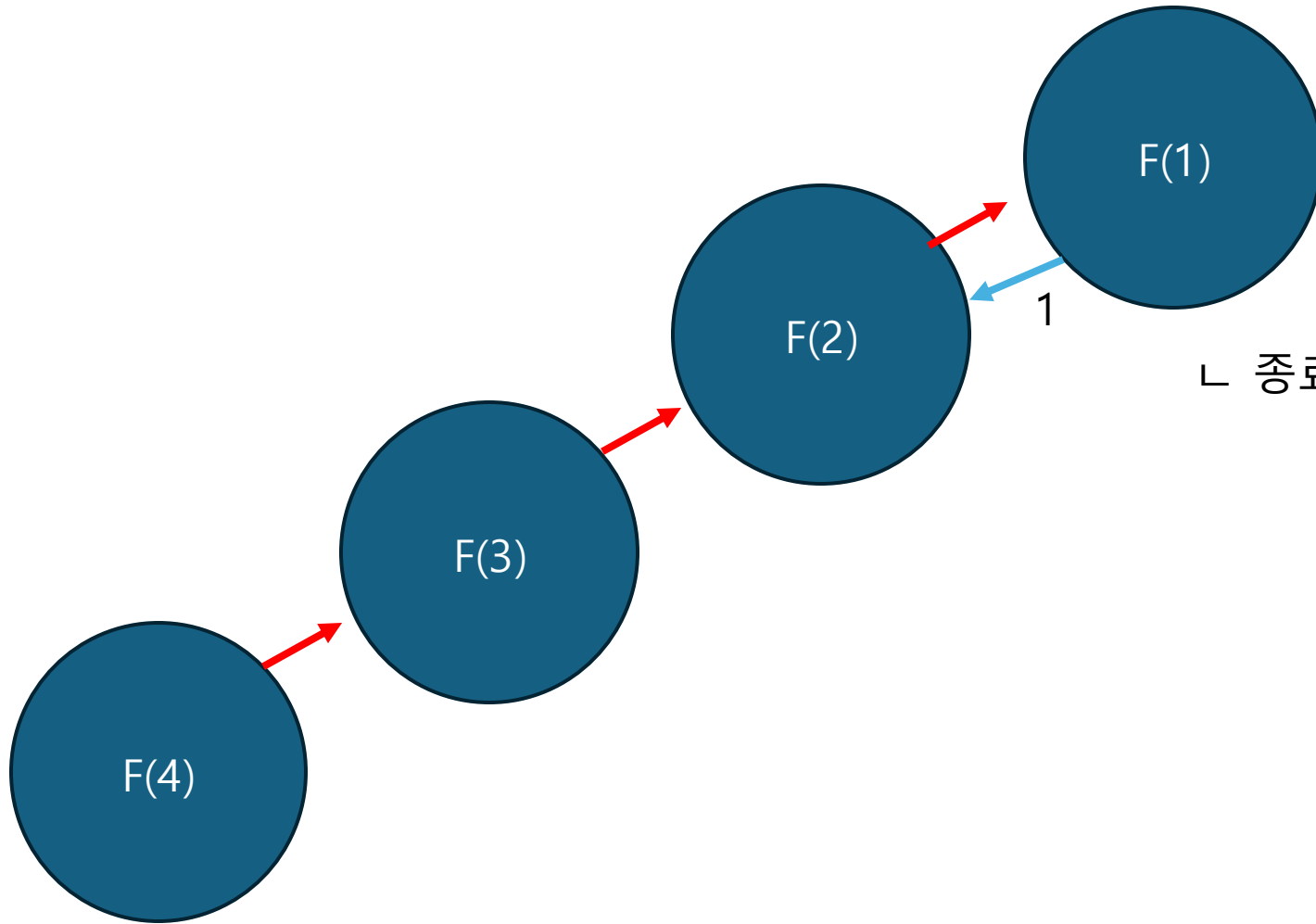
```
int fibo(int n){  
    if(n == 0 or n == 1) return n;  
    return fibo(n - 1) + fibo(n - 2);  
}
```



F(2)도 종료 조건 만족 못해서 F(1)를 호출한다.

재귀 | 01 재귀의 개념과 원리

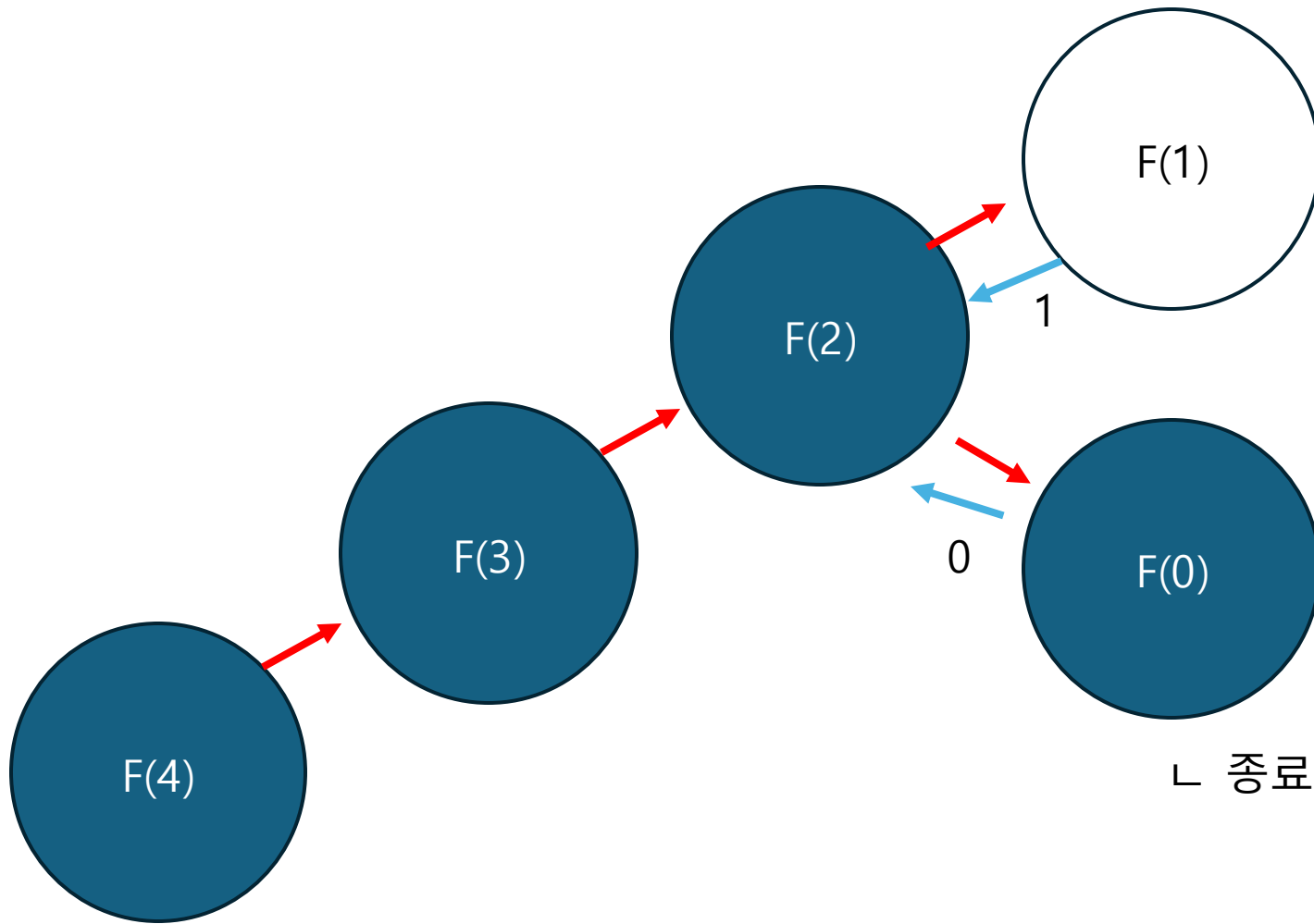
```
int fibo(int n){  
    if(n == 0 or n == 1) return n;  
    return fibo(n - 1) + fibo(n - 2);  
}
```



└ 종료 조건에 만족하므로 1 반환한다.

재귀 | 01 재귀의 개념과 원리

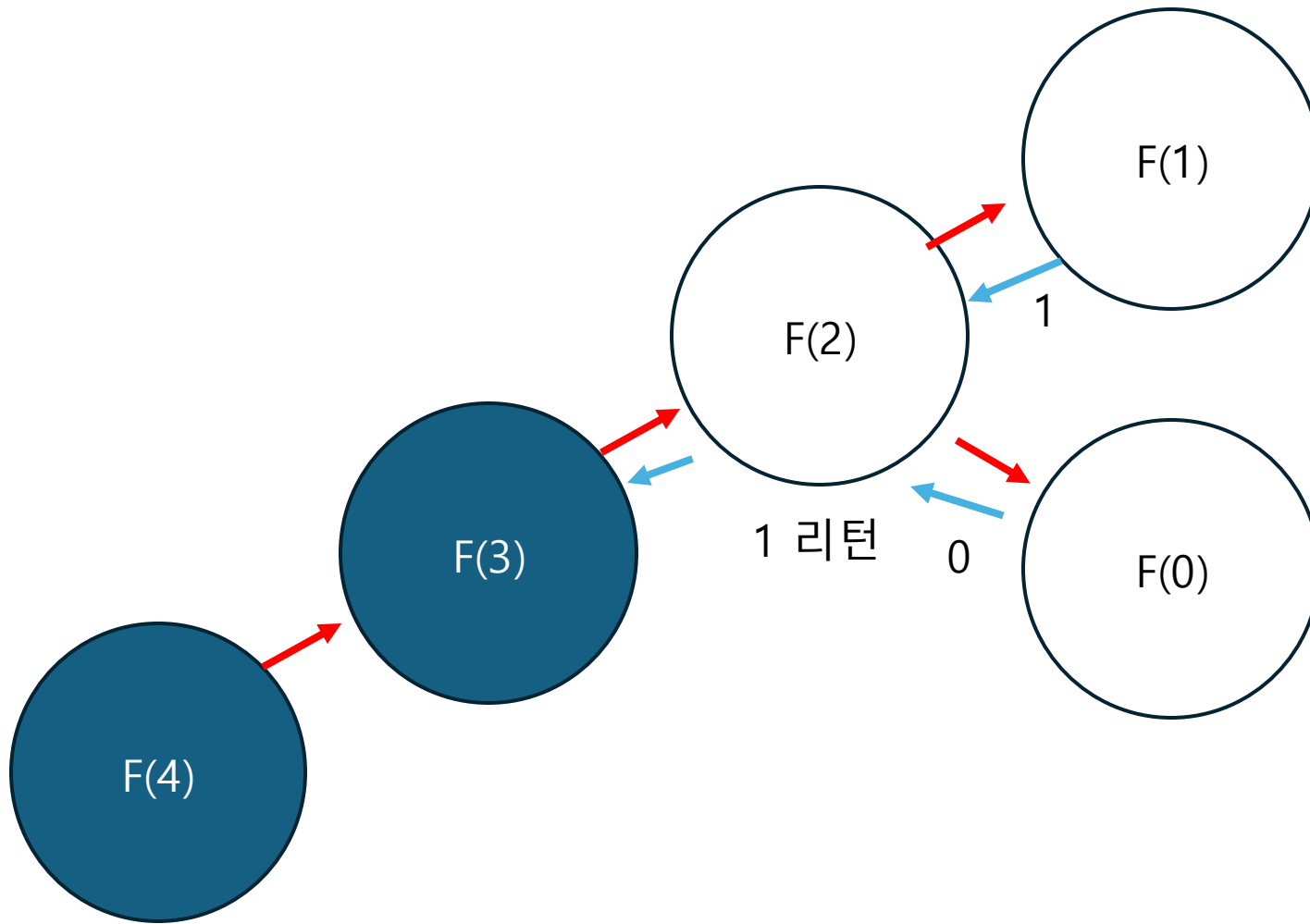
```
int fibo(int n){  
    if(n == 0 or n == 1) return n;  
    return fibo(n - 1) + fibo(n - 2);  
}
```



└ 종료 조건에 만족하므로 0 반환한다.

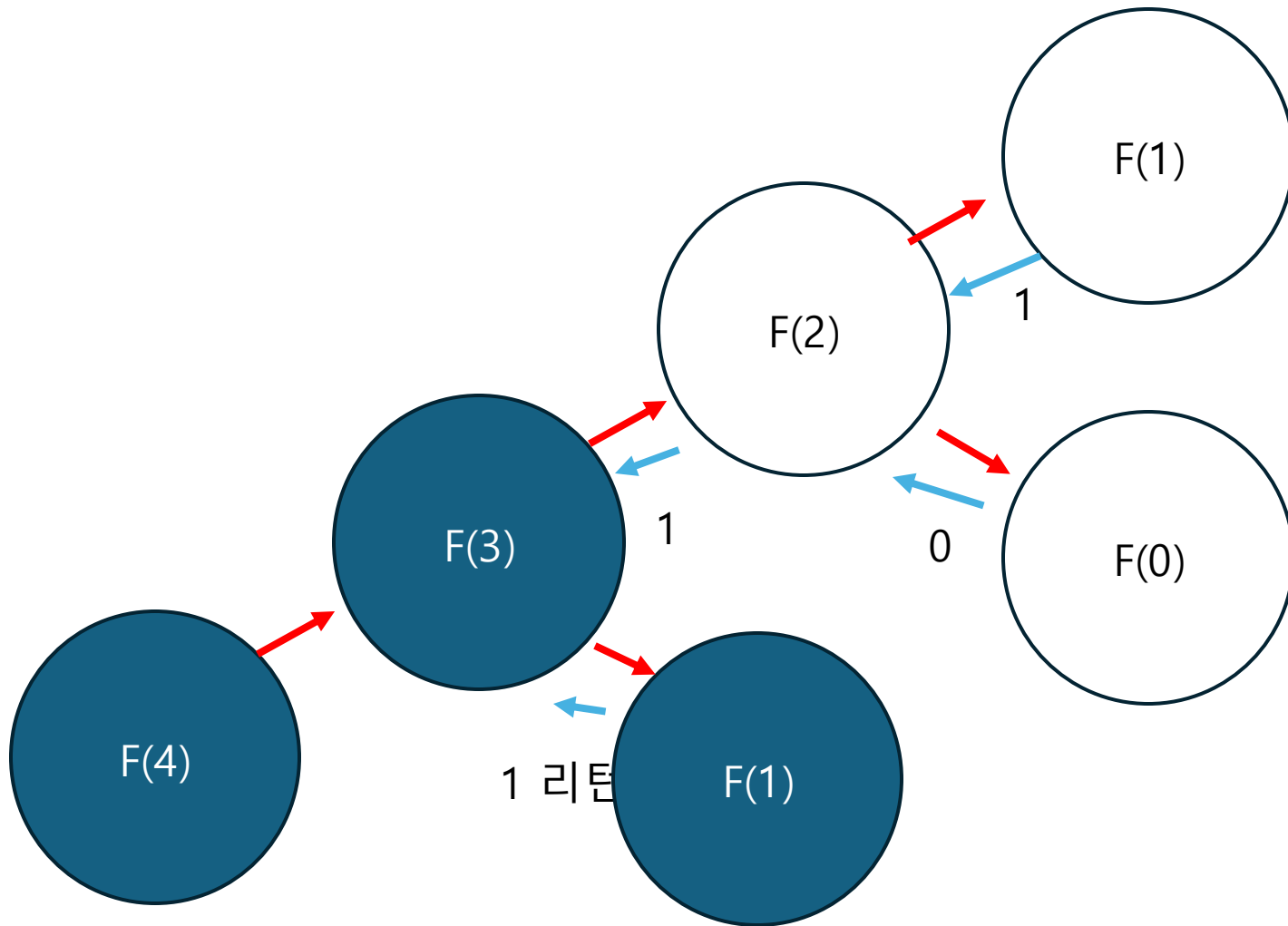
재귀 | 01 재귀의 개념과 원리

```
int fibo(int n){  
    if(n == 0 or n == 1) return n;  
    return fibo(n - 1) + fibo(n - 2);  
}
```



재귀 | 01 재귀의 개념과 원리

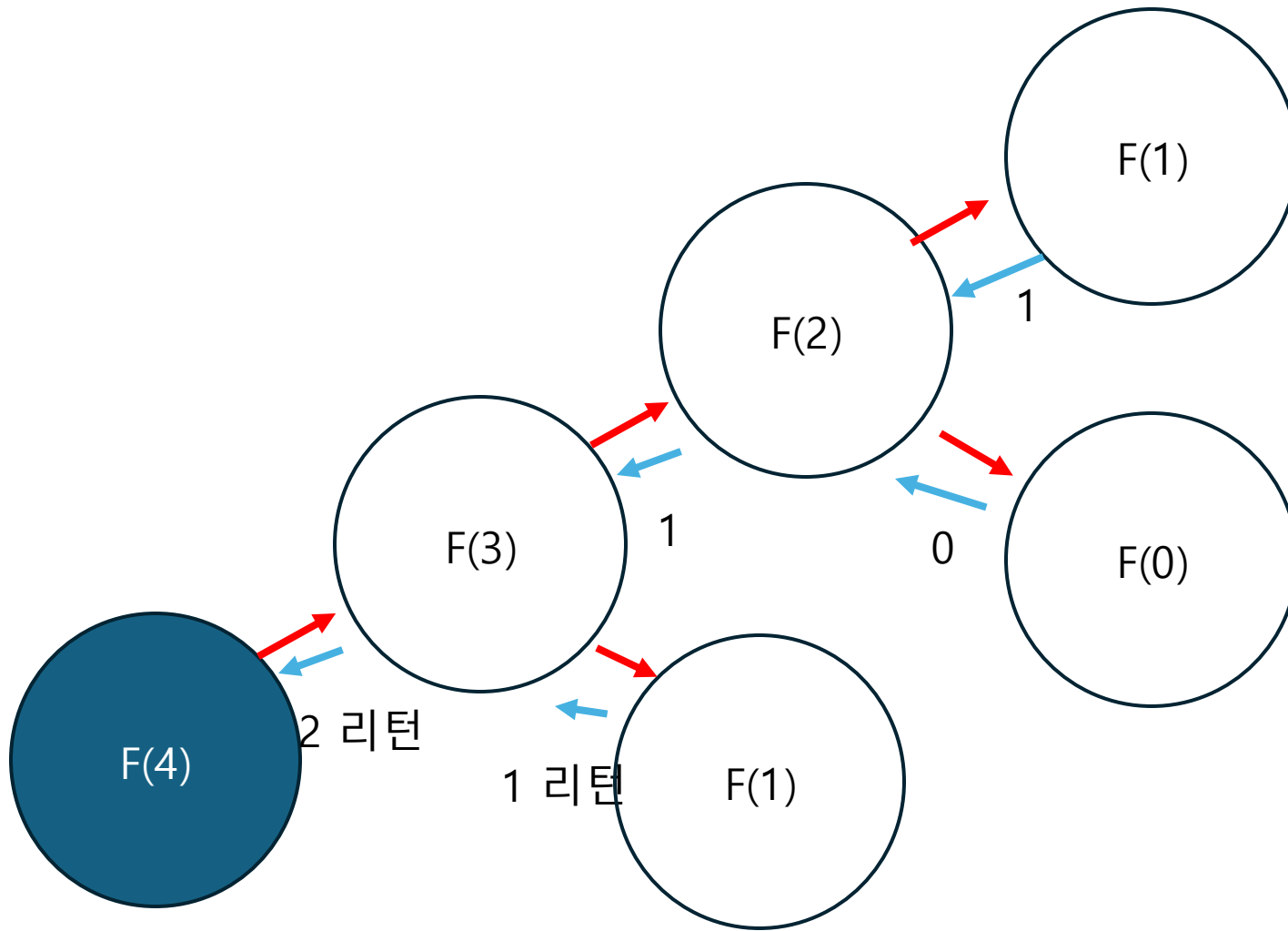
```
int fibo(int n){  
    if(n == 0 or n == 1) return n;  
    return fibo(n - 1) + fibo(n - 2);  
}
```



└ 종료조건을 만족하므로 1을 반환한다. $F(3)$ 에는 함수 2개를 호출하는데 $F(n-1)$ 에 대응하는 $F(2)$ 은 앞에서 이미 계산했고 아직 $F(n-2)$ 을 계산하지 않았다는 점이 포인트!

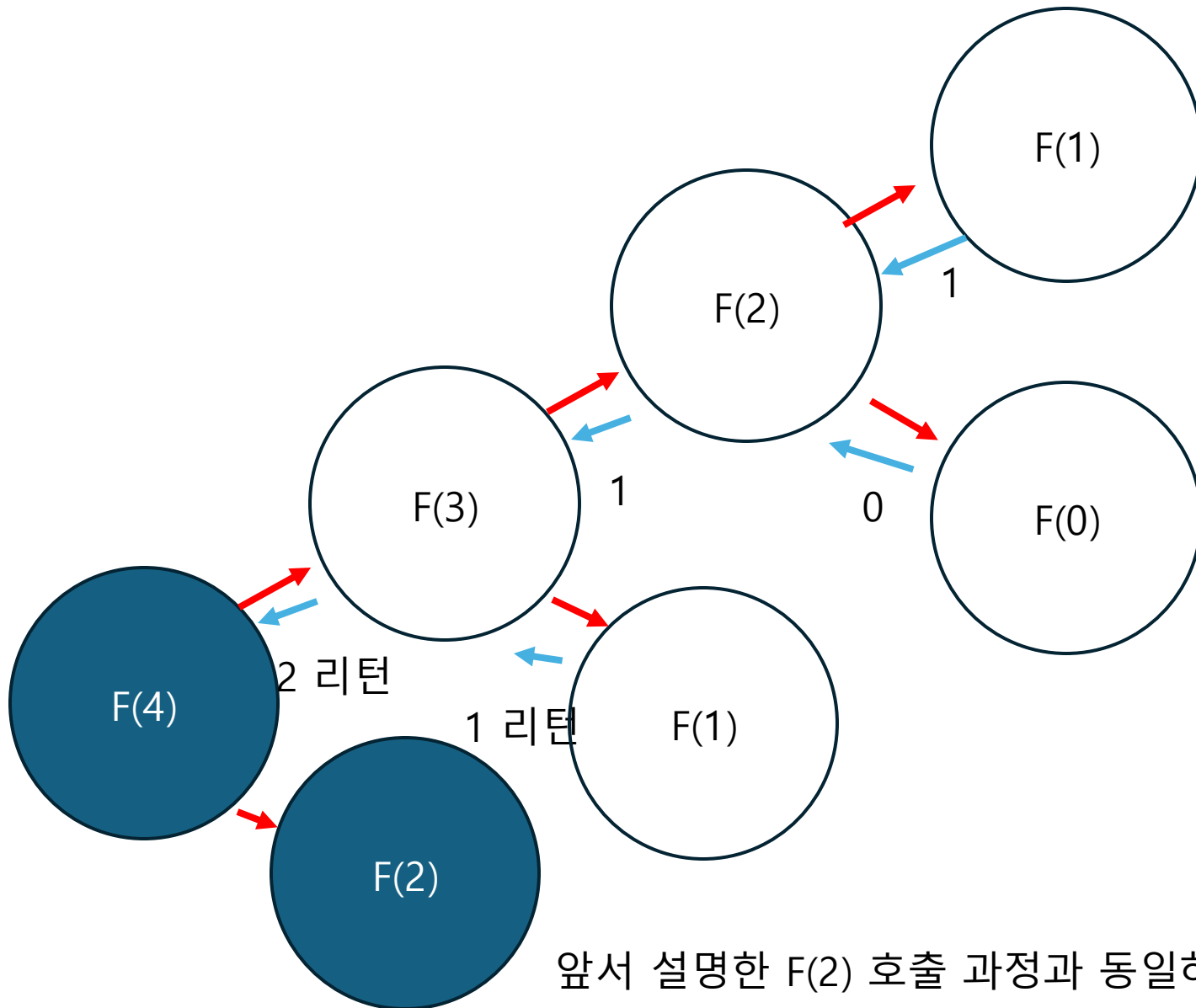
재귀 | 01 재귀의 개념과 원리

```
int fibo(int n){  
    if(n == 0 or n == 1) return n;  
    return fibo(n - 1) + fibo(n - 2);  
}
```



재귀 | 01 재귀의 개념과 원리

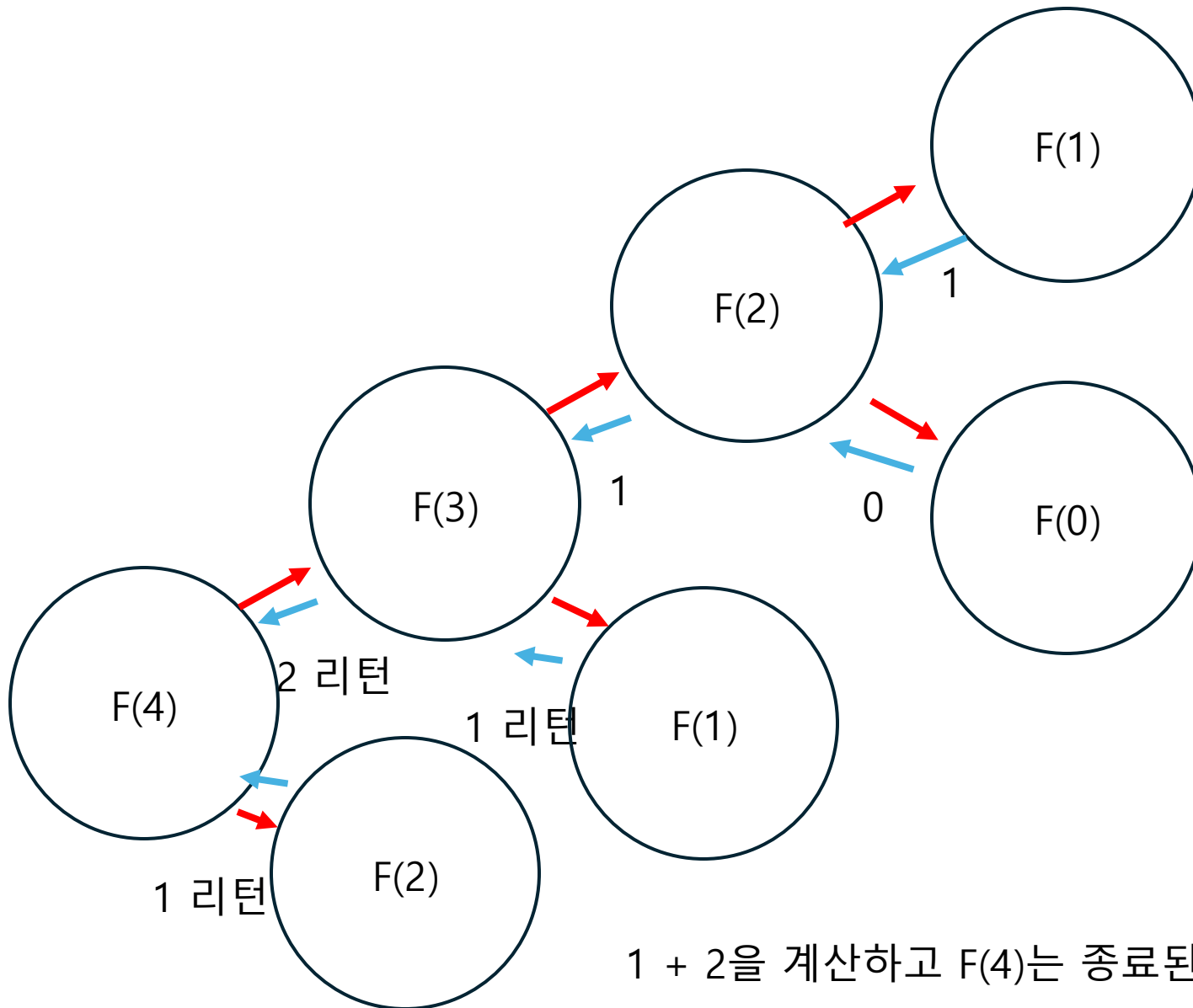
```
int fibo(int n){  
    if(n == 0 or n == 1) return n;  
    return fibo(n - 1) + fibo(n - 2);  
}
```



앞서 설명한 $F(2)$ 호출 과정과 동일하므로 생략하겠다. 1이 리턴된다.

재귀 | 01 재귀의 개념과 원리

```
int fibo(int n){  
    if(n == 0 or n == 1) return n;  
    return fibo(n - 1) + fibo(n - 2);  
}
```



재귀 | 02 기본적인 재귀 함수 구현

매우 중요

- 팩토리얼
$$n! = \begin{cases} 1, & \text{if } n = 0 \text{ or } n = 1 \\ n \times (n-1)!, & \text{if } n \geq 2 \end{cases}$$

- 문제를 작은 문제로 분할하여, 기저 조건에서 종료하도록 설계하면 된다.

팩토리얼은 $F(n) = n * F(n-1)$ 이라는 점화식(단, $F(0) = 1$)으로 표현되며 수학적 귀납법으로 증명할 수 있다.

<구현>

- 함수 $F(n)$ 은 $n!$ 을 반환하는 함수라고 믿는다(정의한다). // 문제를
- 그러면 $F(n-1)$ 은 $(n-1)!$ 을 반환해준다. // 작은 문제로 분할하여
- 가장 작은 입력은 직접 처리해준다. // 기저 조건에서 종료하도록 설계

```
int factorial(int n){  
    if(n == 0 or n == 1) return 1;  
    return n * factorial(n - 1);  
}
```



재귀 | 02 기본적인 재귀 함수 구현

- 피보나치

$$F(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F(n-1) + F(n-2), & \text{if } n \geq 2 \end{cases}$$

- 피보나치 수열은 이전 두 개의 숫자를 더해서 다음 숫자를 만드는 수열이다.

<구현>

- 함수 F(n)은 피보나치의 n번째 항을 반환한다고 믿는다(정의한다). // 문제를
- 그러면 F(n-1)은 n-1번째 항, F(n-2)은 n-2번째 항을 반환해준다. // 작은 문제로 분할하여
- 가장 작은 입력은 직접 처리해준다. // 기저 조건에서 종료하도록 설계

n 값	함수 호출 횟수
5	15
10	177
20	21,891
30	2,692,537
40	3,541,357,581

- 최적화?

```
int fibo(int n){  
    if(n == 0 or n == 1) return n;  
    return fibo(n - 1) + fibo(n - 2);  
}
```

피보나치 수열을 순수한 재귀 함수로 계산하면 중복 계산이 심각하게 많아져 비효율적이다. -> $O(2^n)$

Fibo(n)을 호출하면, 두 개의 하위 호출이 발생한다. 연산량이 지수적(2^n)으로 증가하게 된다.

시간 복잡도, 공간복잡도는 다음 차시에서 다룰 예정.

재귀 | 02 기본적인 재귀 함수 구현

- 최적화? (나중에 다룰 '동적 프로그래밍'의 주된 주제)

단순하게, 계산했으면 결과를 저장공간에 저장해두면 된다.

시간에서 이득을 챙기는 대신, 공간에서 손해를 보는 것이다.

```
int fibo(int n){  
    if(n == 0 or n == 1) return n;  
    return fibo(n - 1) + fibo(n - 2);  
}
```

시간복잡도 : $O(2^n)$

```
int result[101]; // -1로 초기화되어있다고 가정  
result[0] = 0, result[1] = 1;  
  
int fibo(int n){  
    if(result[n] != -1) return result[n];  
    return result[n] = fibo(n - 1) + fibo(n - 2);  
}
```

시간복잡도 : $O(n)$

공간복잡도 : $O(n)$

재귀 | 02 기본적인 재귀 함수 구현

- 피보나치 수열 (반복문)

피보나치 수열은 반복문으로도 계산 가능하다.

```
10  long long fibonacci(int n) {  
11      if (n == 0) return 0;  
12      if (n == 1) return 1;  
13  
14      long long a = 0, b = 1, result = 0;  
15      for (int i = 2; i <= n; ++i) {  
16          result = a + b;  
17          a = b;  
18          b = result;  
19      }  
20      return result;  
21 }
```

재귀 | 03 재귀 사용 시 주의할 점

- 재귀가 너무 깊어지면 stack overflow 가 발생할 수 있으므로, Base Case (종료 조건)를 명확히 설정하는 것이 중요하다.
 - 언어별로 허용되는 최대 재귀 깊이를 확인하면 된다.
- 메모이제이션을 활용하여 중복 연산을 방지하고 성능을 최적화할 수 있다.
- 반복문이 가능한 경우, 반복문이 성능 측면에서 유리할 수 있다.

재귀 | 04 예제 : 백준11729번 하노이 탑 이동 순서

- 문제 요약

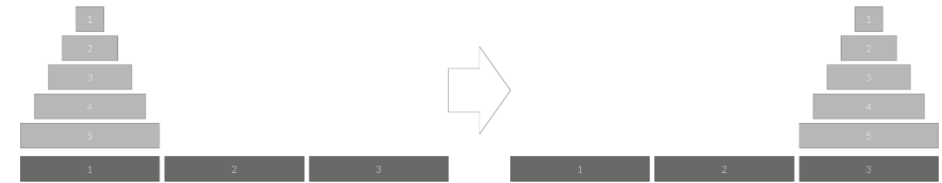
- 세 개의 기둥(A, B, C)이 있고, N개의 원판이 A 기둥에 쌓여 있음.
- 한 번에 하나의 원판만 이동 가능
- 큰 원판이 작은 원판 위에 올 수 없음.
- N개의 원판을 A -> C로 최소 횟수로 이동해야한다.

- 입력

- 첫 번째 장대에 쌓인 원판의 개수 N ($1 \leq N \leq 20$)이 주어진다.

- 출력

- 첫째 줄에 옮긴 횟수 k를 출력한다.
- 두 번째 줄부터 수행 과정을 출력한다. 두 번째 줄부터 k개의 줄에 걸쳐 두 정수 A B를 빈칸을 사이에 두고 출력하는데, 이는 A번째 탑의 가장 위에 있는 원판을 B번째 탑의 가장 위로 옮긴다는 뜻이다.



예제 입력 1 복사

3

예제 출력 1 복사

7
1 3
1 2
3 2
1 3
2 1
2 3
1 3

재귀 | 04 예제 : 백준11729번 하노이 탑 이동 순서

- N개의 원판을 기둥 A에서 기둥 C로 어떻게 옮길까?

문제(N)를

- 가장 큰 원판부터 기둥 C로 옮기면 됨.



작은 문제(N - 1)로 분할하여

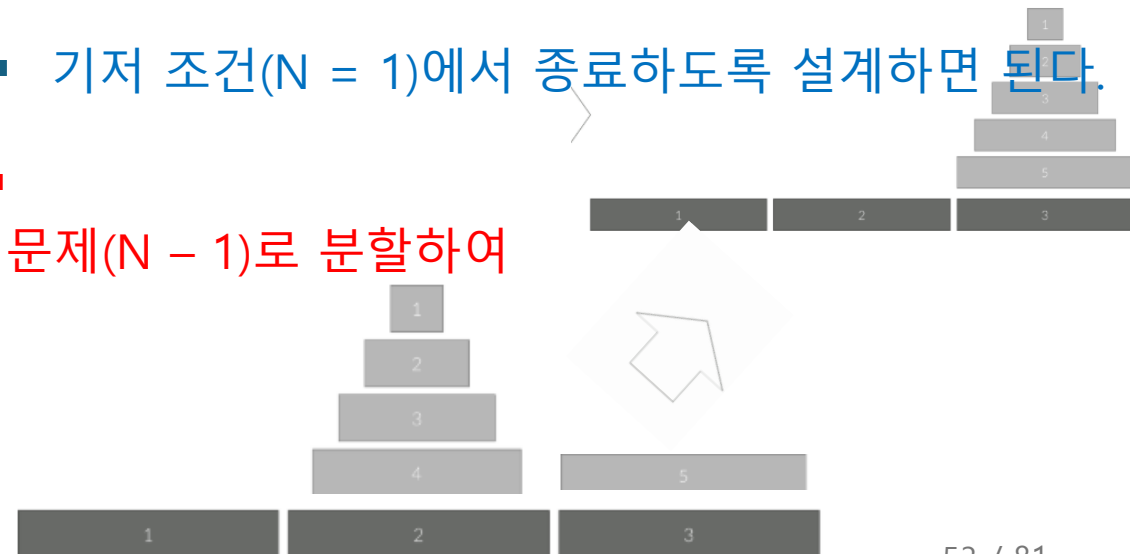
- 가장 큰 원판을 기둥 C로 옮기기 위해 N - 1개의 원판을 기둥 A에서 기둥 B로 옮기면 됨.

- N번째 원판을 기둥 A에서 기둥 C로 옮기면 됨.

기저 조건(N = 1)에서 종료하도록 설계하면 된다.

- N - 1개의 원판을 기둥 B에서 기둥 C로 옮기면 됨.

작은 문제(N - 1)로 분할하여



재귀 | 04 예제 : 백준11729번 하노이 탑 이동 순서

- $f(N, a, b, c)$: **N 개의 원판을 a 에서 c 로 옮기는 과정을 구하는 함수이다.**

$f(N-1, a, b, c)$: **$N-1$ 개의 원판을 a 에서 b 로 옮기는 과정을 구하는 함수이다.**

$f(1, a, b, c)$: **1개의 원판을 a 에서 c 로 옮기는 과정을 구하는 함수이다.**

$f(N-1, b, a, c)$: **$N-1$ 개의 원판을 b 에서 c 로 옮기는 과정을 구하는 함수이다.**

- 가장 작은 입력($N = 1$)은 직접 처리한다.
- 종료 조건은 $N = 1$ 일때이고 a 와 c 를 출력하고 종료함.
- 옮긴 횟수는 $2^N - 1$ 이다.



```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  void f(int n, int a, int b, int c){
5      if(n == 1){
6          cout << a << " " << c << "\n";
7          return;
8      }
9      f(n - 1, a, c, b);
10     f(1, a, b, c);
11     f(n - 1, b, a, c);
12     return;
13 }
14
15 int main(){
16     int n;
17     cin >> n;
18     f(n, 1, 2, 3);
19 }
```

비트마스킹

비트마스킹 | 01 개요

- 비트마스킹

비트마스킹은 정수의 이진수 표현을 이용하여 **집합을 다루는 기법**이다.

예를들어, 5 를 이진수로 나타내면 0000 0101 이다.

1이 있는 위치는 해당 원소가 포함됨을 의미한다.

0이 있는 위치는 해당 원소가 포함되지 않음을 의미한다.

long long int가 64비트를 가질 수 있으므로 최대 64개 원소의 표현이 가능하다.

비트마스킹 | 02 비트 연산자

- 비트 연산자

비트마스킹을 다루기 위해서는 비트 연산자를 알아야 한다.

비트 연산자는 정수의 이진수 표현을 직접 조작하는 연산자이다.

연산 속도가 빨라 알고리즘 문제 해결에 요긴하게 쓰인다.

특히, 비트로 집합을 나타내는 비트마스킹 기법에서 요긴하게 쓰인다.

ex) `INT64_MAX` 대신 `'(1LL << 63) - 1'` 사용, `INT64_MIN` 대신 `'-(1LL << 63)` 사용'

ex) 어떤 정수에 2를 곱하는 것 대신에 쉬프트 연산자 이용 `'a << 1'`

ex) 어떤 정수를 2로 나누는 것 대신에 쉬프트 연산자 이용 `'a >> 1'`

비트마스킹 | 02 비트 연산자

- 비트 연산자 ‘&’ 은 AND 연산을 한다. (이항 연산자)

전부 참(1)이어야지 참이다. 아닌 경우에는 거짓(0)이다.

비트마스킹에서 ‘집합에 특정 원소가 존재하는지 여부를 확인’ 할 수 있다.

char : 8 bits

0	1	0	1	0	1	0	1
---	---	---	---	---	---	---	---

& & & & & & & &

char : 8 bits

0	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

|| || || || || || || ||

결과값

0	1	0	1	0	1	0	1
---	---	---	---	---	---	---	---

비트마스킹 | 02 비트 연산자

- 비트 연산자 ‘|’ 은 OR 연산을 한다. (이항 연산자)

전부 거짓이면 거짓이다. 그렇지 않은 경우 항상 참이다.

비트마스킹에서 ‘집합에 특정 원소를 추가하는 기능’을 한다.

char : 8 bits

0	1	0	1	0	1	0	1
---	---	---	---	---	---	---	---

| | | | | | | |

char : 8 bits

0	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

|| || || || || || || ||

결과값

0	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

비트마스킹 | 02 비트 연산자

- 비트 연산자 ‘^’ 은 XOR 연산을 한다. (이항 연산자)

서로 다를 경우 참이다. 그렇지 않은 경우 거짓이다.

비트마스킹에서 ‘집합에 특정 원소를 토글하는 기능’을 한다.

char : 8 bits

0	1	0	1	0	1	0	1
---	---	---	---	---	---	---	---

^ ^ ^ ^ ^ ^ ^ ^

char : 8 bits

0	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

|| || || || || || || ||

결과값

0	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

비트마스킹 | 02 비트 연산자

- 비트 연산자 '~' 은 NOT 연산을 한다. (단항 연산자)

비트를 반전시키는 역할을 한다. (0 -> 1, 1 -> 0)

char : 8 bits

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

0

~

결과값

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

-1

비트마스킹 | 02 비트 연산자

- 비트 연산자 '<<' 은 비트를 n 칸씩 왼쪽으로 민다. (이항 연산자)

2^n 를 곱한 효과가 나타난다.

ex) $5 \ll 1 == 10$

char : 8 bits

0	0	0	0	0	1	0	1
---	---	---	---	---	---	---	---

5

<< 1

결과값

0	0	0	0	1	0	1	0
---	---	---	---	---	---	---	---

10

비트마스킹 | 02 비트 연산자

- 비트 연산자 '>>' 은 비트를 n 칸씩 오른쪽으로 민다. (이항 연산자)

2^n 로 나눈 효과가 나타난다.

ex) $5 \gg 1 == 2$

char : 8 bits

0	0	0	0	0	1	0	1
---	---	---	---	---	---	---	---

5

>> 1

결과값

0	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---

2

비트마스킹 | 03 비트 연산 활용

- 특정 비트 체크 : k번째 비트가 1인지 확인, 집합에 원소 유무를 확인

```
if (n & (1 << k)) {  
    cout << k << "번째 비트는 1입니다.";  
}
```

- 특정 비트 설정 (집합에 원소 추가)

```
n |= (1 << k); // k번째 비트를 1로 설정
```

- 모든 부분 집합 순회 (조합)

```
#define lld unsigned long long int  
for(lld bitset = 0; bitset <= (~0ULL); bitset++){  
    ...  
}
```


백트래킹

백트래킹 | 01 기본 개념

- 백트래킹이란?
 - 모든 가능한 경우를 고려하면서도 불필요한 탐색을 줄이는 탐색 기법이다.
 - 즉, 백트래킹은 탐색 과정에서 해당 경로가 유망하지 않다면 더 이상 진행하지 않고 되돌아가는 기법을 활용하여 탐색 공간을 줄이고 연산량을 감소시킨다.

백트래킹 | 01 기본 개념

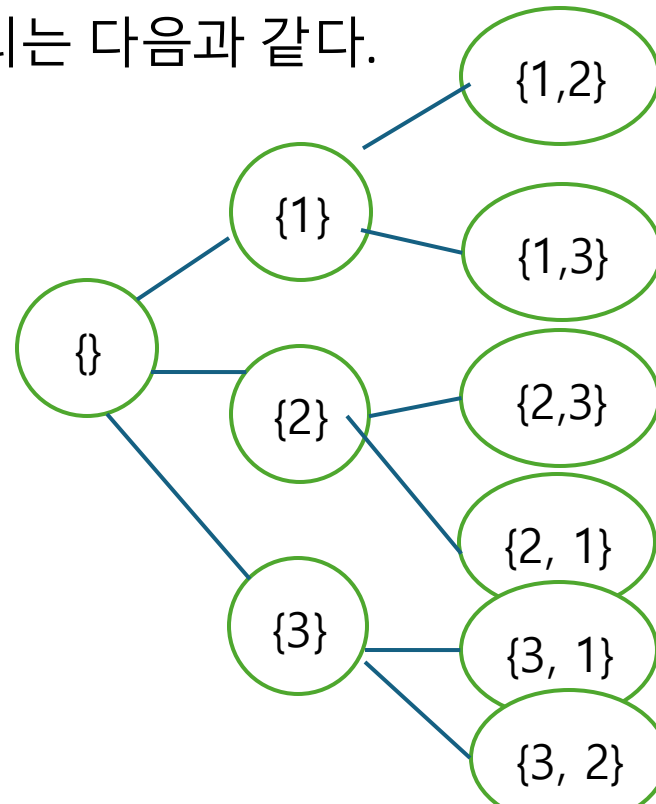
- 기본 원리

1. 모든 경우를 고려하며 탐색을 진행한다. (= 브루트 포스, 완전 탐색)
2. 탐색을 진행하던 중, 현재 상태가 해결책이 될 가능성이 없는 경우(불가능한 경로)라면 더 깊이 탐색하지 않고 가지치기(Pruning)를 수행하여 탐색을 중단하고 이전 단계로 되돌아간다. (백트래킹의 특징)
3. 위 과정을 반복하여 가능한 모든 해를 찾거나 최적의 해를 구한다.

백트래킹 | 01 기본 개념

- 상태 공간 트리(State Space Tree)

- 문제를 해결하기 위한 탐색 공간을 트리 형태로 표현한 구조이다.
- 예) 백준 15649번 N과 M (1) : 1부터 N까지 자연수 중에서 중복 없이 M개를 고른 수열 출력, N = 3, M = 2일 때의 상태 공간 트리는 다음과 같다.



백트래킹 | 01 기본 개념

- 가지치기(Pruning)의 개념과 필요성
 - 가지치기는 탐색 알고리즘에서 불필요한 탐색 경로를 조기에 차단하여 탐색 공간을 줄이는 기법이다.
 - 모든 경우를 탐색해야 하는 완전 탐색(Brute Force) 방식은 비효율적이므로, 불필요한 경로는 미리 배제하여 탐색 속도를 향상시키는 것이 가지치기의 핵심 원리이다.

백트래킹 | 02 구현 방법

- 백트래킹 기본 구조(형태)

```
void backtrack(현재 상태) {  
    if (정답 조건을 만족하면) {  
        정답 처리;  
        return;  
    }  
  
    for (모든 가능한 선택) {  
        if (유효한 선택인지 검사) { // 가지치기 (Pruning)  
            선택 수행;  
            backtrack(다음 상태); // 재귀 호출 (다음 단계 탐색)  
            선택 취소 (이전 상태로 되돌리기); // 백트래킹  
        }  
    }  
}
```

핵심 개념

1. 현재 상태를 기준으로 가능한 모든 선택을 시도
2. 유효하지 않은 경우 즉시 배제하여 탐색을 줄임
3. 유효한 경우에만 재귀 호출(다음 상태로 이동)
4. 탐색이 끝나면 선택을 취소하고 원래 상태로 복구



백트래킹 | 03 예제 (1) : 백준 15649번 N과 M (1)

- 문제 요약
 - 1부터 N까지의 자연수 중에서 중복 없이 M개를 고른 수열을 모두 출력하는 문제이다.
 - 수열은 사전 순으로 출력해야 한다.
 - 예) $N = 3, M = 2$
- 입력
 - 첫째 줄에 자연수 N과 M이 주어진다. ($1 \leq M \leq N \leq 8$)
- 출력
 - 한 줄에 하나씩 문제의 조건을 만족하는 수열을 출력한다.
 - 수열은 사전 순으로 증가하는 순서로 출력해야 한다.

예제 입력 2 복사

4 2

예제 출력 2 복사

1 2
1 3
1 4
2 1
2 3
2 4
3 1
3 2
3 4
4 1
4 2
4 3

백트래킹 | 03 예제 (1) : 백준 15649번 N과 M (1)

- **func(m) : 지금까지 선택한 수는 m개이다.**

- m = M 이면 출력하고 종료 (정답 처리)
 - m : 지금까지 선택한 수의 개수
 - M : 선택 해야하는 수의 개수
- v는 지금까지 선택한 수를 저장하는 벡터이다.
- vis[i]가 참이면, 숫자 i는 쓰였다는 의미이다. 거짓이라면, 안 쓰였다는 의미이다.

백트래킹의 기본 구조

```
void backtrack(현재 상태) {  
    if (정답 조건을 만족하면) {  
        정답 처리;  
        return;  
    }  
  
    for (모든 가능한 선택) {  
        if (유효한 선택인지 검사) { // 가지치기 (Pruning)  
            선택 수행;  
            backtrack(다음 상태); // 재귀 호출 (다음 단계 탐색)  
            선택 취소 (이전 상태로 되돌리기); // 백트래킹  
        }  
    }  
}
```

지금까지 선택한 수는 0개이다

```
#include <iostream>  
using namespace std;  
  
3  
4 int N, M;  
5 bool vis[10];  
6 vector<int> v;  
7  
8 void func(int m){  
9     if(v.size() == M){  
10         for(auto &x : v) cout << x << " ";  
11         cout << "\n";  
12         return;  
13     }  
14  
15     for(int i = 1; i <= N; i++){  
16         if(vis[i] == false){  
17             vis[i] = true;  
18             v.push_back(i);  
19             func(m + 1);  
20             v.pop_back();  
21             vis[i] = false;  
22         }  
23     }  
24     return;  
25 }  
26  
27 int main(){  
28     cin >> N >> M;  
29     func(0);  
30     return 0;  
31 }
```


백트래킹 | 03 예제 (1) : 백준 15649번 N과 M (1)

- 모든 가능한 선택을 시도
- 가지치기 : 유효한 선택인지 검사.
- 선택 수행
- 다음 단계 탐색

선택 취소

백트래킹의 기본 구조

```
void backtrack(현재 상태) {  
    if (정답 조건을 만족하면) {  
        정답 처리;  
        return;  
    }  
  
    for (모든 가능한 선택) {  
        if (유효한 선택인지 검사) { // 가지치기 (Pruning)  
            선택 수행;  
            backtrack(다음 상태); // 재귀 호출 (다음 단계 탐색)  
            선택 취소 (이전 상태로 되돌리기); // 백트래킹  
        }  
    }  
}
```

```
#include <iostream>  
using namespace std;  
  
3  
4 int N, M;  
5 bool vis[10];  
6 vector<int> v;  
7  
8 void func(int m){  
9     if(v.size() == M){  
10         for(auto &x : v) cout << x << " ";  
11         cout << "\n";  
12         return;  
13     }  
14  
15     for(int i = 1; i <= N; i++){  
16         if(vis[i] == false){  
17             vis[i] = true;  
18             v.push_back(i);  
19             func(m + 1);  
20             v.pop_back();  
21             vis[i] = false;  
22         }  
23     }  
24     return;  
25 }  
26  
27 int main(){  
28     cin >> N >> M;  
29     func(0);  
30     return 0;  
31 }
```

백트래킹 | 03 예제 (2) : 백준 9663번 N-Queen

- 문제 요약

- N-Queen 문제는 크기가 $N \times N$ 인 체스판 위에 퀸 N개를 서로 공격할 수 없게 놓는 문제이다.
- N이 주어졌을 때, 퀸을 놓는 방법의 수를 구하자.

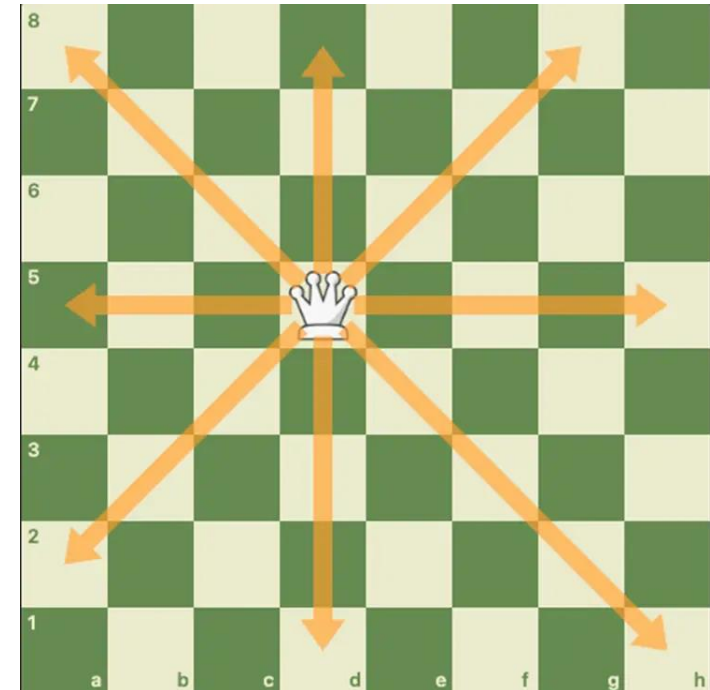
참고) 퀸은 같은 행, 열, 대각선 상에 있으면 공격 가능하다.

예제 입력 1 복사

8

예제 출력 1 복사

92



백트래킹 | 03 예제 (2) : 백준 9663번 N-Queen

- 퀸은 같은 행, 열, 대각선 상에 있으면 공격 가능하다.
 1. 첫번째 행부터 N개의 퀸을 배치
 2. 같은 열, 대각선에 다른 퀸이 있는지 검사
 3. 조건을 만족하면 다음 행으로 이동
 4. N개의 퀸을 모두 배치하면 경우의 수 증가
- 같은 열, 대각선에 다른 퀸이 있는지 검사하는 방법?
 - `bool col[N];` 라는 배열을 정의하자.
 - `col[a] = true`이면, a열에 퀸을 두지 못한다는 의미이다.



백트래킹 | 03 예제 (2) : 백준 9663번 N-Queen

- 같은 열, 대각선에 다른 퀸이 있는지 검사하는 방법
 - `bool col[N]; // col[a] = true`이면, 해당 열에 퀸을 두지 못한다.
 - `row + col`이 같은 값이면, '/' 대각선에 위치한다.
 - `row - col`이 같은 값이면, '\' 대각선에 위치한다. (음수 값 피하기 위해 같은 수를 더해주면 됨)



(row - col 값, \ 대각선)

plaintext							
(0,0)	0	(0,1)	-1	(0,2)	-2	(0,3)	-3
(1,0)	1	(1,1)	0	(1,2)	-1	(1,3)	-2
(2,0)	2	(2,1)	1	(2,2)	0	(2,3)	-1
(3,0)	3	(3,1)	2	(3,2)	1	(3,3)	0

→ 같은 숫자가 같은 \ 대각선

(row + col 값, / 대각선)

plaintext							
(0,0)	0	(0,1)	1	(0,2)	2	(0,3)	3
(1,0)	1	(1,1)	2	(1,2)	3	(1,3)	4
(2,0)	2	(2,1)	3	(2,2)	4	(2,3)	5
(3,0)	3	(3,1)	4	(3,2)	5	(3,3)	6

→ 같은 숫자가 같은 / 대각선

백트래킹 | 03 예제 (2) : 백준 9663번 N-Queen

중요?



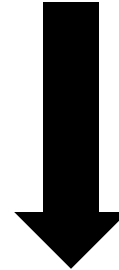
```
void backtrack(현재 상태) {  
    if (정답 조건을 만족하면) {  
        정답 처리;  
        return;  
    }  
  
    for (모든 가능한 선택) {  
        if (유효한 선택인지 검사) {  
            선택 수행;  
            backtrack(다음 상태); // 재귀 호출 (다음 단계 탐색)  
            선택 취소 (이전 상태로 되돌리기); // 백트래킹  
        }  
    }  
}
```

void func(int row) : 현재 row 개의 퀸을 놓은 상태

row == N이면 모든 퀸을 놓은 상태이므로 정답처리

모든 열에 퀸을 놓아보면서 유효한 선택인지 검사

놓으려는 위치에 열과 2개의 대각선에 퀸이 없으면 됨.



```
15    if(!col[c] && !diag1[row - c + N] && !diag2[row + c]){
```

백트래킹 | 03 예제 (2) : 백준 9663번 N-Queen

- Line. 9 ~ 12

N개의 퀸을 전부 놓았으면 경우의 수 증가시키고 종료

- Line. 14 ~ 20

15: 유효성 검사(가지치기)

16: 선택 수행

17: 다음 단계 탐색

18: 선택 취소

row를 지금까지 놓은 퀸의 개수라고 생각하면 된다.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int N, res;
5  bool col[15], diag1[30], diag2[30];
6
7
8  void func(int row){
9      if(row == N){
10         res++;
11         return;
12     }
13
14     for(int c = 0; c < N; c++){
15         if(!col[c] && !diag1[row - c + N] && !diag2[row + c]){
16             col[c] = diag1[row - c + N] = diag2[row + c] = true;
17             func(row + 1);
18             col[c] = diag1[row - c + N] = diag2[row + c] = false;
19         }
20     }
21 }
22
23 int main(){
24     cin >> N;
25     func(0);
26     cout << res << "\n";
27     return 0;
28 }
```



백트래킹 | 04 가지치기 유형

1. 유망성 검사

- 현재 상태에서 정답이 될 가능성이 있는지 검사하여, 가능성이 없으면 더 이상 탐색하지 않음.
- 예) 미로를 탈출할 때 벽이 막힌 공간을 굳이 탈출할 필요 없음.

2. 하한/상한 제한

- 가능한 해의 범위를 설정하여, 해당 범위를 벗어나면 탐색하지 않음.
- 예) N 개의 수가 담긴 배열에서 M 개를 선택해서 모두 더한 것들 중에서 k 보다 작으면서 가장 큰 조합을 고르기.
당연히 M 개를 선택하는 과정에서 k 보다 커지면 탐색을 중지해야함.

3. 최적해를 위한 가지치기

- 현재 상태에서 더 진행해도 지금까지 계산해낸 최적해보다 나빠지는 경우 탐색 종료
- 예) 경로 진행 중에 지금까지 구한 최적해보다 비용이 커지면 탐색할 필요가 없으므로 종료.

결국은, 불필요한 경로를 제거하는 전략을 고민하는 게 가지치기의 핵심

문제 | 재귀, 백트래킹 문제집 위주로 푸시면 됩니다!

10870번 피보나치 수 5

15649, 15650, 15651, 15652번 N과 M

1182 부분수열의 합

2580 스도쿠

9663 N-Queen

11729 하노이 탑 이동 순서

14888 연산자 끼워넣기

14889 스타트와 링크

추천 문제집

<https://www.acmicpc.net/workbook/view/7314>

<https://www.acmicpc.net/workbook/view/7315>

오늘 스터디는 여기서 마무리하겠습니다.

백준 많이 푸세요! 수고하셨습니다!

