Problem Solving 기초

경북대학교 전현승
dogdriip@gmail.com

Table of contents

- 0x00

Problem Solving?

Problem Solving?

THINK, CREATE, SOLVE

Problem Solving?

1. 문제를 해석, 내가 아는 언어로 재정의
2. 풀이 생각
3. 풀이 검증 - 내가 생각한 풀이가 정당한가? 주어진 범위에서 제한 시간 내에 작동하는가?
4. 구현 – 내가 생각한 풀이를 코드로 옮기는 과정
5. 정답 확인 - 맞았습니다! / 틀렸습니다 / 시간 초과 / 메모리 초과 / 런타임 에러 / …

“주어진 문제를 알고리즘과 자료구조를 이용해 창의적으로, 효율적으로 해결”

문제 분석

A+B

시간 제한	메모리 제한	제출	정답	맞은 사람	정답 비율
2 초	128 MB	264874	115186	84822	44.893%

문제

두 정수 A와 B를 입력받은 다음, A+B를 출력하는 프로그램을 작성하시오.

입력

첫째 줄에 A와 B가 주어진다. (0 < A, B < 10)

출력

첫째 줄에 A+B를 출력한다.

예제 입력 1 복사

```
1 2
```

예제 출력 1 복사

```
3
```

문제 분석

- 문제 제목
  - 힌트가 될 때도 있지만 그다지…
- **시간 제한, 메모리 제한**
- 정답자, 정답률 (대회 해당X)
  - 연습 시 문제 선택의 지표로 사용 가능

- **문제 본문**
- **문제에서 주어지는 수의 범위 (소위 'N제한')**
- 입/출력 형식
  - 반드시 문제에서 주어진 형식 그대로 입력받고, 출력해야 한다
- 예제
  - 예제 정도는 잘 돌아가는지 테스트해보고 제출하자

시간 제한, 메모리 제한?

- 우리가 작성한 프로그램이 **문제에 주어진 시간 제한과 메모리 제한을 지켜야 함**
  - 시간 제한이 2초인데, 입력에 100만을 넣었더니 5초가 걸렸다? → 시간 초과
  - 메모리 제한이 128MB인데, int 10^8개짜리 배열을 선언했다? → 메모리 초과
    (* int 한 개는 4바이트)

- 정확한 풀이도 중요하지만, 동시에 **효율적인 풀이**여야 함
- 코드를 작성하기 전에,
  내가 생각한 풀이가 시간 제한과 메모리 제한을 지키는지 생각, 분석, 검증이 꼭 필요
- *"Think twice, code once."*

- 0x01

시간 복잡도, 공간 복잡도

시간 복잡도

- 대략 이 알고리즘이 **“수행 시간이 어느 정도 걸리는가”**를 나타낼 수 있는 척도
- Big-O-Notation으로 나타냄
  - 계수와 최고차항 이외의 항을 모두 제외하고 표기
  - 영향을 미치는 변수가 여러 개일 때는 같이 표기
- 익숙하게 봤던 O(1), O(NlogN), $O(Q*N^2)$ 등의 표기법들

- 예를 들면?

시간 복잡도 – 간단한 예시 1

```
for (int i = 0; i < n; i++) {
    sum += a[i];
    mx = max(mx, a[i]);
}
```

N 크기 for문 안에서 상수시간 연산 2개 수행

→ O(N)

시간 복잡도 – 간단한 예시 2

```
for (int i = 0; i < n; i++) {
    sort(vec[i].begin(), vec[i].end());
}
```

C++ STL sort() 함수 : 시간 복잡도 O(NlogN)

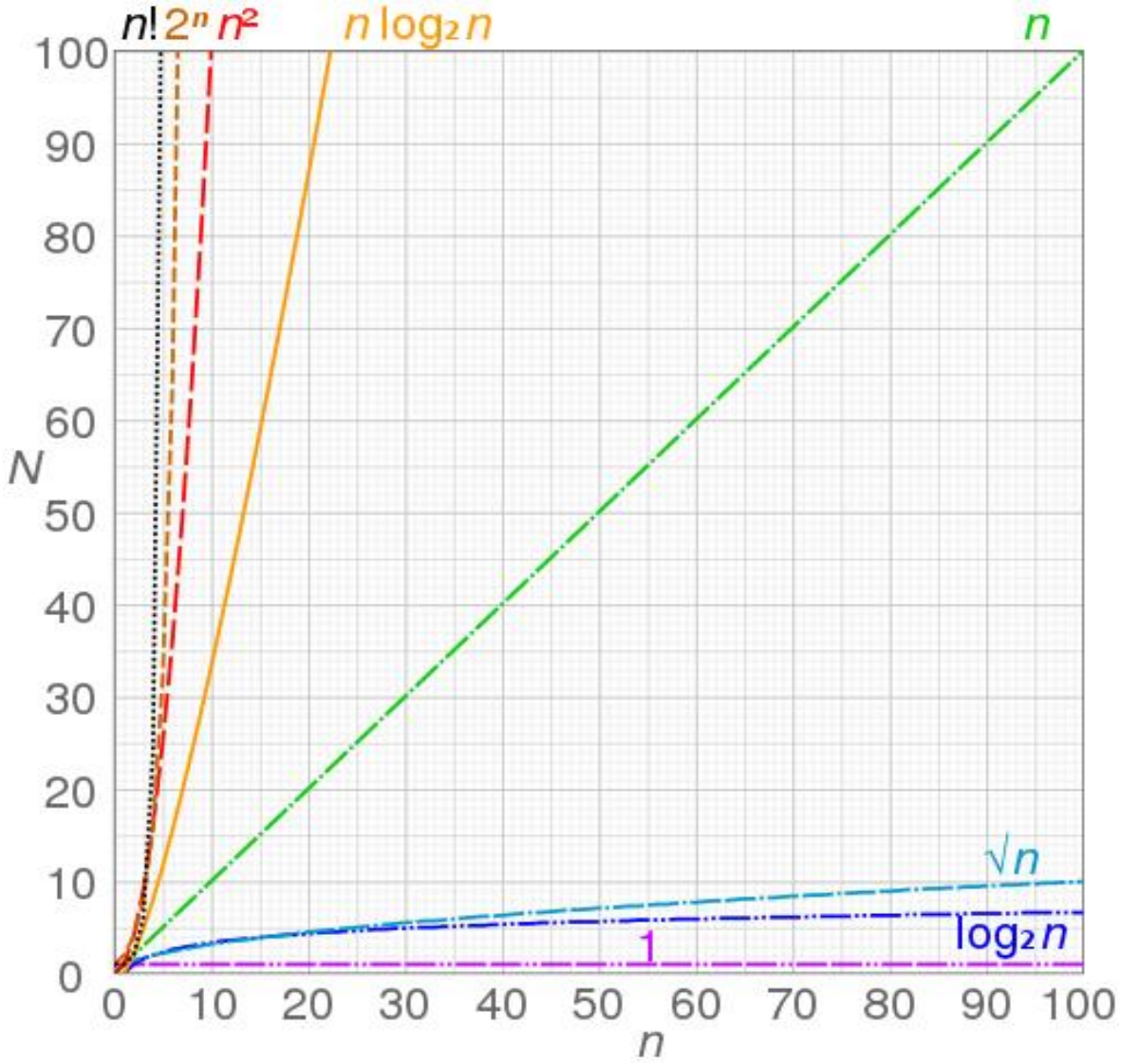
이를 vec[0]부터 vec[n – 1]까지 N번 반복하므로 → $O(N^2 \log N)$

시간 복잡도 – 간단한 예시 3

```
for (int i = 1; i < n; i++) {
    for (int j = 1; j <= k; j++) {
        if (j - coin[i] < 0) continue;
        dp[j] += dp[j - coin[i]];
    }
}
```

O(NK)

자주 쓰이는 시간 복잡도



- O(1) : 단순 계산 (+, *, % 등), 배열 접근 등
- O(logN) : 이분 탐색, 분할 정복 등 반으로 나눠가는 개념
- O(N) : 반복문 1개, 선형탐색 등
- O(NlogN) : 정렬 (머지소트, 퀵소트 등)
- $O(N^2)$, $O(N^3)$: 2중, 3중 반복문, 완전탐색
- $O(2^N)$: 크기가 N인 집합의 부분집합 개수
- O(N!) : 길이 N짜리 순열 채우기

시간 복잡도 분석 팁

- 대략 **1초에 1억 번** 연산이 가능하다고 생각하면 편하다
  - 아주 대략적 수치 – 모든 연산은 속도가 다르며, CPU마다 상이
  - 그래도 알아두면 엄청 편함
  - 최근에는 성능 향상 및 최적화로 단순 연산은 1억 번 넘게 가능한 것 같음

- 예) N=100,000 길이 배열을 정렬하라. (시간 제한 1초)
  - $O(N^2)$ 정렬 알고리즘을 사용하면? → TLE
  - O(NlogN) 정렬 알고리즘을 사용하면? → AC

- 우리가 작성한 프로그램이 시간 제한을 초과하는지 대강 판별 가능
- 하지만 맹신하지 말 것 – 시간 복잡도만이 프로그램 실행 시간을 결정하는 것은 아니다
  - 같은 1억 번 연산이라도 어떤 연산이냐에 따라…

공간 복잡도

- 대부분의 경우 계산하기 쉽다
- 기본 자료형의 크기를 잘 숙지하자
- 공간 복잡도가 문제되는 경우는 거의 없지만…

- 터무니없이 큰 크기의 배열을 잡는 경우
  - 메모리 제한 128MB → 5,000 * 5,000 int형 2차원 배열 정도가 한계
  - 생각 없이 10,000 * 10,000 배열을 선언했다간…
  - 조금 더 효율적인 풀이를 생각해야 할 것
- Queue와 같은 자료구조에서 계속 push만 하고 실수로 pop을 안 해주는 경우

공간 복잡도

- Tip : 크기가 큰 변수들은 전역변수로 선언하는 게 유리

- 전역변수는 Heap 영역에, 지역변수는 Stack 영역에 선언됨
- 프로세스당 기본 Stack 사이즈는 제한이 있다
  - Heap은 제한 없음
- 따라서 크기가 큰 변수들을 지역변수에 선언하면 프로그램이 터질 가능성이 있다

- 0x02

C++ 입출력

왜 C++?

- 알고리즘을 공부하는 데 있어 언어는 상관이 없다
  - 알고리즘은 프로그래밍 언어에 종속적이지 않다
- 그래도 알고리즘 문제풀이를 하려면, 대부분의 사람들이 C++을 추천
- 이유를 몇 개만 쓰자면…
- **빠르다** – Java, Python은 C++에 비해 느려요
  - 속도가 느린 언어에 시간 보너스를 주는 곳도 있지만 그건 출제자 마음
- **방대한 자료** – C++ 사용자가 대다수이다 보니 남의 코드 보기에도 편하고 여러 자료들을 이해하기 수월하다

- 주력 언어가 있으면 그걸 써도 무방하지만, 처음 공부하신다면? → C++
- 코드 설명들도 모두 C++일 예정

들어가기 전에 약간 팁

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

- `bits/stdc++.h`에는 많이 사용하는 표준 라이브러리들이 거의 모두 포함되어 있다
- 단 한 줄만으로 한 번에 다 불러올 수 있음
- 코드 길이는 줄어들지만, 불필요한 헤더가 포함되기에 컴파일 시간이 늘어난다
- 실행시간에는 지장 없음

- gcc 컴파일러 한정 (MS Visual Studio에서는 사용 불가. 따로 헤더 만들어줘야 함)

C++ 입출력

#1000 A+B

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

int main() {
    int a, b; cin >> a >> b;
    cout << a + b << endl;

    return 0;
}
```

C++ 입출력

#1000 A+B

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

int main() {
    int a, b; cin >> a >> b;
    cout << a + b << '\n';

    return 0;
}
```

- **endl**은 매번 출력 버퍼를 flush하기 때문에 매우 느리다
- 특히 반복해서 출력하는 문제에서 **endl**을 사용하면 시간 초과 가능성이 있음
- 따라서 개행은 **endl** 대신 **'\n'**을 사용하는 것을 추천

좀 더 빠른 C++ 입출력

#15552 빠른 A+B

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

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int a, b; cin >> a >> b;
    cout << a + b << '\n';

    return 0;
}
```

- 문제 본문에 설명이 다 돼 있다
- 기존 C 입출력 방식(**printf**, **scanf**, …)이랑 섞어 사용하지 않도록 주의

그 외에 다양한 입출력들

- 입력은 다양한 형식으로 들어올 수 있다
  - 개수가 주어지지 않고 그냥 입력이 끝날 때까지 계속 입력받아야 하는 경우
  - 공백을 무시하고 한 줄을 통째로 입력받아야 하는 경우
  - 공백이 아니라 쉼표로 구분되어 있는 경우

- 처음 공부할 때는, 다양한 문제를 풀어보면서
  입출력에서 10분 이상 막힐 경우, 그때그때 찾아보며 공부하는 것을 추천

그 외에 다양한 입출력들

- 추천 문제 – A+B 시리즈

  #10950 A+B - 3

  #10951 A+B - 4

  #10952 A+B - 5

  #10953 A+B - 6

  #11021 A+B - 7

  #11022 A+B - 8

  #15740 A+B - 9

- 0x03

C++ STL 기초

C++ STL 기초

- “C++로 알고리즘 문제풀이를 시작할 때 이 정도는 알아야 하지 않을까?”
- 물론 어떤 문제를 푸느냐에 따라, 여기에 없는데도 알아야 할 것들이 있을 수 있음
- 고급 주제로 들어갈수록 당연히 알아야 할 것들이 늘어남

C++ STL 기초

- STL (Standard Template Library)
  : C++에서 기본으로 제공하는 표준 라이브러리의 일부.
- 우리가 배울 자료구조들과 대표적인 알고리즘들이 일부 구현되어 있다
  - **큐, 스택** 같은 기초 자료구조부터 **연관 배열, 해시 테이블,** bitset, …
  - O(NlogN) 보장 정렬 함수 sort(), 이분 탐색 lower_bound(), upper_bound(), …

- 기초적인 STL과 기본적인 연산들의 시간 복잡도를 알아보자

그런데 잠깐!

- STL을 생각 없이 막 써도 되나요? 직접 구현하는 것에 의미가 있는 것 아닌가요?

- 물론 각 자료구조와 알고리즘이 어떻게 구현되어 있는지 **원리를 아는 것도 중요**하지만
- **PS/CP에서는 아는 것들을 어떻게 잘 응용할 줄 아는지가 더 중요**
- 문제를 빠르고 정확하게 풀어야 한다.
  검증된 STL을 이용해서 빠르게 풀면 그만이다.
  문제를 풀면서 밑바닥부터 직접 구현할 필요가 없다.

그런데 잠깐!

- 물론 STL이 제한된 환경도 있지만,
  대부분의 환경에서는 표준 라이브러리를 지원하므로
  주의사항을 숙지하며 사용하면 된다

구분	검정시간	지원언어	사용가능한 라이브러리	샘플문제	추천 연습문제
A형	3시간	C/C++/Java/Python	제한 없음	풀어보기	D2~4
B형	4시간	C/C++/Java	라이브러리 사용 불가 (단, C언어의 경우 동적할당을 위한 <malloc.h> 가능)	풀어보기	D4~6
C형	4시간	C/C++	라이브러리 사용 불가 (단, C언어의 경우 동적할당을 위한 <malloc.h> 가능)	풀어보기	D5~7

vector<T>

- T에는 자료형이 들어간다. → vector<int>, vector<string> 등…
- **#include <vector>**
- vector<T> : 가변 배열, 동적 배열.
  - 임의 원소 접근, 맨 뒤에 원소 추가 : O(1)
  - 임의 위치에 원소 추가, 임의 원소 삭제 : O(N) 주의!

```
vector<int> v;
v.push_back(10);
v.push_back(20);
cout << v.size() << '\n';        // 2
v.push_back(30);
cout << v.size() << '\n';        // 3
cout << v[0] << '\n';            // 10
cout << v.empty() << '\n';       // 0 (false)
v.clear();
```

sort(), stable_sort()

- `#include <algorithm>`

- sort(시작 iterator, 끝 iterator, (비교함수))
- sort(배열 시작 주소, 배열 끝 주소, (비교함수))
- O(NlogN)을 보장

- 기본적으로 sort()는 unstable sort이고,
  stable sort (정렬 후에도 동일한 원소의 원래 순서가 유지됨) 함수인 stable_sort()도 제공
- 사용법은 위와 동일
- O(NlogN) ~ O(Nlog(N)^2)

sort(), stable_sort()

```
vector<int> v = {4, 2, 5, 3, 1};
sort(v.begin(), v.end());                    // {1, 2, 3, 4, 5}가 됨
sort(v.begin(), v.end(), greater<int>());    // {5, 4, 3, 2, 1}가 됨
```

- Tip : STL에는 유용하게 쓸 수 있는 함수 객체들도 포함되어 있다
- `#include <functional>`
- 비교함수 자리에 `greater<T>()` 을 넣으면 내림차순 정렬도 바로 가능

pair<T, T>

- #include <utility>

- pair : 2개의 데이터를 묶어서 관리할 수 있는 자료형

```
pair<int, int> p;
p = make_pair(4, 6);                    // (4, 6)을 p에 저장

cout << p.first << ‘ ’ << p.second;     // 4 6

p.first = 46;
p.second = 100;                         // 직접 값 수정 가능
```

pair<T, T>

- 기본적으로 비교 연산 시 first로 먼저 결정. first가 같다면 second로 결정

```
pair<int, int> p[5];
p[0] = make_pair(100, 1);
p[1] = make_pair(10, 120);
p[2] = make_pair(5, 120);
p[3] = make_pair(5, 1);
p[4] = make_pair(50, -100);

sort(p, p + 5); // 기본 비교연산으로 pair<int, int>형 배열을 정렬하면?

for (int i = 0; i < 5; i++) {
    cout << "(" << p[i].first << "," << p[i].second << ") ";
}
// 출력 : (5,1) (5,120) (10,120) (50,-100) (100,1)
```

tuple<T, T>

- #include <tuple>

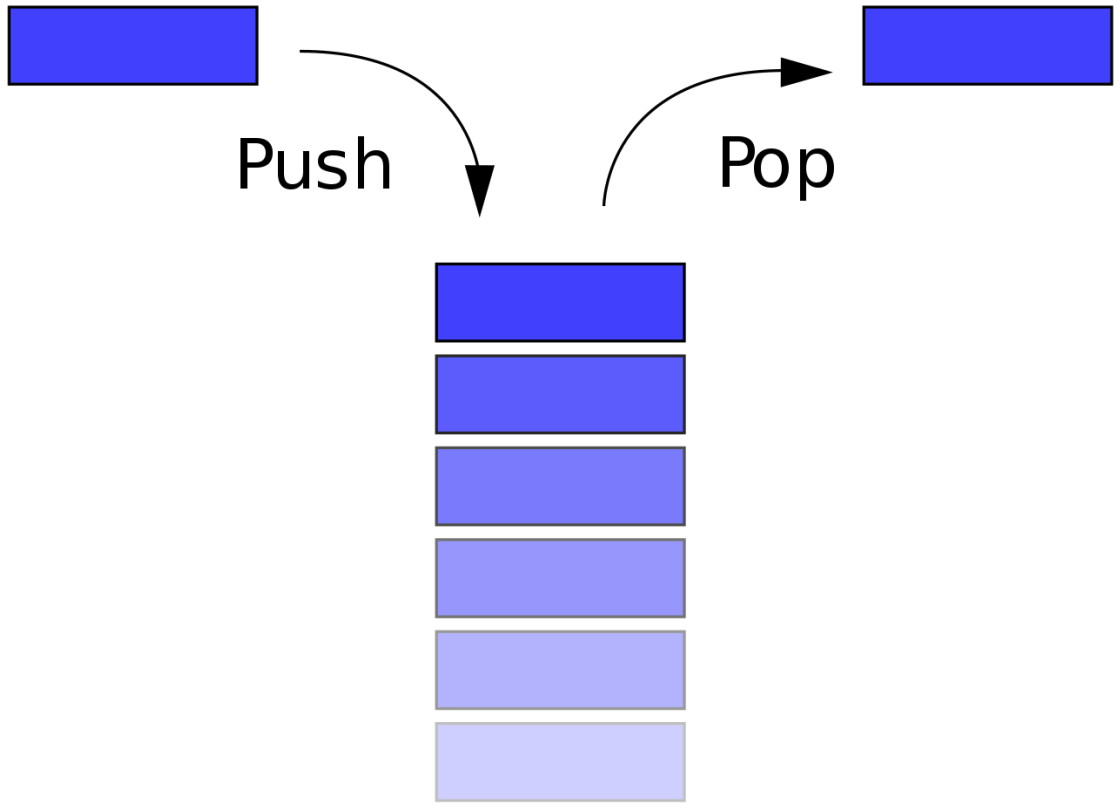
- tuple : 2개 이상의 데이터를 묶어서 관리할 수 있는 자료형

```
tuple<int, string, double> tp;
tp = make_tuple(46, “Best gardener”, 4.6);

int first = get<0>(tp);       // first = 46
string second = get<1>(tp);   // second = "Best gardener"
double third = get<2>(tp);    // third = 4.6
```

stack<T>

- 스택 : First-In-Last-Out 자료구조



stack<T>

- #include <stack>

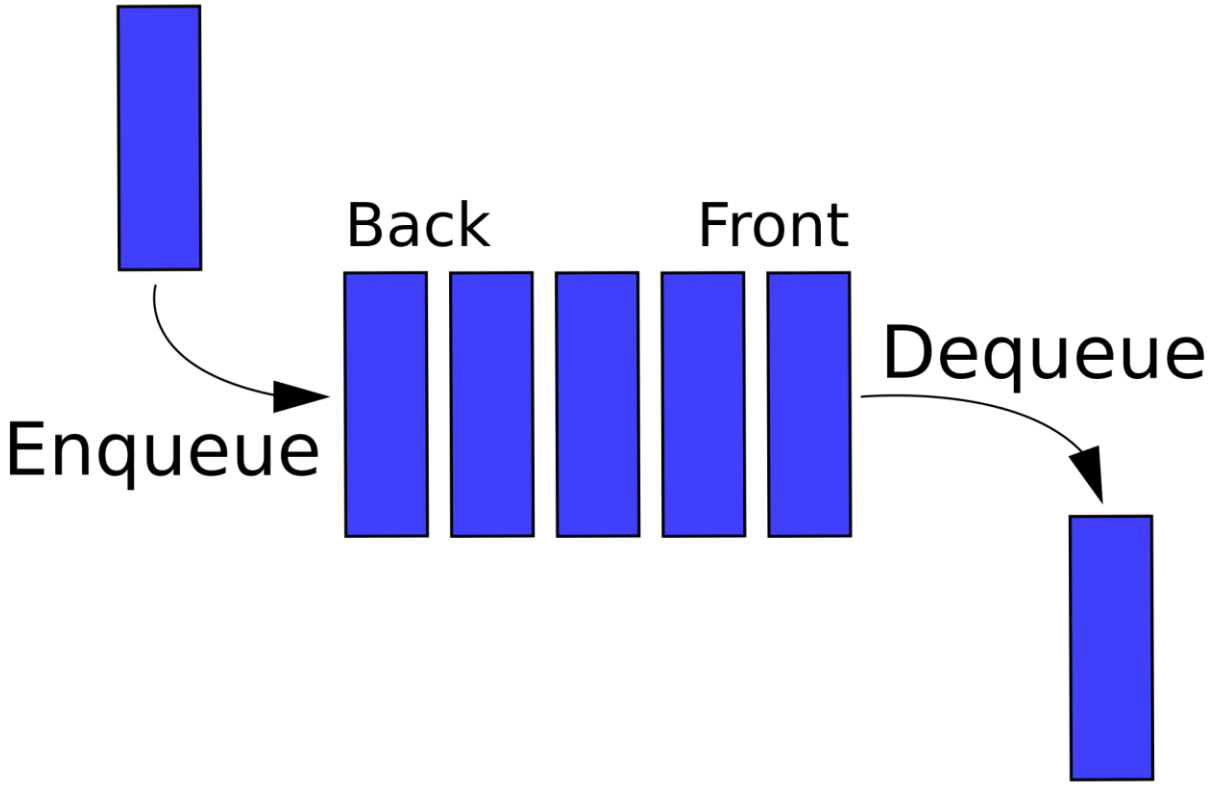
```
stack<int> st;

st.push(1);           // 스택에 1 추가
st.push(2);           // 스택에 2 추가
int a = st.top();     // 스택의 제일 위에 있는 값 반환 (여기서는 2)
st.pop();             // 스택의 제일 위에 있는 값 삭제
cout << st.size();    // 스택의 크기 (데이터 수) 반환

if (st.empty()) {
    cout << "스택이 비었다!\n";
} else {
    cout << "스택에 뭔가 있다!\n";
}
```

queue<T>

- 큐 : First-In-First-Out 자료구조



queue<T>

- #include <queue>

```
queue<int> q;

q.push(1);          // 큐에 1 추가
q.push(2);          // 큐에 2 추가
int a = q.front();  // 큐의 제일 앞에 있는 값 반환 (여기서는 1)
q.pop();            // 큐의 제일 앞에 있는 값 삭제
cout << q.size();   // 큐의 크기 (데이터 수) 반환

if (q.empty()) {
    cout << "큐가 비었다!\n";
else {
    cout << "큐에 뭔가 있다!\n";
}
```

deque<T>

- 덱 (Deque = Double-ended Queue)
- 말 그대로 **양쪽에서 push, pop 모두 가능한 큐**
- 양쪽으로 열린 큐라고 생각하면 된다
- 메소드도 거의 다른 게 없음

- `#include <deque>`
- `push_front(), push_back()`
- `pop_front(), pop_back()`
- `front(), back()`

Practice

#2557 Hello World

#11718 그대로 출력하기

#16430 제리와 톰

#17496 스타후르츠

#3009 네 번째 점

#10828 스택

#10845 큐

#10866 덱

#2751 수 정렬하기 2

#16435 스네이크버드

#11650 좌표 정렬하기

Sources

- http://icpckorea.org/ (로고 이미지)
- https://commons.wikimedia.org/wiki/File:Comparison_computational_complexity.svg
- https://commons.wikimedia.org/wiki/File:Data_stack.svg
- https://commons.wikimedia.org/wiki/File:Data_Queue.svg