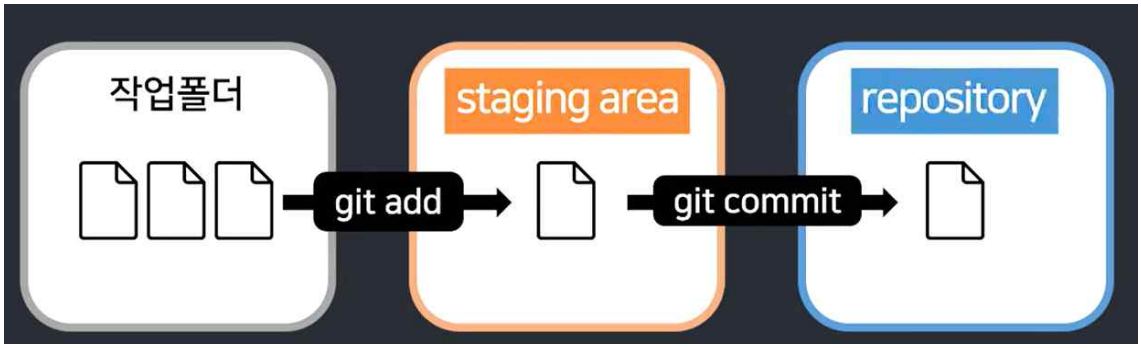
<깃허브, 깃 개념>

1. 깃이란?
- 버전 관리 도구(변경된 내용을 관리하는 도구)
- 언제 누가 뭘 변경했는지를 commit을 통해 비교할 수 있다.



git add 과정을 "Staging"이라고 부르기도 함
(결국 Staging한다 = commit 할 파일을 골라두었다는 의미)

- 깃허브는 코드 저장 공간
즉 코드 저장 공간에, 깃으로 변경된 사항까지 한 번에 저장할 수 있다.
우리 스터디에서는 각자의 폴더에 각자 푼 문제를 올리고, 타인의 코드에 comment를 남기는 식으로 스터디가 진행될 예정

2. 우리가 사용할 명령어 모음

(우선 깃을 설치해야한다)

1) 작업할 폴더를 만든다(git bash에서 mkdir <폴더명>해도 되고, 파일 탐색기 창에서 폴더 만들어도 되고)
2) 작업할 폴더로 이동한다(또는 파일 탐색기에서 폴더 이동하고, 마우스 우클릭 -> open git bash here해도 됨)
3) 맨 처음 쓰는 경우

git init

- 작업할 폴더에서 git을 사용하고 싶다면 이 명령어부터 입력해야 한다(즉, 맨 처음 한번만 하면 된다)
- 이제부터 깃 프로그램이 이 해당 폴더를 감시하기 시작함

cf) 깃허브 리포지토리의 내용을 로컬 컴퓨터로 복제하는 경우
: 복제하고 싶은 폴더 위치로 이동한 뒤, 클론하려는 리포지토리의 주소창을 복붙하여 입력한다.

git clone https://github.com/username/repo.git

- 이 경우에도 git init은 생략한다.

4) 히스토리를 생성한다(commit을 만든다)

git add <파일명>

(선택) ***git status***

git commit -m "<commit 메시지 내용>"

- git add 통해서 앞에서 배웠듯 staging을 하고,
- git status 통해 현재 staging된 파일 목록을 확인하고,
- 이 staging된 파일 목록을 커밋으로 올려 히스토리를 만든다(= 깃허브 저장소에 올릴 준비를 한다).
- 이때 add할 파일이 여러 개일 경우

git add <파일1>
git add <파일2>
git add <파일3> <파일4>

이런 식으로도, 한번에 여러 파일을 staging 해두는 것이 가능하다.

commit은 언제 하는가?

- 우리 스터디에서는 문제 풀이가 핵심이므로
    1) 문제를 풀 때마다 문제별로 하나씩 커밋을 만들어, 나중에 각 문제의 히스토리를 쉽게 확인할 수 있다.
    2) 버그를 수정하였을 때에도, 수정한 내용과 이유를 간단히 커밋으로 남겨 올릴 수도 있다.

하는 상황에 commit을 하면 된다.
(와중 저는 헷갈려서 여러 번 commit하고 있습니다... 그냥 열심히 해봅시다!)

git add와 git commit이 따로 있는 이유

- 만약 어떤 웹페이지나 앱을 만든다고 하면, 특정 파일들은 버전이 거의 없기도 하다(ex. 이미지 파일은 변동이 거의 없다) -> 따라서 특정 파일의 버전에만 변화를 줄 수 있는 기능을 위해, add와 commit이 분리되어 있다.
- 동시에 실행할 수 있는 명령어 또한 존재한다

1) 이미 추적되고 있는 변경된 파일을 자동으로 add하여 commit수행(새로운 파일을 추가하진 않음)

git commit -am "<Commit message>"

2) 새로운 파일까지 포함하여 모든 파일을 추가하고 커밋하는 방법

git add . && git commit -m "<Commit message">

5) 깃허브 리포지토리와 내 로컬 리포지토리를 연결한다.

git remote add origin https://github.com/username/repo.git

(선택) ***git remote -v***

- git remote add : 새로운 원격 저장소를 추가하는 명령어

- origin : 원격 저장소의 이름(별칭), 수정 가능
- 이 명령어를 통해 로컬 저장소에 'origin'이라는 이름으로 깃허브 리포지토리가 추가된다
- git remote -v 명령어 통해 현재 설정된 원격 리포지토리의 목록을 확인할 수 있다.

6) 이제, 만들어진 히스토리를 깃허브 리포지토리로 보낸다.

git push origin master

git push origin main

- master는 Git 저장소의 branch 이름
- 현재 우릿 스터디에서는 branch이름이 main이므로, git push origin main을 입력해야 함

cf) branch란?

- Git에서 독립적인 작업 흐름을 나타내며, 소프트웨어의 동시 작업에서 중추와 같은 역할을 한다.
- 여러 브랜치를 사용하여 서로 다른 기능이나 버그 수정을 동시에 작업할 수 있다.
- master branch의 조건

    1) 언제나 실행 가능하고

    2) 언제나 배포가 가능해야 한다.

    -> 따라서 master branch에서 직접 작업하는 것을 지양하고, 새로운 기능을 만들거나 실험하는 경우에는 새로운 branch를 만들어서 작업을 진행한다.

    - 장점 : 새 브랜치를 그대로 폐기하면 되므로 폐기가 쉽고, 특정 기능에 대해 모아서 확인이 쉽다.
        - 이러한 브랜치를 feature branch, topic branch라고 한다.

(PULL REQUEST 부분의 사진에서 branch에 대해 좀 더 자세히 알 수 있다.)

- branch 만드는 법

git checkout -b <branch 이름>

- 특정 branch에 커밋 올리는 법

git push -u origin <branch 이름>

7) 깃허브 리포지토리에서 로컬 리포지토리로 소스를 가져오는 명령어 : git pull, git fetch, git clone

git pull <원격 저장소 명 - 보통 origin> <branch 명 - 보통 master>

- 깃허브 리포지토리에서 최신 변경 이력을 다운로드하여 내 로컬 리포지토리로 그 내용을 적용시키는 것. Merge까지 동시에 수행한다.

git fetch <원격 저장소 명 - 보통 origin> <branch 명 - 보통 master>

- fetch 작업 후 변경 내용이 충돌하지 않았는지 직접 확인할 수 있도록, pull과 다르게 Merge를 바로 하지 않는다.

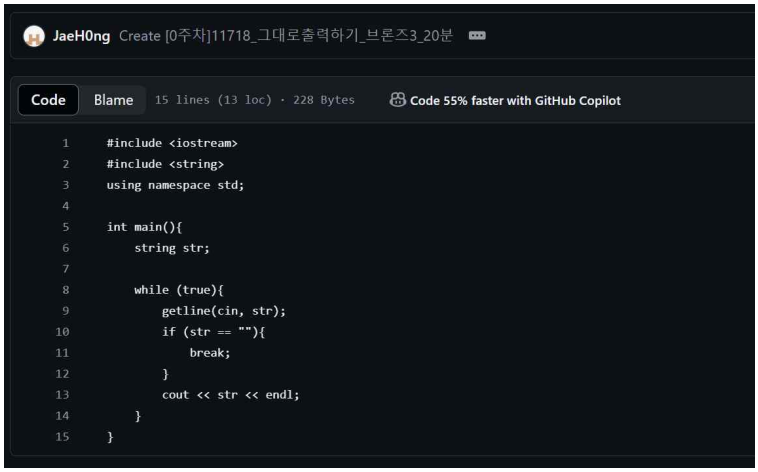
cf. 만약 git add ~ git push origin master까지 했는데 그 사이에 다른 협업자가 코드를 수정해서 올렸다고 가정하면

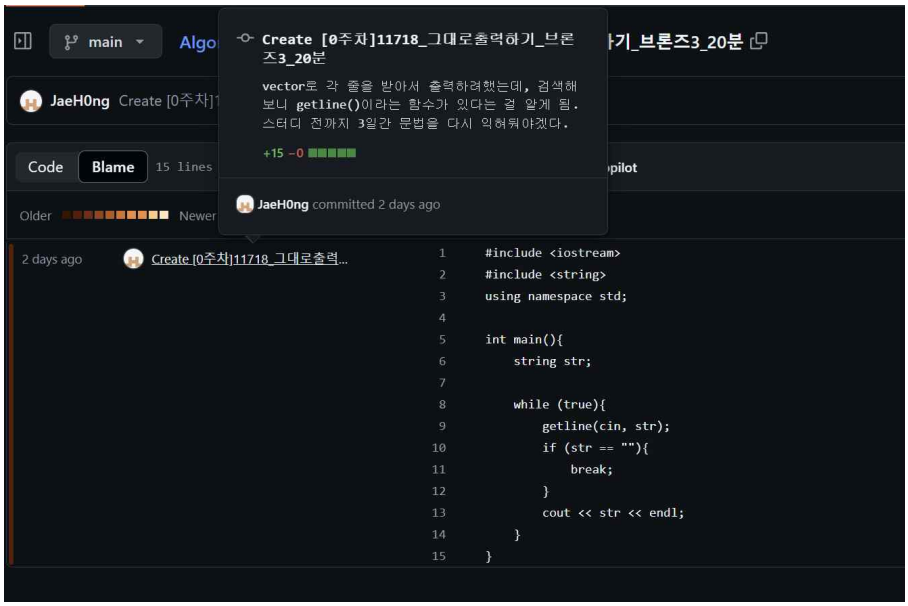
```
hint: 'git pull ...') before pushing again
```

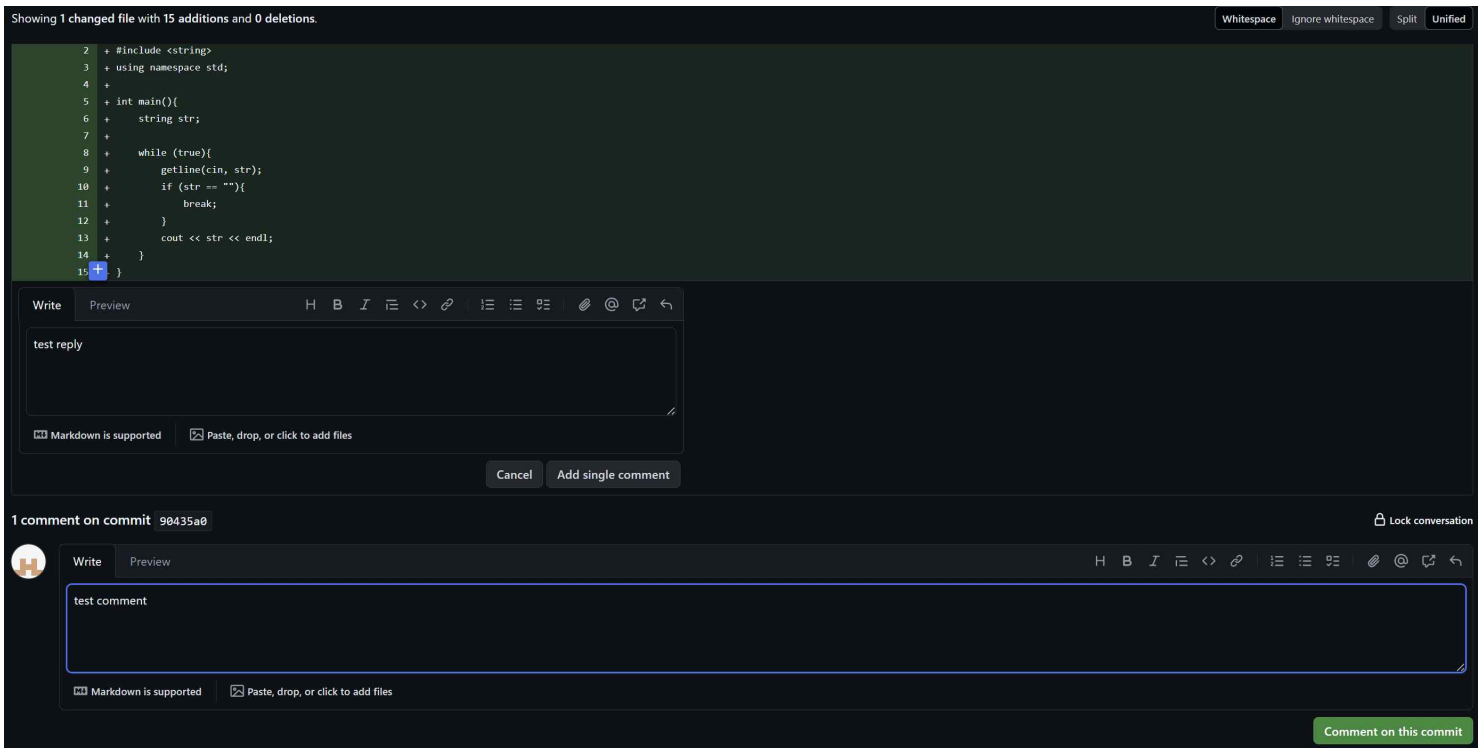
과 같은 메시지가 포함된 에러 메시지가 뜰 것이다. 깃허브 리포지토리의 수정사항이 나의 로컬 리포지토리에 반영되지 않았기 때문에 뜨는 메시지로, ***git pull origin master***를 먼저 한 뒤에 push해야한다.

(저희가 작성하는 코드에서는 아무래도 협업보다는 각자 푼 문제를 기록하는 용도이기 때문에, fetch가 본격적으로 사용되지는 않을 것 같습니다. 저도 협업을 경험하지 못해서 이 부분은 아직 잘 모르겠습니다. 나중에 혹시 경험하게 된다면 그때 공유하겠습니다)

cf) 다른 스터디원의 코드에 comment 다는 방법



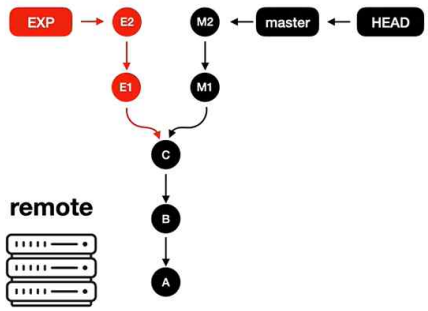


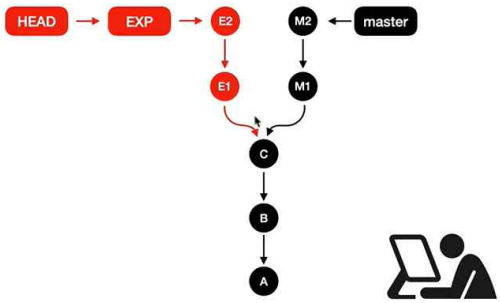


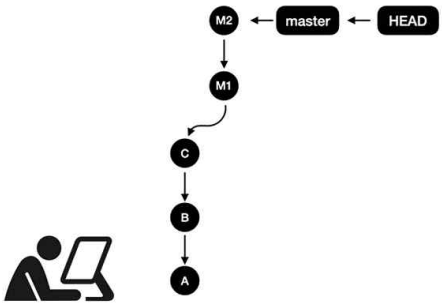
- 위 사진과 같이, 다른 스터디원의 소스코드에 들어간 뒤 Blame - 소스코드 선택 후
- 코드별로 reply를 달거나 전체 comment를 달아가면서 서로의 코드에 의견을 남겨서 스터디 방식을 좀 더 구체화할 수 있다.

(온라인으로 스터디할 경우에는 이 comment 기능을 활용해서 코드 리뷰할 예정)

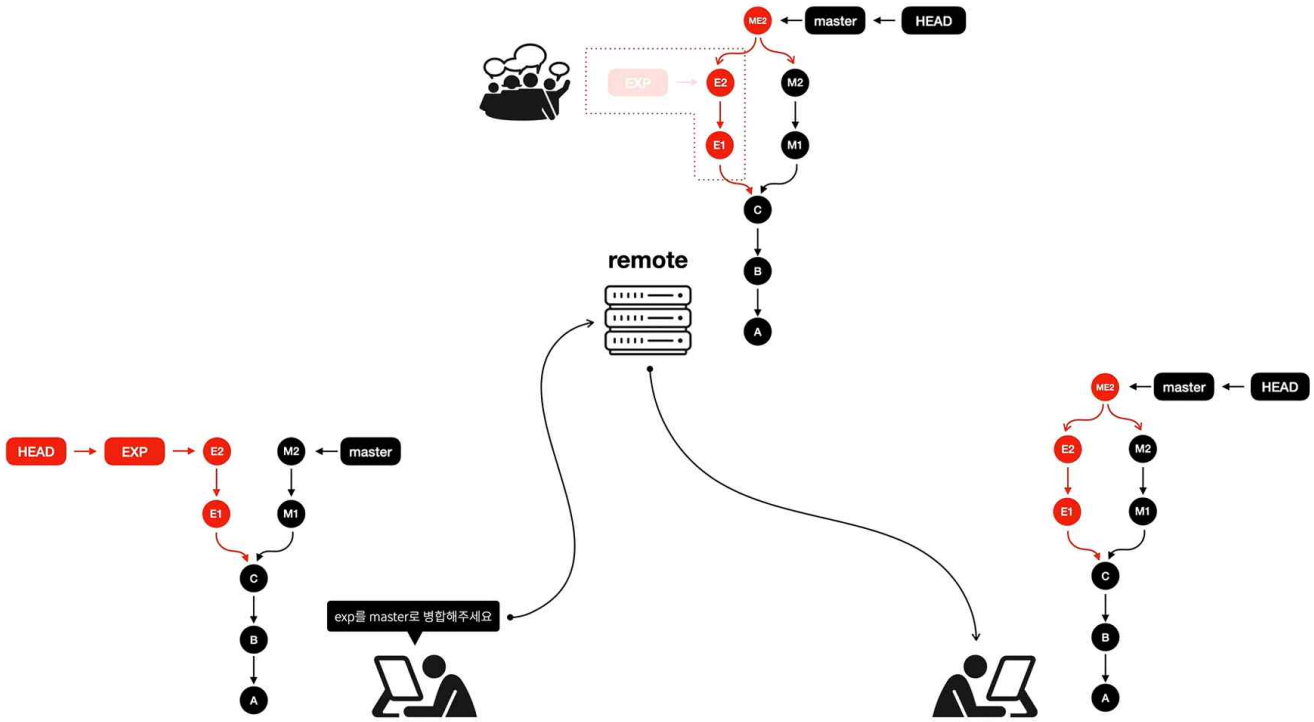
PULL REQUEST







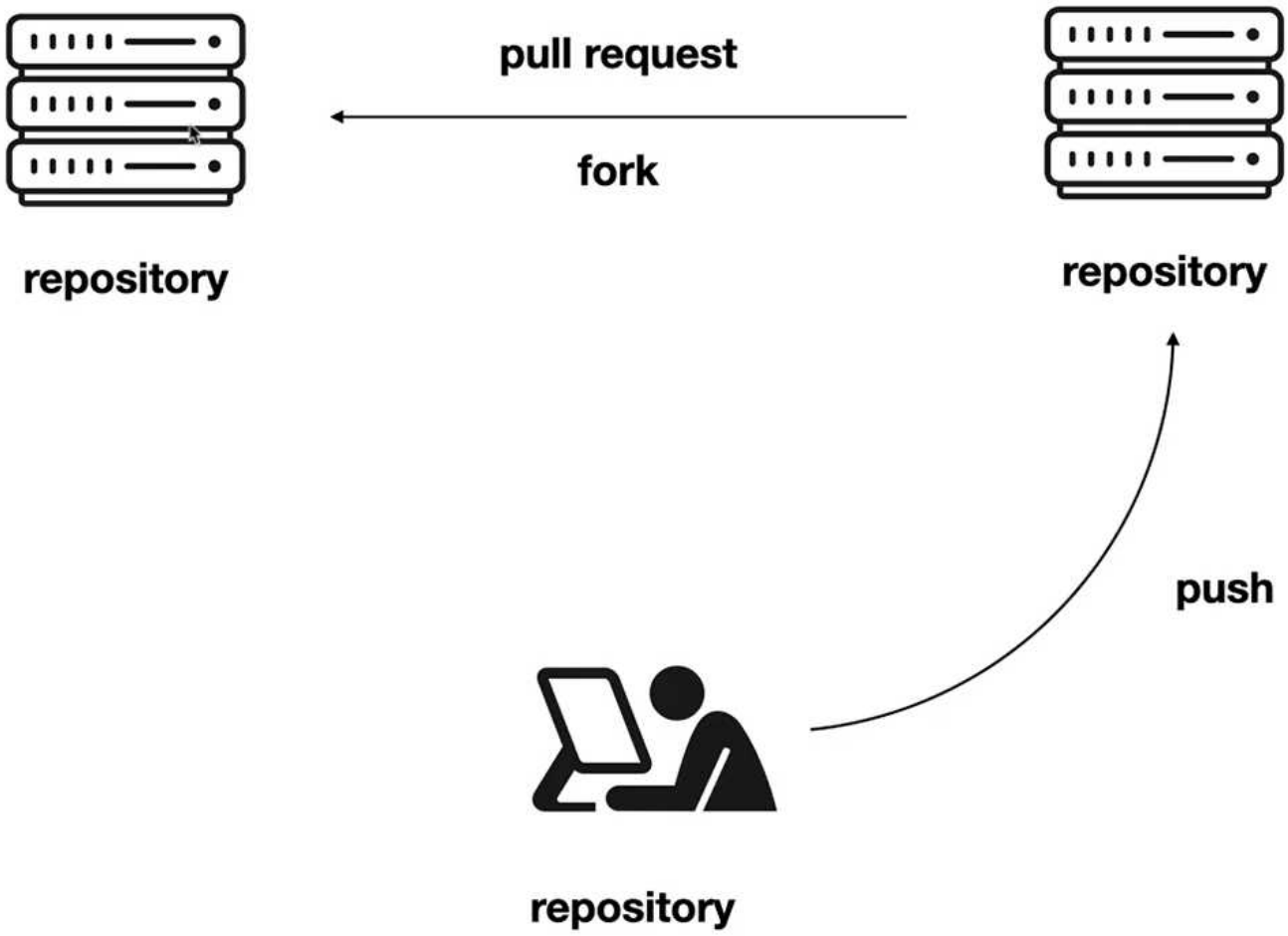
(왼쪽이 나, 중간이 깃허브, 오른쪽이 동료라고 생각하면 된다)



(내가 pull request를 보내고 다른 동료들과의 토론을 통해 나의 request를 사용하기로 하면, master branch는 내가 보낸 request를 가리킨다. 이후 다른 동료들은 새롭게 설정된 master branch를 통해 작업을 이어나갈 수 있다.)

pull request 2가지 방법

1. 나의 리포지토리
2. 오픈 소스 방식(우리 스터디)



- fork : 타인의 원격저장소를 나의 원격 저장소로 복제한다
이후 나의 원격저장소에 코드를 push하여 코드를 저장해간다.
- 나의 원격저장소에 코드 내용이 다 정리되면, original repository로 '내 코드를 가져가달라(pull)'고 request하는 것
- 이유 : original 리포지토리에 대한 수정 권한이 없기 때문

cf) 협업에서 comment 활용하는 방법
https://youtube.com/playlist?list=PLuHgQVnccGMBXv1OKe3Hn3Jq6F735-uWm&si=peiu1NSiTA9WWTcc
출처 : 생활코딩 유튜브

<시간복잡도>

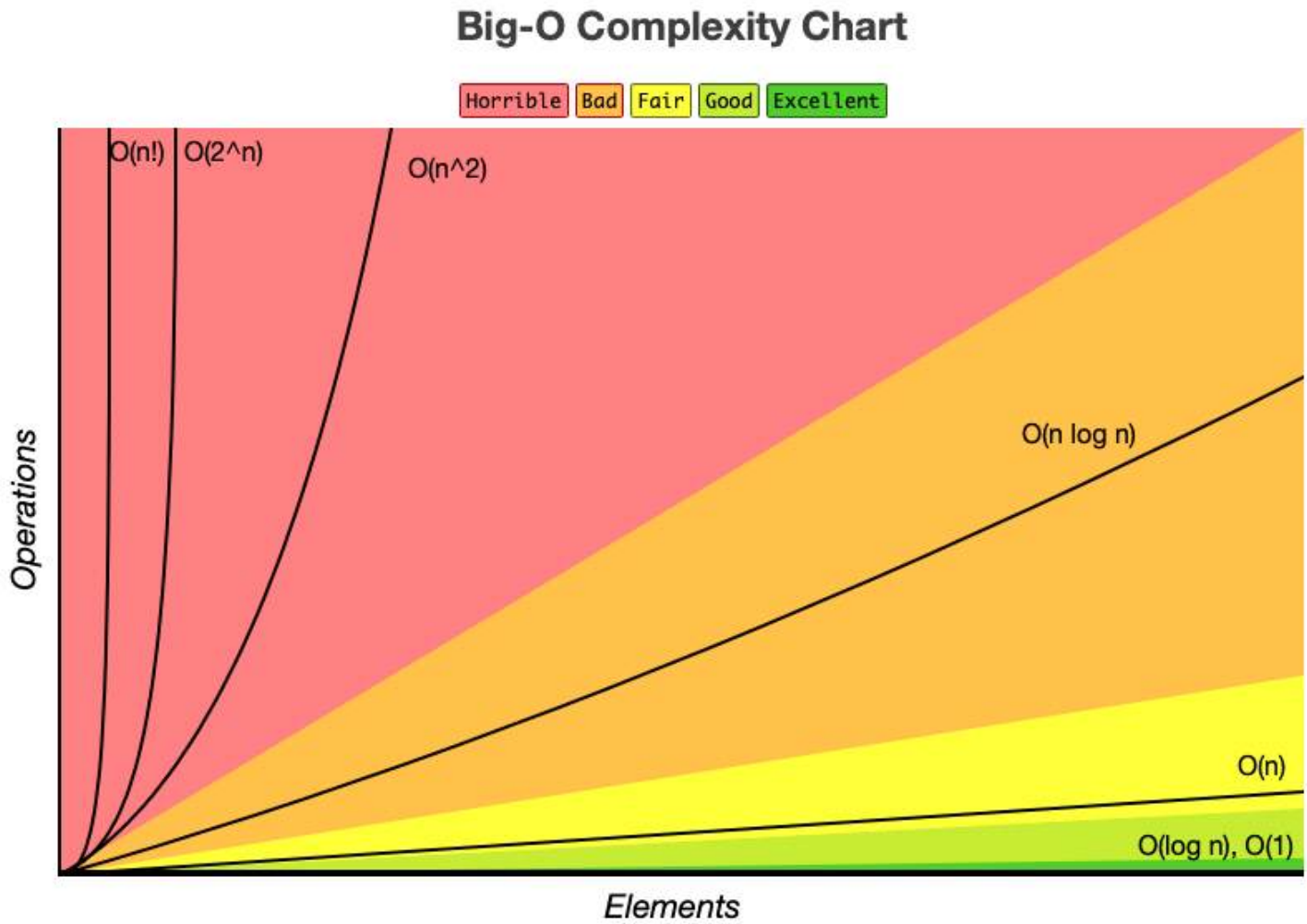
시간복잡도 간단 리뷰

- 알고리즘 : 문제 해결 방법을 정의한 일련의 단계적 절차이자, 어떤 문제를 풀기 위한 동작들의 모임
- 특정 문제를 풀어내는 그 '효율성'에 대해 고민하기 시작.
- 효율적인 알고리즘이란? 알고리즘이 수행을 시작하여 결과가 도출될 때까지
    - 시간이 짧고(시간복잡도)
    - 연산하는 컴퓨터 내의 메모리와 같은 자원을 덜 사용하는 것(공간복잡도)
    - 이 둘은 흔히 반비례 같은 상관관계이다.
- 대용량 시스템이 보편화되면서 과거에 비해 공간복잡도는 덜 중요해졌고, 시간복잡도가 더 중요해졌다.

Big - O

- 흔히 시간복잡도는 빅 오(big-O) 형태로 나타내며, 절대적인 시간을 의미하는 것이 아니다. 보통 최악의 경우(worst case) 또는 평균의 경우로 고려한다.

cf) 빅 오를 사용하는 이유는 단순히 worst case이기 때문은 아니다. best, worst, average의 모든 case에 대해 omega, theta, O 표기법은 각각 다 존재한다. 빅 오 표기법은 각각의 케이스의 상한선을 의미할 뿐이다.



Big-O(빅-오) 표기법의 종류

O(1)	상수시간 입력의 크기와 상관 없이 항상 같은 시간이 걸리는 알고리즘	배열, 해시 테이블
O(log n)	로그시간 문제를 해결하는데 필요한 단계들이 연산마다 줄어드는 알고리즘	이진탐색, 힙 삽입/삭제
O(n)	선형시간 입력값이 증가됨에 따라 시간 또한 같은 비율로 증가하는 알고리즘 모든 입력값을 적어도 한 번 이상은 살펴봐야 한다.	정렬되지 않은 리스트에서 최대값 최소값 탐색
O(n log n)	선형로그시간 데이터의 수가 2배로 늘 때, 연산 횟수는 2배 조금 넘게 증가한다.	힙 정렬, 병합 정렬
O(n^2)	제곱시간 2중 반복을 도는 알고리즘	버블 정렬
O(2^n)	지수시간 입력값의 제곱만큼 증가하는 알고리즘	피보나치(재귀), 부분집합의 모든 경우의 수를 도출
O(n!)	팩토리얼 시간 빅오 중 가장 느린 알고리즘	순열의 모든 경우의 수 도출

▼ **표 5-1** 자료 구조의 평균 시간 복잡도

자료 구조	접근	탐색	삽입	삭제
배열(array)	O(1)	O(n)	O(n)	O(n)
스택(stack)	O(n)	O(n)	O(1)	O(1)
큐(queue)	O(n)	O(n)	O(1)	O(1)
이중 연결 리스트(doubly linked list)	O(n)	O(n)	O(1)	O(1)
해시 테이블(hash table)	O(1)	O(1)	O(1)	O(1)
이진 탐색 트리(BST)	O(logn)	O(logn)	O(logn)	O(logn)
AVL 트리	O(logn)	O(logn)	O(logn)	O(logn)
레드 블랙 트리	O(logn)	O(logn)	O(logn)	O(logn)

▼ 표 5-2 자료 구조 최악의 시간 복잡도

자료 구조	접근	탐색	삽입	삭제
배열(array)	O(1)	O(n)	O(n)	O(n)
스택(stack)	O(n)	O(n)	O(1)	O(1)
큐(queue)	O(n)	O(n)	O(1)	O(1)
이중 연결 리스트(doubly linked list)	O(n)	O(n)	O(1)	O(1)
해시 테이블(hash table)	O(n)	O(n)	O(n)	O(n)
이진 탐색 트리(BST)	O(n)	O(n)	O(n)	O(n)
AVL 트리	O(logn)	O(logn)	O(logn)	O(logn)
레드 블랙 트리	O(logn)	O(logn)	O(logn)	O(logn)

우리에게 알고리즘 스터디에서 시간복잡도가 중요한 이유

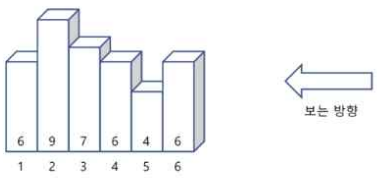
17608번 제출 맞힌 사람 숏코딩 재채점 결과 채점 현황 내 제출 질문 게시판

막대기

시간 제한	메모리 제한	제출	정답	맞힌 사람	정답 비율
1 초 (추가 시간 없음)	512 MB	27586	11674	9404	43.205%

문제

아래 그림처럼 높이만 다르고 (같은 높이의 막대기가 있을 수 있음) 모양이 같은 막대기를 일렬로 세운 후, 왼쪽부터 차례로 번호를 붙인다. 각 막대기의 높이는 그림에서 보인 것처럼 순서대로 6, 9, 7, 6, 4, 6 이다. 일렬로 세워진 막대기를 오른쪽에서 보면 보이는 막대기가 있고 보이지 않는 막대기가 있다. 즉, 지금 보이는 막대기보다 뒤에 있고 높이가 높은 것이 보이게 된다. 예를 들어, 그림과 같은 경우엔 3개(6번, 3번, 2번)의 막대기가 보인다.



N개의 막대기에 대한 높이 정보가 주어질 때, 오른쪽에서 보아서 몇 개가 보이는지를 알아내는 프로그램을 작성하려고 한다.

입력

첫 번째 줄에는 막대기의 개수를 나타내는 정수 N (2 ≤ N ≤ 100,000)이 주어지고 이어지는 N줄 각각에는 막대기의 높이를 나타내는 정수 h(1 ≤ h ≤ 100,000)가 주어진다.

출력

오른쪽에서 N개의 막대기를 보았을 때, 보이는 막대기의 개수를 출력한다.

- 알고리즘 문제에서는 입력의 크기(위 예시에서는 정수 N)와 시간 제한(위 예시에선 1초)이 주어진다
- 코딩 테스트에서 문제의 제한 시간은 보통 1~5초
- **일반적인 CPU 기반의 PC나 채점용 컴퓨터에서 1초에 실행할 수 있는 최대 연산 횟수는 약 1억번**이다.
- What if...
    - 문제에서 주어진 N 최댓값이 10만이고 주어진 제한 시간이 1초라면?

=> 시간복잡도가 $O(n^2)$인 알고리즘의 연산횟수는 100,000 * 100,000 = 100억번 이므로, 이 알고리즘은 사용할 수 없다.

- ***각 주차 발표자는 이를 고려하여, 문제 풀이가 시작되면 해당 문제는 시간복잡도가 얼마나 되는 알고리즘을 활용할 수 있는지를 언급해주자.***

(참고)

1초에 최대 연산 횟수(최대 입력 크기)

시간 복잡도	최대 연산 횟수
O(N)	약 1억번
O(N^2)	약 1만번
O(N^3)	약 500번
O(2^N)	약 20번
O(N!)	10번

제한 시간이 1초 일 경우, N의 범위에 따른 시간 복잡도 선택

- **N의 범위가 500:** 시간 복잡도가 `O(N^3)` 이하인 알고리즘을 설계
- **N의 범위가 2,000:** 시간 복잡도가 `O(N^2)` 이하인 알고리즘을 설계
- **N의 범위가 100,000:** 시간 복잡도가 `O(NlogN)` 이하인 알고리즘을 설계
- **N의 범위가 10,000,000:** 시간 복잡도가 `O(N)` 이하인 알고리즘을 설계
- **N의 범위가 10,000,000,000:** 시간 복잡도가 `O(logN)` 이하인 알고리즘을 설계

시간복잡도 참고자료

https://velog.io/@jiyaho/%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98-%EC%8B%9C%EA%B0%84-%EC%A0%9C%ED%95%9C%EA%B3%BC-%EC%8B%9C%EA%B0%84-%EB%B3%B5%EC%9E%A1%EB%8F%84

https://velog.io/@joychae714/%EC%9E%90%EB%A3%8C%EA%B5%AC%EC%A1%B0

https://hi-claire.tistory.com/m/25

https://yummy0102.tistory.com/490