[자료구조와 알고리즘] 코랩세션 Wrap Up 리포트

작성팀: 6조 이영기

Wrap Up 리포트 작성 내용

1) 논의 주제
    a) 보라색으로 작성된 주제는 예시 주제로, 팀 내 논의하고 싶은 주제가 있다면 해당 주제로 논의해주세요!
2) 팀원별 핵심 아이디어
3) 논의 과정
4) 최종 논의 결과 및 회고

공통 예시 주제 (Day 01 - Day 05)

Step 01. Pseudo code: 내 풀이를 10~20줄 내외로 요약(입력 출력, 핵심 로직, 예외 포함)

Step 02. 시간 공간 복잡도: 최선 평균 최악 중 무엇을 기준으로 말하는지 명시

Step 03. 효율성 주장: “왜 이 방식이 합리적인가?”를 대안 1~2개와 비교로 설명 (반례 엣지케이스 포함)

Day 01 코랩 세션

1. 논의 주제

- 핵심 과제: 정수 배열 nums에서 합이 target이 되는 두 숫자의 인덱스를 찾는 Two Sum 문제 해결
- 주요 쟁점: 배열 인덱스 기반 접근 방식 및 메모리 구조의 이해
    - 브루트 포스(Brute Force) 방식과 해시맵(Hash Map) 방식의 메커니즘 차이 비교
    - 데이터 크기 증가에 따른 시간복잡도 O(N^2) vs O(N)의 실제적 영향력 파악

---

2. 팀원별 핵심 아이디어

- 브루트 포스 전략 (아이디어 제안):
    - 모든 인덱스 쌍 (i, j)를 이중 반복문으로 직접 비교하여 합이 target인 경우를 찾음

    - "자물쇠 번호를 풀기 위해 0000부터 9999까지 전부 대입하는 것"과 같은 직관적인 방식 제안
- 해시맵 전략 (아이디어 제안):
    - 리스트를 단 한 번 순회하며, target - 현재값이 메모장(Dictionary)에 있는지 확인
    - "입구에서 방명록을 확인하고 바로 목적지로 가는 것"과 같은 효율적인 탐색 방식 제안

---

3. 논의 과정

- Step 1. 브루트 포스 구현 및 한계점 인식:
    - 이중 for문을 통해 인덱스 i와 j를 순회하며 정답을 찾는 코드를 작성함
    - 전체 비교 방식은 입력값 N 에 대해 N times N 번의 연산이 필요하여 데이터가 10배 늘면 연산은 100배 늘어난다는 점을 분석함
- Step 2. 해시맵을 통한 성능 개선 시도:
    - enumerate를 사용하여 인덱스와 값을 동시에 관리하고, 한 번 본 숫자를 seen 딕셔너리에 저장하는 로직을 구축함
    - "타겟이 명확할 때 사용할 수 있는 최적의 솔루션"이라는 인사이트를 도출함
- Step 3. 알고리즘 효율성 비교 실험:
    - 데이터 개수가 100만 개일 때, 브루트 포스는 약 2.7시간이 걸리지만 해시맵은 0.01초 만에 끝난다는 수치적 차이를 시뮬레이션함

---

4. 최종 논의 결과 및 회고

- 최종 결과:
    - 시간복잡도: O(N^2) 에서 O(N) 으로 획기적으로 개선됨을 확인
    - 트레이드 오프(Trade-off): 해시맵 방식은 시간 효율성이 뛰어나지만, 추가적인 저장 공간(메모리)이 필요하다는 점을 명확히 함
- 회고 및 향후 과제:
    - 구현 난이도: 브루트 포스가 구현은 쉽지만, 대규모 데이터 처리 시 발생하는 비용(시간) 이 곧 프로젝트의 성패와 직결됨을 깨달음
    - 학습 성과: "이미 본 데이터를 기억하느냐, 매번 새로 찾아 헤매느냐"의 차이가 알고리즘 설계의 핵심임을 이해함
    - 향후 다짐: 단순히 루프를 돌리는 방식에서 벗어나, 추가 메모리를 활용해 시간을 버는 해시맵과 같은 효율적인 사고를 체득하기로 함

Day 02 코랩 세션

1. 논의 주제

- 핵심 과제: 스택(Stack) 자료구조를 활용하여 주어진 문자열 s가 올바른 괄호 구성인지 판별하는 알고리즘 구현
- 주요 학습 포인트:
  - 스택의 LIFO(Last-In, First-Out) 구조 및 push pop 연산 원리 이해
  - 문자열 내 괄호 패턴을 스택으로 검증하는 로직 설계
  - 시간복잡도 O(N) 및 공간복잡도 O(N) 알고리즘의 분석과 효율성 검토

---

2. 팀원별 핵심 아이디어

- 스택 기반 매칭 아이디어 (학생 제안):
  - 문자열을 왼쪽에서 오른쪽으로 순회하며 열린 괄호((, {, [)는 스택에 쌓고(push), 닫힌 괄호가 나오면 스택의 가장 위(top) 원소를 꺼내(pop) 짝이 맞는지 대조함
  - 스택이 비어있는 상태에서 닫힌 괄호가 나오거나, 꺼낸 원소와 짝이 맞지 않으면 즉시 False를 반환함
- 데이터 구조 최적화:
  - 괄호의 짝을 매칭할 때 if-else 문 대신 딕셔너리(dict) 를 사용하여 코드 가독성과 확장성을 높임
- 예외 케이스 및 확장성 고려:
  - 단순히 괄호만 있는 문자열이 아니라, (3+4) (2+5)와 같이 숫자와 연산자가 포함된 수식에서도 괄호의 유효성을 검증할 수 있도록 설계

---

3. 논의 과정

- Step 1. 문제 분석 및 엣지 케이스 도출:
  - 입력이 비어있거나(""), 열린 괄호만 있거나((((((), 순서가 뒤섞인 경우(([)]) 등 다양한 실패 사례를 정의함
- Step 2. 의사코드(Pseudocode) 설계:
  - 구현 전 "순회 → 열린 괄호 push → 닫힌 괄호 pop & 비교 → 최종 스택 확인"으로 이어지는 단계별 로직을 자연어로 먼저 정립함
- Step 3. 코드 구현 및 디버깅:
  - 처음 구현 시 발생할 수 있는 '빈 스택에서의 pop 오류'를 방지하기 위해 if not stack: return False와 같은 예외 처리를 선행하도록 로직을 보완함
  - 숫자나 연산자(+,) 등 괄호 이외의 문자는 무시하고 넘어가는 else: continue 로직을 추가하여 수식 지원 기능을 강화함

---

4. 최종 논의 결과 및 회고

- 최종 논의 결과:
  - 시간복잡도: 문자열을 한 번만 순회하므로 O(N) 달성
  - 공간복잡도: 최악의 경우 모든 문자가 열린 괄호일 때 N 만큼 쌓이므로 O(N) 소요
  - 도구의 적합성: 괄호는 나중에 열린 것이 먼저 닫혀야 하므로, 순서를 기억하지 못하는 해시테이블이나 선입선출인 큐보다 스택이 가장 적합한 도구임을 확인함

- 회고:
  - 단순한 문제 같지만 pop 연산 전 스택 상태를 확인하는 디테일이 프로그램의 안정성을 결정함을 체득함
  - 이전 문제(Two Sum)에서 배운 해시맵과 이번 스택의 차이점을 비교하며, 문제의 특성(순서가 중요한지, 값이 중요한지)에 따라 올바른 자료구조를 선택하는 안목을 기르는 계기가 됨

왜 해시맵만으로는 어려울까요? (논의 과정)

제출하신 Day 2 미션 파일의 회고 내용과 일맥상통하는 부분입니다.

- 순서 기억의 부재: 해시맵은 "값이 어디에 있는가"는 잘 찾지만, "어떤 순서로 들어왔는가"를 자체적으로 관리하지 않습니다.
- 복잡도 증가: 위 코드처럼 인덱스를 일일이 관리하면, 짝을 찾을 때마다 리스트를 뒤져야 하므로 시간복잡도가 O(N) 보다 나빠질 가능성이 큽니다.
- 스택의 본질: 결국 "가장 최근에 열린 괄호"를 찾는 과정에서 `indices[target_open].pop()`과 같은 연산을 쓰게 되는데, 이는 해시맵을 스택의 저장소로만 쓰는 셈이 됩니다.

최종 결론 및 회고 정리

- 팀원별 핵심 아이디어: 해시맵은 괄호의 짝 정보를 저장하는 용도로는 매우 훌륭하지만(코드 가독성 향상), 전체 로직을 담당하기엔 순서 제어 능력이 부족하다는 점을 확인했습니다.
- 논의 과정: "이미 본 데이터를 기억하느냐"라는 측면에서 해시맵과 스택은 닮았지만, 괄호 문제는 '가장 최근의 기억' 이 중요하므로 LIFO(후입선출) 방식인 스택이 가장 경제적이라는 결론에 도달했습니다.
- 최종 결과: 괄호 검증 알고리즘에서 해시맵은 '매핑 테이블' 로서 스택을 보조할 때 가장 빛이 납니다.

Day 03 코랩 세션

추가 예시 주제 : DFS와 BFS의 차이를 정리하고, 이 미션 조건에서 어떤 탐색이 더 효율적인지

근거(복잡도 메모리 목표: 최단거리 등)로 선택해봅시다.

Step 01. Pseudo code (BFS)

- 입력: M x N 크기의 2차원 문자열 배열 grid ('1': 육지, '0': 바다)
- 출력: 섬의 총 개수 count (정수)

Python

```
# 1. 초기화: 섬 개수(count) = 0, 격자 크기(rows, cols) 확인
# 2. 격자 순회: 모든 칸 (r, c)를 하나씩 검사
# 3. 육지 발견: if grid[r][c] == '1' (방문하지 않은 육지라면)
  # 3-1. count를 1 증가
  # 3-2. 큐(Queue) 생성 및 시작 좌표 (r, c) 삽입
  # 3-3. grid[r][c] = '0' (방문 표시: 다시 안 오도록 바다로 변경)
  # 3-4. BFS 탐색 시작: while 큐가 비어있지 않은 동안
    # a. 큐에서 현재 좌표 (curr_r, curr_c) 추출
    # b. 상, 하, 좌, 우 4방향에 대해 다음 좌표 (nr, nc) 계산
    # c. 유효성 검사: (nr, nc)가 격자 내에 있고 grid[nr][nc] == '1'이면
      # - 큐에 (nr, nc) 삽입
      # - grid[nr][nc] = '0' (방문 표시)
# 4. 결과 반환: 최종 count 값
```

Step 02. 시간 공간 복잡도 (최악의 경우 기준)

- 시간 복잡도: O(m*n)
  - 이유: 모든 칸을 최대 한 번씩 방문합니다. 육지인 칸은 큐에 들어갔다 나오며 4방향을 검사하고, 바다인 칸은 if문 조건에서 바로 걸러지므로 전체 칸 수에 비례하는 연산만 수행합니다.
- 공간 복잡도: O(m,n)
  - 이유: 큐에 저장되는 최대 좌표의 수는 격자의 너비나 높이 중 작은 값에 비례하여 쌓입니다

Step 03. 효율성 주장

이 방식이 합리적인 이유는 다른 대안들과 비교했을 때 안정성과 명확성이 높기 때문입니다.

비교군	특징 및 한계점	효율성 판단
대안 1: 재귀 DFS	코드는 짧으나 섬이 매우 클 경우(예: 수만 칸) Recursion Error 발생 위험이 큼.	BFS 승: 대규모 데이터 처리 시 안정적임.

대안 2: 단순 루프	큐 스택 없이 매번 전체 격자를 다시 훑으며 연결성을 체크하면 중복 연산이 기하급수적으로 늘어남.	BFS 승: 한 번 방문한 노드를 즉시 지워 중복을 원천 차단함.

- 반례 엣지케이스 대응:
  - 격자가 비어있는 경우: if not grid 조건으로 즉시 0 반환.
  - 전체가 육지인 경우:O(m*n) 내에 모든 칸을 0으로 만들며 정상 종료.
  - 대각선 육지: 4방향 탐색 로직(nr, nc 계산)을 통해 대각선은 별개의 섬으로 정확히 분리 처리.

Day 04 코랩 세션

상황별 최적의 도구 선택 (Quick Select, Heap, Binary Search)

1. 퀵셀렉트 (Quick Select)

- 핵심 원리: 퀵 정렬의 원리를 이용하되, 피벗(Pivot)을 기준으로 찾고자 하는 값이 포함된 구역만 재귀적으로 파고듭니다. 나머지 절반은 버립니다.
- 목표: 정렬되지 않은 배열에서 K번째로 크거나 작은 값 찾기.
- 시간 복잡도: 평균 O(N) , 최악 O(N^2) .
- 논의 포인트: "전체를 정렬(O(N log N))하지 않고도 특정 순위의 값을 찾을 수 있는 이유는 무엇인가?" (절반씩 버리는 분할 정복의 힘)

---

2. 힙 (Heap Priority Queue)

- 핵심 원리: 완전 이진 트리 구조를 유지하며, 부모 노드가 자식 노드보다 항상 크거나 작은 상태를 유지합니다.
- 목표: 최댓값 또는 최솟값을 빠르게 추출하거나, 실시간으로 우선순위를 관리하기.
- 시간 복잡도: 삽입 삭제 O(log N) , K 번째 값 찾기 O(N log K) .
- 논의 포인트: "데이터가 실시간으로 계속 추가되는 상황(Stream)이라면, 퀵셀렉트보다 힙이 유리한 이유는 무엇인가?" (유지보수 비용의 차이)

---

3. 이진 탐색 (Binary Search)

- 핵심 원리: 이미 정렬된 데이터에서 중앙값을 기준으로 탐색 범위를 매 단계마다 절반씩 줄여나갑니다.
- 목표: 특정 값(Target)의 **위치(인덱스)**를 찾거나 존재 여부 확인하기.
- 시간 복잡도: O(log N) .

- 논의 포인트: "배열이 회전되어 있거나(Rotated) 일부만 정렬된 특이한 상황에서도 이진 탐색의 논리(절반 버리기)를 적용할 수 있는가?"

---

4. [비교 요약] 알고리즘 선택 가이드

상황	추천 알고리즘	이유
정렬된 상태에서 특정 값 찾기	이진 탐색	가장 빠름 (O(log N))
정렬 안 된 데이터에서 딱 하나(K번째) 찾기	퀵셀렉트	평균 성능이 가장 우수 (O(N))
데이터가 계속 추가되면서 최댓값 유지	힙 (Heap)	동적 데이터 처리에 최적화
모든 데이터의 완벽한 순서가 필요할 때	병합 퀵 정렬	정렬 이후 이진 탐색 등 활용 가능

Day 05 코랩 세션

추가 예시 주제

- (데일리미션) 각 정렬 알고리즘은 어떤 데이터 제약(거의 정렬됨, 안정성, 메모리, 최악 보장 등)에서 가장 적합한지 사례로 토론해보세요.

- (심화미션) 최소힙 없이 같은 기능을 구현할 대안을 제시하고, 연산별 시간복잡도 관점에서 효율성이 어떻게 변하는지 비교해보세요.

주제: 비교하지 않는 정렬, 카운팅 정렬 (Counting Sort)

1. 카운팅 정렬의 정의

- 비교 기반 정렬O(N log N)의 한계를 깨는 알고리즘: 두 수를 비교해서 자리를 바꾸는 방식이 아니라, 각 숫자의 출현 빈도를 세어서 위치를 바로 지정하는 방식입니다.
- 시간 복잡도: O(N + K) (여기서N은 데이터 개수, K는 데이터의 최대값).

---

2. 핵심 3단계 (동작 원리)

1. Counting (세기): 각 숫자가 몇 번 나왔는지 count 배열에 기록합니다.
2. Cumulative Sum (누적합): 각 숫자가 결과 배열에서 어디서 끝날지 **'예약 번호'**를 매깁니다.
3. Placement (배치): 원본 배열을 뒤에서부터 읽으며, 예약 번호를 하나씩 깎으면서(-= 1) 결과 배열에 배치합니다.

Q1. 왜 뒤에서부터(역순으로) 채워넣나요?

- 답변: 데이터에 같은 숫자가 있을 때, 원래 순서를 유지하기 위해서입니다 (안정 정렬). 앞에서부터 채우면 원래 앞에 있던 숫자가 뒤로 갈 수 있습니다.

Q2. `max(list)`가 너무 크면 어떤 문제가 생기나요?

- 답변: 만약 숫자는 2개인데 최대값이 10억이라면, `count` 배열의 크기가 10억이 되어 메모리 낭비가 심각해집니다. 이럴 때는 카운팅 정렬 대신 다른 정렬을 써야 합니다.

Q3. 카운팅 정렬을 쓸 수 없는 데이터 타입은?

- 답변: 소수(실수) 데이터. 배열의 인덱스는 정수여야 하므로 `0.5`, `1.2` 같은 값은 카운팅 정렬을 바로 적용할 수 없습니다.

5. 코랩 세션 논의 질문 (Discussion Questions)

1. 메모리 효율성: "퀵셀렉트는 제자리(In-place)에서 동작하여 메모리가 적게 드는데, 합병 정렬이나 힙 생성과 비교했을 때 공간 복잡도 면에서 어떤 이점이 있을까?"
2. 안정성: "데이터가 거의 정렬되어 있는 최악의 경우, 퀵셀렉트의 성능 저하(O(N^2))를 어떻게 방지할 수 있을까?" (예: 랜덤 피벗 선택)
3. 실전 적용: "로그 분석 시스템에서 실시간으로 가장 에러가 많이 발생하는 상위 10개 항목을 뽑아야 한다면, 힙과 퀵셀렉트 중 무엇이 더 적합할까?"

Q1. 메모리 효율성: 퀵셀렉트 vs 합병 정렬 vs 힙

- 답변 핵심: **"추가 공간을 사용하는가(In-place 여부)"**가 포인트입니다.
- 상세 설명:
  - 퀵셀렉트: 원본 배열 안에서 요소들의 위치만 바꾸는(Swap) In-place 알고리즘입니다. 따라서 추가적인 메모리 공간이 거의 들지 않습니다(O(1)).
  - 합병 정렬: 데이터를 쪼개서 담아둘 복사본 리스트가 반드시 필요합니다. 그래서 데이터 양만큼의 추가 메모리(O(N))가 소모됩니다.
  - 힙: 기존 배열을 힙 구조로 재배치(heapify)하면 O(1) 공간으로 가능하지만, 보통 파이썬의 heapq 등을 사용해 새로운 리스트를 만들면 O(N) 공간이 필요합니다.
- 결론: 메모리가 극도로 제한된 환경(임베디드 등)에서는 퀵셀렉트가 가장 유리합니다.

Q2. 퀵셀렉트의 최악의 경우(O(N^2)) 방지 대책

- 답변 핵심: **"피벗(Pivot)을 얼마나 영리하게 고르는가"**가 해결책입니다.
- 상세 설명:
  - 문제 상황: 이미 정렬된 배열에서 맨 앞이나 맨 뒤를 피벗으로 고르면, 한 쪽은 0개, 다른 쪽은 N-1 개로 나뉘어 속도가 급격히 느려집니다.
  - 해결책 1 (Random Pivot): 피벗을 무작위로 고르면 최악의 상황을 만날 확률이 수학적으로 거의 0에 수렴합니다.
  - 해결책 2 (Median-of-Three): 맨 앞, 맨 뒤, 중간값 중 "중간값"을 피벗으로 선택하여 데이터를 최대한 균등하게 분할합니다.
- 결론: 실전에서는 랜덤 피벗만 사용해도 평균적으로 매우 안정적인 O(N) 성능을 얻을 수 있습니다.

Q3. 실전 적용: 실시간 상위 10개 항목 추출 (힙 vs 퀵셀렉트)

- 답변 핵심: **"데이터가 정적인가, 동적인가"**를 따져야 합니다.
- 상세 설명:
  - 퀵셀렉트: 데이터가 이미 다 모여 있는 배치(Batch) 상황에서 유리합니다. 한 번 실행해서 상위 10개를 툭 뽑아내고 끝낼 때 가장 빠릅니다.
  - 힙 (Heap): 데이터가 초 단위로 계속 들어오는 스트리밍(Stream) 상황에서 유리합니다.
    - 크기가 10인 '최소 힙(Min-Heap)'을 유지하면서, 새로운 데이터가 들어올 때마다 힙의 최솟값과 비교해 교체하면 됩니다.
    - 이 방식은 전체 데이터를 다 저장할 필요 없이 항상 딱 10개만 유지하면 되므로 메모리와 시간 면에서 훨씬 효율적입니다.
- 결론: 실시간 시스템이라면 **힙(Heap)**이 정답입니다.

"결국 절대적으로 우월한 알고리즘은 없습니다. 메모리가 중요한지(In-place), 데이터가 실시간으로 들어오는지(Stream), 혹은 **데이터의 범위가 한정적인지(Counting)**에 따라 우리가 가진 도구 상자에서 가장 적절한 도구를 꺼내 쓰는 능력이 중요합니다."