2주차: 슬라이딩 윈도우와 스택

로그의 연속 구간과 반복 처리

이벤트 로그 "abba c"에서 같은 문자가 두 번 나오지 않는 가장 긴 연속 구간을 찾습니다. 공백도 하나의 문자이며, "ba c"의 길이는 4입니다.

로그가 길어질 때 구간마다 문자를 다시 모으면 같은 기록을 반복해서 읽습니다. 구간의 양끝이 움직일 때 들어오는 문자와 나가는 문자만 반영하면 어떨까요?

이번 주의 주제: 구간 상태 갱신과 가장 최근에 저장한 정보부터 꺼내는 스택

2주차의 문제와 학습 목표

3차시: 고정 길이와 가변 길이 구간. 4차시: 깊이별 최솟값과 후위 표기 식의 계산 순서.

- 구간 경계와 저장 상태의 의미
- 포인터의 전체 이동 수와 시간 비용
- 스택의 복원 정보와 피연산자 순서
- 양수 조건, 중복 최솟값, 음수 나눗셈 검증

실습 문제: LeetCode 3, 209, 155, 150.

고정 길이 구간의 갱신

`nums=[1,2,4,2]`, 폭 `w=2`이면 연속 구간의 합은 `[3,6,6]`입니다. 처음에는 `sum(nums[:w])`를 계산합니다.

새 오른쪽 인덱스	나가는 값	들어오는 값	갱신된 합
초기 구간 0..1	없음	1, 2	3
2	nums[0] = 1	nums[2] = 4	3 - 1 + 4 = 6
3	nums[1] = 2	nums[3] = 2	6 - 2 + 2 = 6

새 오른쪽 인덱스 `r`: 나가는 위치는 `r-w`. `range(w, len(nums))`는 마지막 인덱스까지 방문합니다.

고정 길이 구간의 구현과 비용

```
def fixed_window_sums(nums, width):
    if not 1 <= width <= len(nums):
        raise ValueError("width must be within 1..len(nums)")
    total = sum(nums[:width])
    result = [total]
    for right in range(width, len(nums)):
        total += nums[right] - nums[right - width]
        result.append(total)
    return result
```

정수 연산을 상수 비용으로 보면 시간은 O(n). 공간 비용: 초기 슬라이스 O(w), 결과 O(n-w+1). `total += value`는 누적이고 `total =+ value`는 양의 부호를 붙인 값을 대입합니다.

이전 구간 합에서 나가는 값 하나를 빼고 들어오는 값 하나를 더합니다. `right-width`가 나가는 위치입니다. 매번 길이 width인 구간을 새로 슬라이스하여 합산하면 O(n·width)가 되지만 이 코드는 초기 구간만 합산합니다.

고정 길이와 가변 길이의 경계

고정 길이 구간은 오른쪽 끝을 한 칸 옮길 때 왼쪽 끝도 한 칸 옮깁니다. 가변 길이 구간은 문제 조건을 만족하도록 왼쪽 끝을 움직입니다.

문제	유지하는 정보	왼쪽 끝을 옮기는 조건
고정 폭의 합	현재 합	새 값이 들어올 때마다 한 칸
중복 없는 최장 구간	현재 문자 집합	새 문자가 이미 존재하는 동안
목표 합 이상의 최단 구간	현재 합	합이 목표 이상인 동안

포함되는 양끝: `left`, `right`. 길이: `right-left+1`. 축소 뒤에는 구간이 비어 있을 수도 있습니다.

LeetCode 3: 문제 정의와 단순 풀이

Longest Substring Without Repeating Characters는 문자가 중복되지 않는 연속 부분문자열의 최대 길이를 요구합니다. 문자를 건너뛰어 고르는 부분수열과 구분합니다.

입력: 길이 0~100000의 문자열; 영문자·숫자·기호·공백 포함. 빈 문자열의 답: 0

단순 풀이에서는 시작점마다 빈 집합을 만들고 중복이 나타날 때까지 오른쪽으로 읽습니다. 최악에는 많은 구간을 다시 방문하므로 $O\left(n^2\right)$ 번 문자를 검사합니다.

중복 없는 구간의 불변식

새 문자 추가 전 `active`: `text[left:right]`의 서로 다른 문자 집합. 새 문자 `char`가 이미 있으면 `text[left]`를 지우고 `left`를 한 칸 옮깁니다.

중복 문자가 없어질 때까지 반복한 뒤 `char`를 추가합니다. 이제 집합은 양끝을 포함한 `text[left:right+1]`의 문자와 일치하고, 구간 안에는 중복이 없습니다.

최장 길이는 새 문자를 넣은 다음 갱신합니다. 왼쪽 끝을 옮길 때 구간 밖의 문자나 새 오른쪽 문자를 잘못 지우지 않도록 인덱스를 확인합니다.

LeetCode 3의 상태 추적

right, 문자	left 이전	지우는 문자	삽입 후 구간	best
0, a	0	없음	a	1
1, b	0	없음	ab	2
2, b	0	a, b	b	2
3, a	2	없음	ba	2
4, 공백	2	없음	ba␣	3
5, c	2	없음	ba␣c	4

␣는 공백 표시입니다. 세 번째 문자를 넣기 전에 두 번 제거해야 하므로 `if` 한 번으로는 충분하지 않습니다.

중복 없는 최장 구간의 구현

```
def longest_unique(text):
    active, left, best = set(), 0, 0
    for right, char in enumerate(text):
        while char in active:
            active.remove(text[left])
            left += 1
        active.add(char)
        best = max(best, right - left + 1)
    return best
```

"abba"는 2, 빈 문자열은 0, 공백 하나는 1입니다. 반환값은 길이이며, 문자열 자체가 필요하면 최장 구간의 시작점도 저장합니다.

중복이 사라질 때까지 old left의 문자를 제거한 뒤 left를 옮깁니다. abba의 두 번째 b에서 a 하나만 지우면 여전히 b가 남으므로 if 한 번으로 끝낼 수 없습니다. 새 문자를 넣고 나서 길이를 계산합니다.

최장 구간의 정당성과 전체 이동 수

중복 문자의 이전 위치를 포함하는 구간은 새 문자와 함께 사용할 수 없습니다. 그 위치를 지날 때까지만 왼쪽 끝을 옮기므로 각 오른쪽 끝에서 가능한 가장 긴 구간을 검사합니다. 이전에 제외한 시작점은 이미 중복을 포함하므로 다시 유효해지지 않습니다.

포인터 이동: 오른쪽 n번, 왼쪽 최대 n번. 중첩된 `while`의 전체 실행 횟수: 최대 n번

right는 n번, left도 뒤로 가지 않아 최대 n번 움직입니다. 겉보기 중첩 반복을 n×n으로 세지 않고 전체 이동을 합하면 최대 2n입니다. set 연산이 기대 O(1)이므로 기대 시간 O(n), 동시에 든 문자 수가 최대 min(n,a)이므로 보조 공간 O(min(n,a))입니다. a는 문자 종류 수입니다.

LeetCode 209: 목표 합과 양수 조건

Minimum Size Subarray Sum은 합이 `target` 이상인 연속 구간의 최소 길이를 요구하며, 해당 구간이 없으면 0을 반환합니다.

입력 길이는 1부터 100000, 각 값은 1부터 10000, `target`은 1부터 10^9. 모든 원소와 목표가 양수

`target=6`, `nums=[1,2,4,2]`이면 `[2,4]` 또는 `[4,2]`의 길이 2가 답입니다. 시작점마다 합을 누적하는 단순 풀이는 최악 $O(n^2)$ 시간입니다.

유효한 구간의 기록과 축소

`total`은 현재 구간의 합입니다. 오른쪽 값을 더한 뒤 합이 목표 이상인 동안 현재 길이를 기록하고 왼쪽 값을 뺍니다.

동작	구간	total	best
1, 2까지 추가	[1,2]	3	미발견
4 추가, 길이 기록	[1,2,4]	7	3
1 제거, 길이 기록	[2,4]	6	2
2 제거, 축소 종료	[4]	4	2
마지막 2 추가, 기록	[4,2]	6	2

마지막에도 4를 제거한 뒤 합이 2가 되어 축소를 멈춥니다. 길이 기록이 제거보다 먼저

최단 구간의 구현

```
def min_subarray(target, nums):
    total = left = 0
    best = len(nums) + 1
    for right, value in enumerate(nums):
        total += value
        while total >= target:
            best = min(best, right - left + 1)
            total -= nums[left]
            left += 1
    return 0 if best > len(nums) else best
```

`best=n+1`: 유효한 구간을 아직 찾지 못한 상태

`target=8`, `[1,2,4]`이면 합이 부족하므로 0을 반환합니다.

right의 값을 먼저 더합니다. 목표 합을 만족하는 동안 현재 길이를 기록하고 왼쪽 값을 빼서 더 짧은 구간을 시험합니다. best=n+1은 아직 답을 못 찾았다는 표시이며, 가능한 구간 길이는 최대 n입니다.

최단 구간의 정당성과 비용

양수 값을 오른쪽에 더하면 합이 증가하고 왼쪽에서 빼면 감소합니다. 현재 합이 목표보다 작을 때 더 줄여도 유효한 구간이 나오지 않습니다.

합이 충분하면 길이를 기록한 뒤 줄입니다. 버린 시작점은 이미 유효한 길이를 기록했으며, 그 시작점을 이후에 다시 사용하면 더 긴 구간만 만듭니다. 따라서 더 짧은 답을 잃지 않습니다.

원소마다 합에 한 번 더하고 최대 한 번 뺍니다. 양 끝 이동의 합이 2n 이하이므로 전체 O(n) 시간입니다. 구간을 복사하지 않고 total, left, right, best만 저장하므로 보조 공간 O(1)입니다. 입력이 n칸인 것과 추가 작업 공간은 별도로 셉니다.

음수를 허용한 확장의 반례

공식 문제의 양수 조건을 없애고 `target=3`, `nums=[1,-1,3]`을 넣어 봅니다.

전체 합 3에서 길이 3을 기록합니다. 맨 앞의 1을 빼면 합이 2가 되어 축소를 멈추므로, 실제 답인 `[3]`의 길이 1을 놓칩니다.

다음 값 -1까지 제거하면 합이 다시 3이 됩니다. 음수를 빼면 합이 커질 수 있어 앞 페이지의 단조성 근거가 사라집니다. 이 반례는 공식 입력 범위 밖이며, 양수 조건이 왜 필요한지 확인하는 예입니다.

3차시 이해 확인

1. `nums=[1,2,4,2]`, 폭 2에서 마지막 구간의 합은 얼마인가요? 나가는 인덱스도 적습니다.
2. `"abba"`에서 `while`을 `if`로 바꾸면 처음 어긋나는 상태는 무엇인가요?
3. `for` 안의 `while`이 전체 O(n)번만 실행되는 근거는 무엇인가요?
4. `target=3`, `[1,-1,3]`에서 양수 전용 풀이가 반환하는 값과 실제 답을 비교

작업 취소와 스택의 복원 상태

편집기에서 작업을 두 번 수행한 뒤 한 번 취소하면 가장 최근 작업부터 되돌립니다. 이 순서를 후입선출(LIFO)이라고 합니다.

최솟값을 계속 보여 주는 기록 도구라면 마지막 값을 제거한 뒤 이전 최솟값도 복원해야 합니다. 현재 최솟값 하나만 저장해 두면 그 값이 제거되었을 때 정보가 부족합니다.

4차시는 값과 함께 저장할 보조 상태를 설계하고, 같은 꺼내기 순서로 후위 표기 계산을 수행합니다.

Python 리스트의 스택 연산

```
stack = []
stack.append(9)
stack.append(4)
print(stack[-1])     # 4: inspect without removal
right = stack.pop()  # 4
left = stack.pop()   # 9
```

스택의 맨 위: 리스트 끝. `append`와 끝의 `pop`은 상각 $O(1)$, `stack[-1]`은 $O(1)$. `pop(0)`은 뒤의 원소를 옮겨 $O(d)$ 시간이 듭니다.

빈 리스트에서 `pop()`이나 `[-1]`을 사용하면 예외가 납니다. d는 현재 깊이이며, 상각 비용은 여러 연산에 걸친 총비용으로 계산합니다.

LeetCode 155: 깊이별 최솟값

Min Stack은 `push`, `pop`, `top`, `getMin`의 상수 시간 연산을 요구합니다. 값은 부호 있는 32비트 정수이며 호출은 최대 30000회입니다. `pop`, `top`, `getMin`은 비어 있지 않은 스택에서만 호출됩니다.

매번 `min(stack)`을 계산하면 조회마다 O(d) 시간이 듭니다. 각 깊이에 `(값, 그 깊이까지의 최솟값)`을 함께 저장합니다.

새 최솟값은 `min(새 값, 이전 최솟값)`입니다. 마지막 쌍을 제거하면 바로 아래 쌍에 이전 최솟값이 남습니다.

중복 최솟값의 상태 추적

연산	저장한 (값, 최솟값)	getMin 결과
push(2)	[(2,2)]	2
push(2)	[(2,2),(2,2)]	2
push(5)	[(2,2),(2,2),(5,2)]	2
pop()	[(2,2),(2,2)]	2
pop()	[(2,2)]	2

같은 최소를 가진 항목이 각각 자기 깊이의 정보를 저장합니다. 값 2를 한 번 제거해도 아래의 2가 남으므로 최솟값은 유지됩니다.

Min Stack의 객체와 구현

```
class MinStack:
    def __init__(self):
        self.entries = []

    def push(self, val):
        minimum = min(val, self.entries[-1][1]) if self.entries else val
        self.entries.append((val, minimum))

    def pop(self):
        self.entries.pop()

    def top(self):
        return self.entries[-1][0]

    def getMin(self):
        return self.entries[-1][1]
```

각 튜플은 (현재 값, 이 깊이까지의 최솟값)입니다. pop은 마지막 튜플만 없애 이전 깊이의 최솟값을 다시 드러냅니다. getMin은 전체를 훑는 min 연산 없이 마지막 튜플의 두 번째 칸만 읽습니다.

entries[-1][1]에서 −1은 마지막 튜플, 1은 그 튜플의 두 번째 값입니다. 예를 들어 [(4,4),(7,4)]이면 top은 7, getMin은 4입니다. 빈 리스트에서는 조건식의 else 쪽 val만 평가하므로 없는 마지막 항목을 읽지 않습니다.

인스턴스의 독립 상태와 self

`self.entries`는 각 객체를 생성할 때 새 리스트로 만듭니다. `first.push(2)`는 Python이 `first`를 `self`로 전달하는 메서드 호출입니다.

```
first = MinStack()
second = MinStack()
first.push(2)
second.push(7)
print(first.getMin(), second.getMin())  # 2 7
```

클래스 본문에 `entries=[]` 하나를 두고 공유하면 서로 다른 스택의 값이 섞일 수 있습니다. 일반 함수에는 인스턴스가 자동으로 전달되지 않습니다.

Min Stack의 정당성과 비용

첫 항목의 최소는 그 값 자신입니다. 이전 항목까지의 최소가 맞다고 가정하면, 새 항목의 최소는 이전 최소와 새 값 중 작은 값이므로 역시 맞습니다. pop은 마지막 항목만 제거해 이전 깊이의 정보를 그대로 드러냅니다.

깊이 d에서 getMin과 top은 마지막 튜플 한 칸만 읽어 O(1)입니다. push·pop은 list 재할당을 포함한 분할상환 O(1)입니다. 각 깊이마다 값 두 개를 저장하므로 2d개 값, 즉 O(d) 공간입니다. getMin마다 전체 min을 계산하면 한 질의가 O(d)로 바뀝니다.

LeetCode 150: 후위 표기와 부분식

Evaluate Reverse Polish Notation은 연산자가 피연산자 뒤에 오는 식을 계산. `9 4 - -2 /`는 `(9-4)/(-2)`를 뜻하며 결과는 -2

토큰은 1부터 10000개, 피연산자는 -200부터 200의 정수, 연산자는 `+ - * /`입니다. 입력은 유효한 식이며 0으로 나누지 않습니다. 중간 결과와 답은 부호 있는 32비트 범위입니다.

- 숫자: 스택에 추가
- 연산자: 오른쪽, 왼쪽 순서로 꺼내 계산한 결과를 스택에 추가

후위 표기 계산의 상태 추적

토큰	계산 전 스택	수행하는 연산	계산 후 스택
9	[]	push(9)	[9]
4	[9]	push(4)	[9,4]
-	[9,4]	left=9, right=4	[5]
-2	[5]	push(-2)	[5,-2]
/	[5,-2]	5 / -2, 0 방향 절삭	[-2]

스택은 지금까지 계산을 마친 부분식의 값을 원래 순서로 보존합니다. 뺄셈과 나눗셈에서는 꺼낸 순서와 피연산자 순서를 구분해야 합니다.

음수 나눗셈의 절삭 방향

공식 문제는 0 방향으로 절삭합니다. Python의 `//`는 음의 무한대 방향으로 내림하므로 음수 몫에서 차이가 납니다.

식	Python //	0 방향 절삭
5 / 2	2	2
5 / -2	-3	-2
-5 / 2	-3	-2
-5 / -2	2	2

`abs(left) // abs(right)`로 크기를 계산하고 두 값의 부호가 다를 때 음수를 붙입니다. 부동소수점 변환 없이 정수 연산만 사용합니다.

`(left < 0) != (right < 0)`는 두 부호가 다른지 검사합니다. 5와 −2에서는 False != True가 True여서 −2, −5와 −2에서는 True != True가 False여서 +2입니다. 연산자 토큰 `"-"`와 숫자 토큰 `"-2"`는 서로 다른 문자열이므로 후자는 int로 변환됩니다.

후위 표기 계산의 구현

```
def rpn(tokens):
    stack = []
    for token in tokens:
        if token not in {"+", "-", "*", "/"}:
            stack.append(int(token))
            continue
        right, left = stack.pop(), stack.pop()
        if token == "+":
            value = left + right
        elif token == "-":
            value = left - right
        elif token == "*":
            value = left * right
        else:
            quotient = abs(left) // abs(right)
            value = -quotient if (left < 0) != (right < 0) else quotient
        stack.append(value)
    return stack[0]
```

연산자를 만나면 먼저 꺼낸 값이 오른쪽 피연산자입니다. 두 번째 값이 왼쪽이며 계산 결과 하나를 다시 넣습니다. 나눗셈은 절댓값끼리 나눈 몫에 부호를 붙여 0 방향 절삭을 구현합니다.

부분식 불변식과 계산 비용

숫자를 읽으면 계산이 끝난 부분식 하나가 생깁니다. 연산자를 읽으면 마지막 두 부분식의 값을 올바른 순서로 결합합니다. 이 과정을 반복하면 유효한 전체 식의 마지막에 값 하나가 남습니다.

`["9","4","-"]`는 5이고, 순서를 뒤집어 빼면 -5입니다. `[-2]`가 아니라 문자열 토큰 `["-2"]`를 입력하면 숫자 하나로 읽어 -2를 반환합니다.

t개 토큰 각각에서 최대 두 번 pop, 한 번 append, 산술 연산 한 번을 수행하여 O(t) 시간입니다. 문제의 중간 값은 32비트 범위라 산술을 O(1)로 셉니다. 아직 결합되지 않은 피연산자가 많이 쌓이면 스택 크기가 O(t)이므로 보조 공간도 O(t)입니다. 마지막 한 값만 남는다는 사실이 실행 중 공간을 줄이지는 않습니다.

4차시 이해 확인

1. `push(2), push(2), pop(), getMin()`의 마지막 결과와 보존된 정보를 설명
2. `MinStack` 객체 두 개가 같은 리스트를 공유하면 어떤 입력으로 오류를 찾을 수 있나요?
3. `["9","4","-"]`에서 첫 번째 `pop`은 어느 피연산자인가요?
4. `5//-2`와 공식 문제의 `5/-2`는 각각 얼마인가요?
5. `getMin`에서 `min(entries)`를 호출하면 조회 비용이 어떻게 달라지나요?

확인: 남은 튜플은 (2,2), 최소는 2입니다. 첫 스택에만 push하고 둘째 스택이 비어 있는지 확인하면 공유 오류를 찾습니다. 첫 pop은 4입니다. 5//−2는 −3, 요구값은 −2입니다. 매번 min을 계산하면 조회가 O(n)이 됩니다.

문제 조건에 따른 상태 선택

문제	저장한 상태	갱신 근거	비용
중복 없는 최장 구간	집합, 왼쪽 끝	중복 제거 후 삽입	평균 O(n) 시간
목표 합의 최단 구간	합, 왼쪽 끝	양수와 유효 길이 기록	O(n) 시간
Min Stack	깊이별 값과 최소	이전 최소의 복원	조회 O(1)
후위 표기 계산	부분식 값의 스택	오른쪽부터 꺼내 결합	O(t) 시간

시간을 계산할 때 포인터 이동, 집합 연산, 리스트 재할당을 구분합니다. 결과가 맞는 예제와 함께 상태가 깨지는 반례도 설명합니다.

독립 실습과 제출물

학생 실습에서 입력·출력·제약을 적은 뒤 `student_week_02.py`에 풀이를 작성합니다. 각 문제에 상태의 의미, 정당성, 시간·공간 비용, 직접 만든 경계 입력을 첨부

2주차 코드 · 풀이 읽기자료

2주차 자료 전체 · 교사용 진행안

핵심 참고자료

Erik Demaine·Srini Devadas, Introduction to Algorithms, MIT 6.006 (2011년 가을 학기). 알고리즘의 상태·정당성·비용 분석을 함께 공부합니다.

전체 문제 정의와 제약: 각 LeetCode 공식 페이지

- 3: Longest Substring Without Repeating Characters
- 209: Minimum Size Subarray Sum
- 155: Min Stack
- 150: Evaluate Reverse Polish Notation