5. Array-Based Sequence

2024학년도 가을학기

정보컴퓨터공학부 황원주 교수

부산대학교
PUSAN NATIONAL UNIVERSITY

강의내용

- Array(배열), List(리스트)
  - 선형자료구조(Linear data structure)
  - 필요성: 배열의 역사
  - 배열, 리스트
  - 시간복잡도

수업의 구성

배경/필요성→정의→ 예 → 이론 → (관찰+)구현 → 시간/공간복잡도

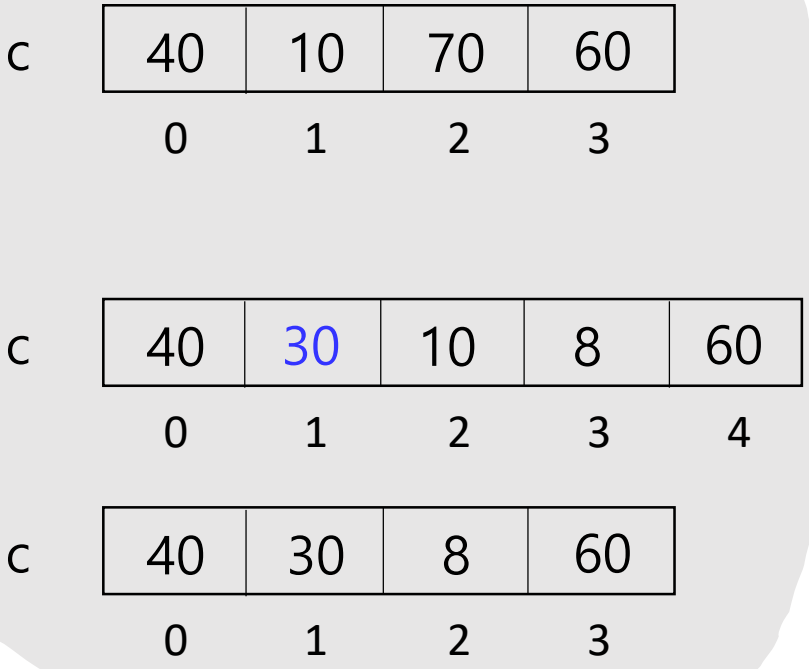
선형자료구조(Linear data structure)

- 선형자료구조 (Linear data structure)
  - 데이터 요소가 순차적(sequential)으로 배열되는 자료구조
  - 각 요소는 2개의 요소들(앞의 요소와 뒤의 요소)만 연결되어 있으므로, single run으로 모든 요소들을 순회(traversal)할 수 있음
  - 배열
  - 스택, 큐
  - 연결리스트
  - 해시 테이블
- 비선형자료구조 (Nonlinear data structure)
  - 데이터의 요소가 순차적으로 배열되지 않은 자료구조
  - 한 요소에 3개 이상의 요소들이 연결될 수 있으므로, single run으로 모든 요소들을 순회할 수 없음
  - 트리
  - 그래프

배열, 리스트

- 정의
  - 데이터를 연속적인 메모리 공간에 저장하고, 각 요소(element)는 인덱스를 이용하여 임의 접근(random access)할 수 있는 가장 많이 쓰이는 기본적인 자료구조
- 예/관찰: c = [40, 10, 70, 60]
  - 읽기: print(c[2])
  - 쓰기: c[2]=8
  - 삽입: insert(1, 30)
  - 삭제: delete(2)
  - 탐색: search(60)

메모리 공간



배열과 리스트의 차이점

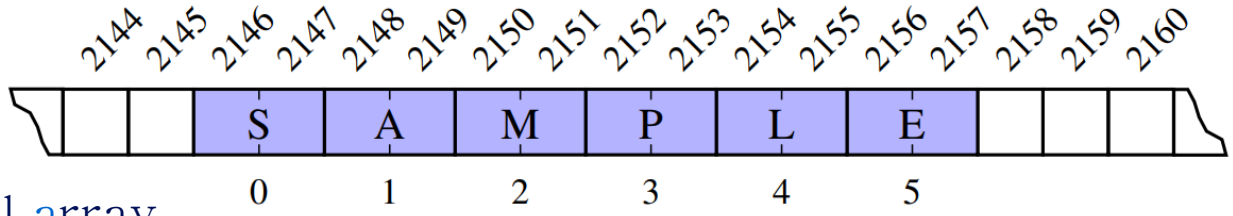
- 구현
  - Python의 list와 array로 구현

"Python Array uses less memory size than Python List."
"The short answer is: you may never use it because Lists and NumPy Arrays usually are much better alternatives. That's also the reason why rarely people knows Python has a built-in Array type."
[출처] Do You Know Python Has Built-In Array?

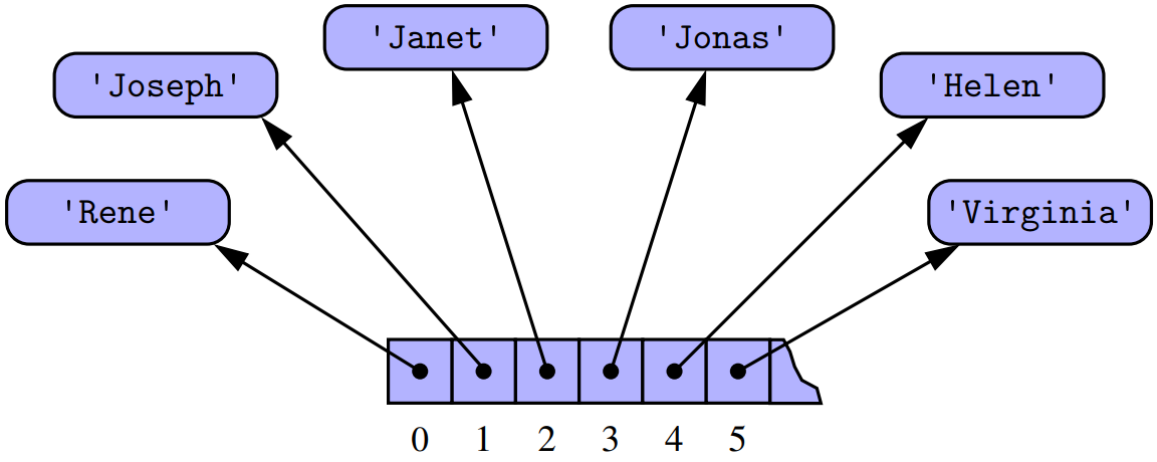
- 배열과 리스트의 차이점
  - 배열은 요소에 실제 값(데이터)이 저장되는 compact array지만, 리스트는 요소에 데이터가 아닌 데이터가 저장된 곳의 주소(address 또는 reference)가 저장되는 referential array이다
  - 리스트는 크기 (셀의 개수)가 필요에 따라 자동으로 증감하는 동적 배열(dynamic array)이나 배열은 정적 배열(static array)이다. 따라서 사용자는 리스트의 크기를 전혀 신경 쓰지 않아도 된다

Low-level array

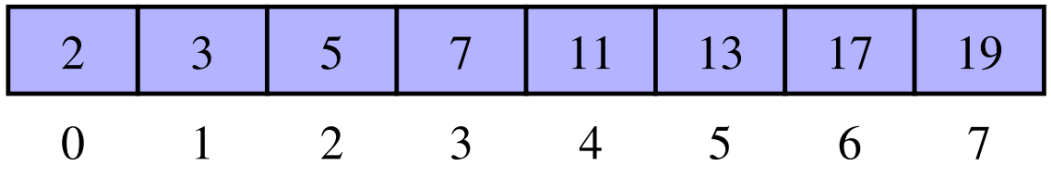
- 배열(Array)는 값들을 연속적으로 저장한 형태
- 할당할 요소의 개수가 미리 정해 짐
- Compact array
  - 실제 값(데이터)이 저장



- Referential array
  - 데이터가 저장된 곳의 주소(address 또는 reference)가 저장



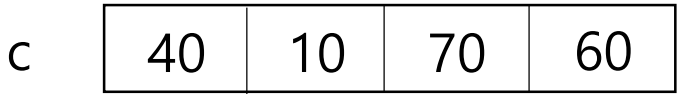
- Python의 array
  - Compact array



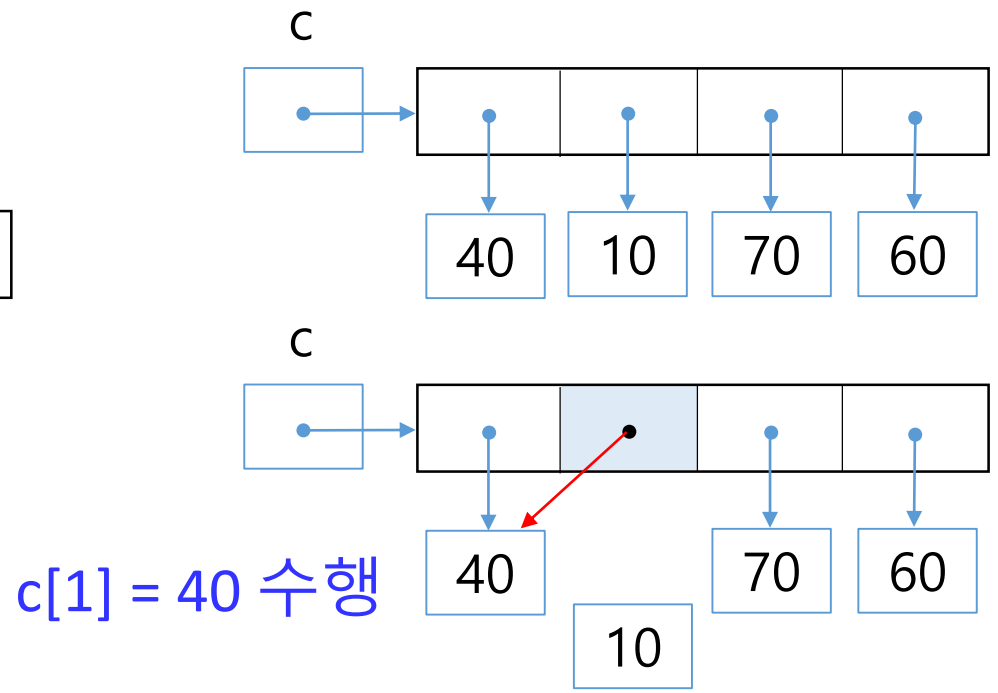

```
import array

# creating array
primes = array.array('i', [2, 3, 5, 7, 11, 13, 17, 19])
```

Type code	C Type	Python Type	Minimum size in bytes
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	Unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

- C 언어(Python) 배열의 요소에는 실제 값(데이터)이 저장되지만, Python 리스트의 요소에는 데이터가 아닌 **데이터가 저장된 곳의 주소(address 또는 reference)**가 저장된다
  - Python에서 c = [40, 10, 70, 60]을 수행하면, 4개의 정수들이 (a)와 같이 저장되는 것이 아니라 실제로 (b)와 같이 메모리에 저장된다.



(a) C언어의 경우



(b) Python의 경우

값이 저장된 곳의 주소 계산 가능

- C의 경우
  - 만약, 2번째 셀(cell)에 저장된 값을 알고 싶다면, 그 셀의 시작 주소는 c의 시작 주소 + 2*(값이 차지하는 byte 수)로 계산이 가능하다. (왜? 메모리상에 연속적으로 저장되어 있기 때문에)
  - 결국, **배열의 시작 주소, 저장된 값의 종류(바이트 개수), 몇 번째에 저장되어 있는지를 나타내는 인덱스** 세 가지 정보만으로 값이 저장된 곳의 주소를 계산할 수 있다 **[매우 중요한 사실!]**
  - c[2]의 시작 주소 = c[0]의 시작 주소 + 2*4 (index=2, sizeof(int)=4)
  - 인덱스로 메모리 주소를 계산하면 O(1) 시간에 **임의접근**이 가능하다
- Python의 경우
  - [중요] C 언어 배열의 셀에는 실제 값(데이터)이 저장되지만, Python 리스트의 셀에는 데이터가 아닌 데이터가 저장된 곳의 주소(address 또는 reference)가 저장된다
  - 항상 객체의 주소만 저장하기 때문에, 리스트의 셀의 크기를 메모리의 주소를 표현할 수 있는 (4 바이트 또는) 8 바이트로 고정하면 된다.
  - 인덱스로 메모리 주소를 찾으면 O(1) 시간 **임의접근**이 가능하다

리스트는 동적배열!

- C 언어의 배열과의 또 다른 중요한 차이점은 list의 크기 (셀의 개수)가 필요에 따라 자동으로 증가, 감소한다는 것이다

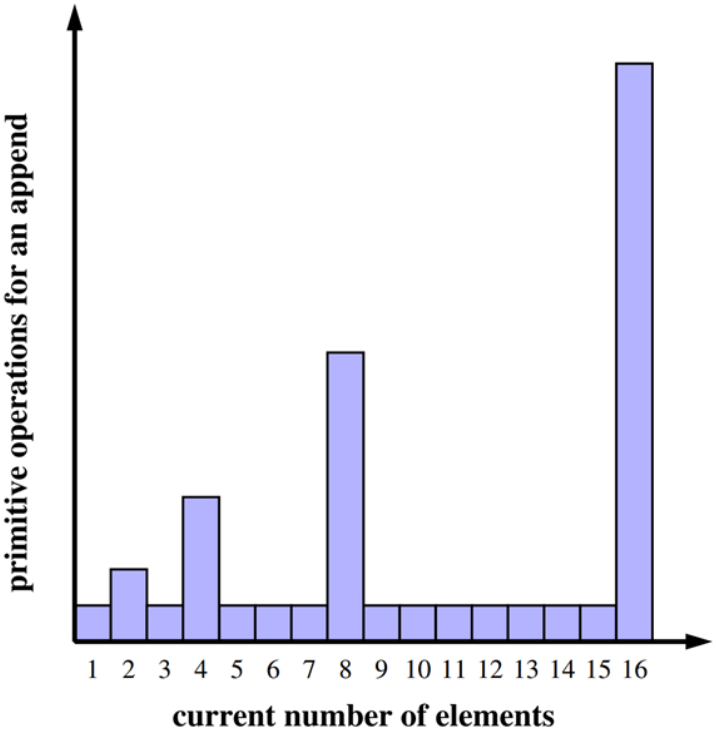
```
>>> A = []
>>> sys.getsizeof(A)
64                          # 빈 리스트도 64바이트 할당
>>> A.append("Python") # append 후 96바이트로 재 할당
>>> sys.getsizeof(A)
96
```

- 그래서 append 또는 insert 연산을 위한 공간(메모리)이 부족하면 더 큰 메모리를 할당받아 새로운 리스트를 만들고 이전 리스트의 값을 모두 이동한다. 반대로 pop 연산을 하면서 실제 저장된 값의 개수가 리스트 크기에 비해 충분히 작다면 더 작은 크기의 리스트를 만들고 모두 이동한다 → 마치, 식구가 많으면 큰 집으로 이사하고, 식구가 줄어들면 작은 집으로 이사하는 것과 같은 방식임
- 이렇게 필요에 따라 크기가 변하는 배열을 동적 배열 (dynamic array)라 부른다 (C에서의 배열은 동적 배열이 아니므로 사용자가 직접 상황에 맞게 처리해야 한다) 따라서 사용자는 배열의 크기를 전혀 신경 쓰지 않아도 된다 [매우 중요]

- 동적 배열인 리스트를 위해선 python 내부적으로 현재 리스트의 크기(capacity)와 리스트에 저장된 실제 값의 개수(n)를 항상 알고 있어야 한다 (이를 위한 내부 변수가 필요함) 이런 추가 정보를 위한 메모리가 필요하므로 빈 리스트 A는 0바이트보다 클 수 밖에 없다 (초기값 예: `capacity = 1, n = 0`)
- 추가 메모리 이외에 연산 시간에도 영향을 준다. 특정 append 연산에서 메모리를 늘려야 한다면 이전 리스트의 값을 새로운 리스트로 일일이 이동해야 한다 (아래와 같은 방식으로)

```
def append(A, value):
    1. if A.capacity == A.n:  # resize 필요!
        a. allocate a new list B with larger memory and
           update B.capacity and B.n
        b. for i in range(A.n):
            B[i] = A[i]
        c. dispose A
        d. A = B # B 이름을  A로  바꿈
    2. A[n] = value
    3. A.n += 1
```

- 단계 1.b에서 A에 저장된 값의 개수만큼 시간이 걸림: O(n)
  - 결국, append 연산이 resize가 일어나지 않는 경우에는 O(1) 시간, 일어나는 경우에는 O(n) 시간이 걸림
- 그러나 resize가 append할 때마다 발생하는 것이 아니라 가끔 발생하는 것 (크기가 2^1, 2^2, 2^3, … 일 때만 발생하는 것)이므로 평균 시간을 계산해보면 O(n) 시간보다 훨씬 작다
- 크기가 2배 씩 증가하거나 반으로 감소하는 경우에 append, pop 연산 시간은 평균적으로 O(1)임을 증명할 수 있다. (이에 대한 내용은 뒤의 **hash table 자료구조에서 다시 자세히 언급**)
- 즉, 평균(amortized analysis)적으로 따지면 한 번의 append 연산의 평균비용이 O(1)이라는 의미이다!



시간복잡도

예) C = [40, 10, 70, 60]

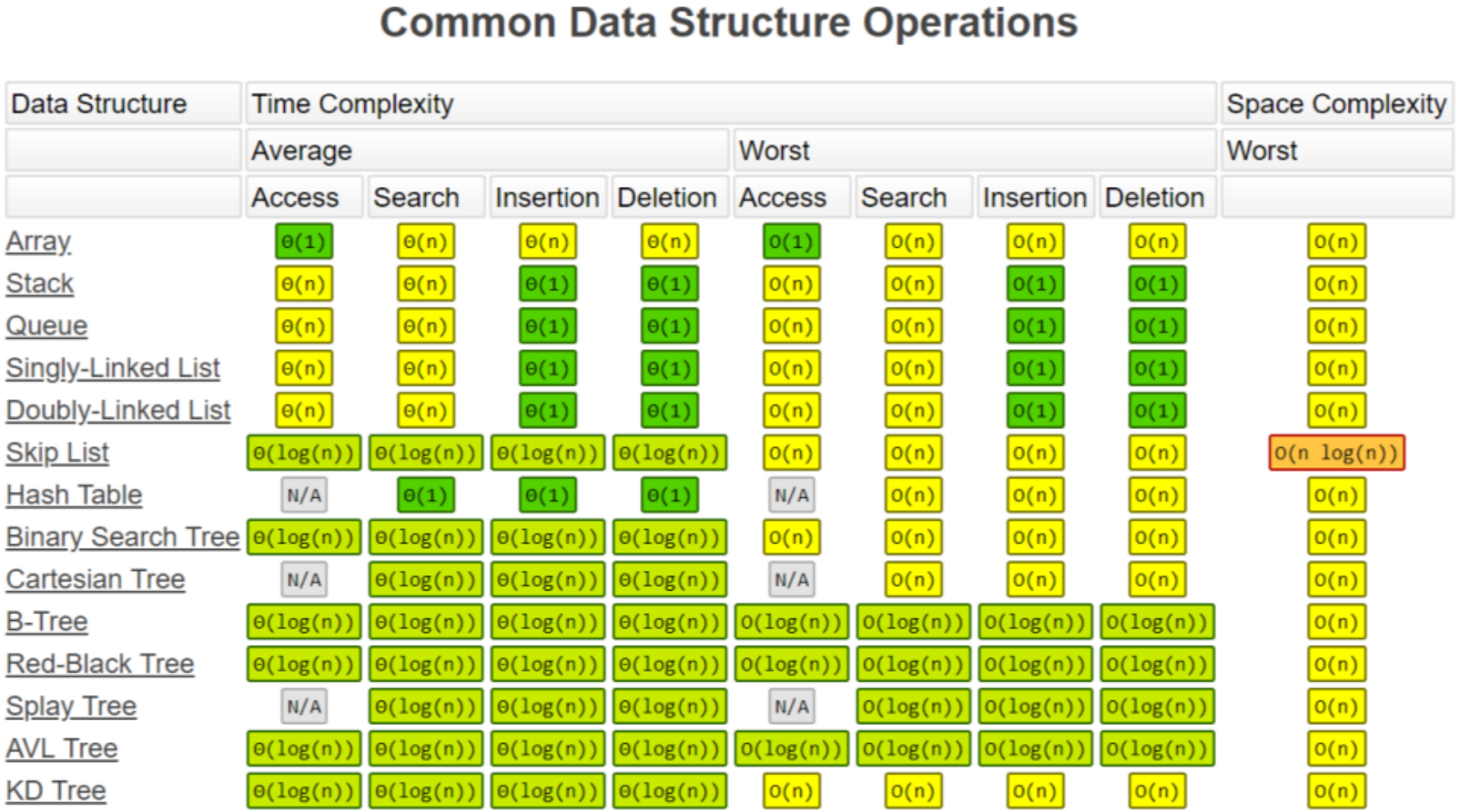
- **읽기/쓰기**
  - C[i] = C[j] + 1 : C[j] 값을 읽은 후(단위시간) 1을 더하고(단위시간) C[i]에 씀(단위시간) (읽기/쓰기: O(1) 시간)
- **삽입**
  - C.append(50) : 맨 오른쪽에 삽입하므로 평균적으로 O(1) 시간 필요
  - C.insert(0, 10) : 맨 앞에 10을 삽입하므로 C[0], … , C[len(C)−1]을 모두 오른쪽으로 이동하기 위한 C의 크기만큼 시간 필요. 최악의 경우 (+ 평균적인 경우)에 O(len(C)) = O(n) 시간 필요
- **삭제**
  - C.pop() : 맨 오른쪽 요소를 반환하고 삭제하므로 평균적으로 O(1) 시간 필요
  - C.pop(2) : index 2인 요소를 반환하고 삭제하므로 C[3], … , C[len(C)−1] 값이 왼쪽으로 한 칸씩 이동하는 시간 필요, 최악의 경우가 C.pop(0)일 때이므로 O(len(C)) = O(n) 시간 필요
  - C.remove(9) : 처음으로 등장하는 9를 찾아 제거. 최악의 경우 (+ 평균적인 경우)엔 O(n) 시간 필요
- **탐색**
  - C.index(9), C.count(9) : 9가 처음으로 등장하는 index와 총 횟수를 계산해야 하므로 최악의 경우 O(n) 시간 필요

파이썬 list 객체의 시간복잡도

Operation	Average Case	Amortized Worst Case
Copy	O(n)	O(n)
Append[1]	O(1)	O(1)
Pop last	O(1)	O(1)
Pop intermediate[2]	O(n)	O(n)
Insert	O(n)	O(n)
Get Item	O(1)	O(1)
Set Item	O(1)	O(1)
Delete Item	O(n)	O(n)
Iteration	O(n)	O(n)
Get Slice	O(k)	O(k)
Del Slice	O(n)	O(n)
Set Slice	O(k+n)	O(k+n)
Extend[1]	O(k)	O(k)
Sort	O(n log n)	O(n log n)
Multiply	O(nk)	O(nk)
x in s	O(n)	
min(s), max(s)	O(n)	
Get Length	O(1)	O(1)

[출처: https://wiki.python.org/moin/TimeComplexity]

자료구조 시간복잡도와 공간복잡도



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

[출처: http://bigocheatsheet.com/]

코딩테스트

LeetCode 문제

- 125. Valid Palindrome
- 209. Minimum Size Subarray Sum
- 724. Find Pivot Index
- 1480. Running Sum of 1d Array
- 560. Subarray Sum Equals K
- 728. Self Dividing Numbers

125. Valid Palindrome

125. Valid Palindrome

Easy 2262 4113 Add to List Share

Given a string `s`, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

Example 1:

```
Input: s = "A man, a plan, a canal: Panama"
Output: true
Explanation: "amanaplanacanalpanama" is a palindrome.
```

Example 2:

```
Input: s = "race a car"
Output: false
Explanation: "raceacar" is not a palindrome.
```

- Palindrome은 왼쪽부터 읽어도 오른쪽부터 읽어도 같은 문자열을 말한다
  - 예: 우영우, 기러기, radar, madam, 소주 만병만 주소 등

Two pointers technique

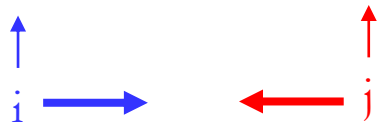
- 정의
  - Two pointers technique은 일반적으로 **탐색**에 사용되며 주어진 자료구조 상의 하나의 루프에서 **두 개의 포인터**를 사용한다.
  - string, array, linked list와 관련된 코딩 테스트를 해결할 때 자주 사용되는 접근 방법이다
- 목적
  - Two pointers technique을 사용하기 전에 대부분의 경우 자료구조는 어떤 방식으로든 **정렬**하며, 이렇게 하면 단 하나의 루프에서 두 개의 포인터를 사용하여 각 항목을 한 번만 탐색하므로 시간 복잡도를 O(n2) 또는 O(n3)에서 O(n)으로 줄이는데 도움이 된다.
  - 따라서 입력 string/array/linked list의 정렬 여부에 따라 Two pointers technique은 (정렬이 안되어 있을 때) O(n log n) 시간 복잡도 또는 (정렬이 되어 있을 때) 더 나은 O(n)을 취할 수 있다.

Two pointers technique 종류

- Opposite directional
  - 한 포인터는 **처음**에서 시작하고 다른 포인터는 **끝**에서 시작한다. 이 두 포인터들은 어떤 조건을 만족할 때까지 서로 마주보며 움직인다.
    문제: 125, 167, 283, 344, 27
- Equi-directional
  - slow-runner and fast-runner: 한 포인터가 다른 포인터보다 앞서게 하여 병합 지점이나 중간 위치, 길이 등을 판별할 때 유용하게 사용한다. 보통 fast-runner는 두 칸씩 건너뛰고 slow-runner는 한 칸씩 이동하게 한다. 이 때 fast-runner가 끝에 도착하면, slow-runner는 정확히 **중간 지점에 도착**하게 된다. 이와 같이 중간 위치를 찾아내면, 여기서부터 값을 비교하거나 뒤집기를 시도하는 등 여러모로 활용할 수 있다.
    문제: 234, 141, 3, 26
  - sliding window: 고정 사이즈의 윈도우가 이동하면서 윈도우 내에 있는 데이터를 이용하여 문제를 풀이하는 알고리즘. 주로 정렬된 배열을 대상으로 하는 two pointer technique과 달리 슬라이딩 윈도우는 **정렬여부에 관계없이** 사용.
    문제: 209, 239, 76, 424

관찰 1: Opposite directional two pointers technique

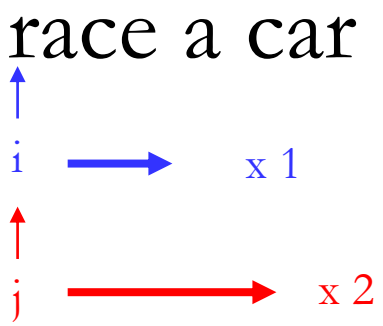
race a car




```
def isPalindrome(s):
    i, j = 0, len(s) - 1

    while i < j:
        while i < j and not s[i].isalnum():
            i += 1
        while i < j and not s[j].isalnum():
            j -= 1

        if s[i].lower() != s[j].lower():
            return False

        i += 1
        j -= 1

    return True
```

#isalnum() : 문자열이 영문이나 숫자로 이루어져 있으면 True를 반환하고, 공백, 특수문자가 있으면 False를 반환

관찰 2: Equi-directional two pointers technique (slow-runner and fast-runner)



- j가 문자열의 끝에 도달할 때까지 이동, 이 때 i는 문자열의 중간까지 이동함(i는 이동하면서 값들을 리스트에 저장)
- 리스트에 저장된 값을 앞에서부터 읽고,
- 문자열을 j가 위치한 곳에서부터 뒤에서 읽으면서 비교

209. Minimum Size Subarray Sum

209. Minimum Size Subarray Sum

Medium 4401 150 Add to List Share

Given an array of positive integers `nums` and a positive integer `target`, return the minimal length of a **contiguous subarray** $[nums_l, nums_{l+1}, \ldots, nums_{r-1}, nums_r]$ of which the sum is greater than or equal to `target`. If there is no such subarray, return `0` instead.

Example 1:

```
Input: target = 7, nums = [2,3,1,2,4,3]
Output: 2
Explanation: The subarray [4,3] has the minimal length under the problem constraint.
```

Example 2:

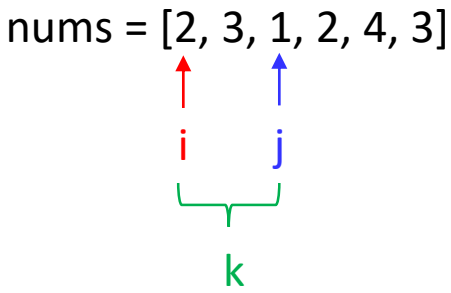
```
Input: target = 4, nums = [1,4,4]
Output: 1
```

Example 3:

```
Input: target = 11, nums = [1,1,1,1,1,1,1,1]
Output: 0
```

관찰 1 : Brute force

- 가능한 모든 subarray의 합을 계산하고 조건을 만족할 마다 minimal length를 업데이트



- 풀이

시간복잡도: $O(n^3)$

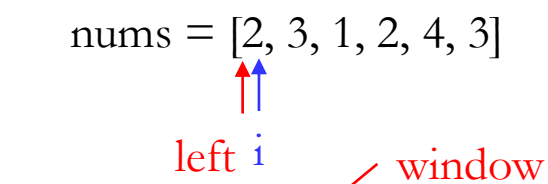
```
def minSubArrayLen(target, nums):
    min_len = float('inf')
    for i in range(0, len(nums) + 1):
        for j in range(i, len(nums)):
            sum = 0;
            for k in range(i, j + 1):
                sum += nums[k]
            if (sum >= target):
                min_len = min(min_len, (j - i + 1))
                break #Found the smallest subarray with sum>=s starting with index i,
hence move to next index
    return min_len if (min_len != (len(nums) + 1)) else 0
```

관찰 2: sliding window

- 리스트의 원소가 모두 양의 정수이므로 sliding window를 고려할 수 있음
- window 내 합을 target보다 크거나 같게 유지하고 가능한 가장 작은 window size를 유지하기 위한 최적의 움직임은?
- sliding window = expanding phase + shrinking phase
  - expanding phase: 조건을 만족할 때까지 right (코드에서는 i)를 증가
  - shrinking phase: 조건을 만족하면 left를 증가

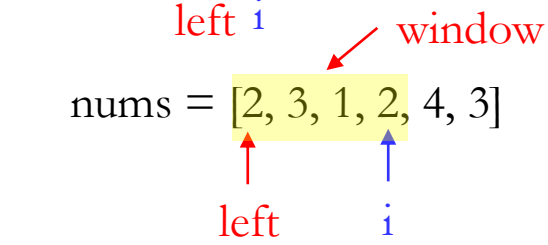
조건: sum ≥ target (=7)

초기 값:



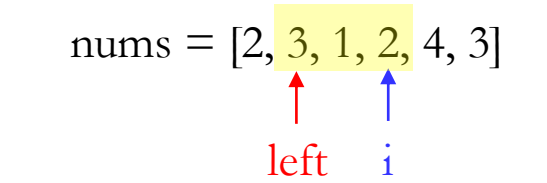
sum = 0
min_len = infinite

expanding phase:



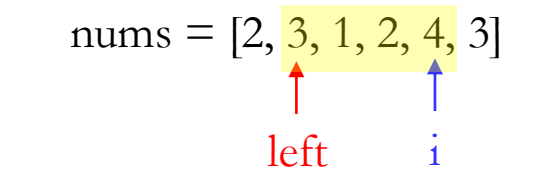
sum = 8
min_len = 4

shrinking phase:



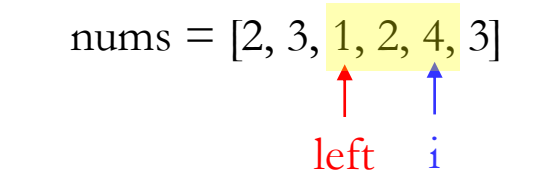
sum = 6
min_len = 3

expanding phase:



sum = 10
min_len = 4

shrinking phase:



sum = 7
min_len = 3

풀이

```
def minSubArrayLen(target, nums):
    min_len = float('inf')
    left = 0
    sum = 0
    for i in range(0, len(nums)):
        sum += nums[i]
        while (sum >= target):
            min_len = min(min_len, i + 1 - left)
            sum -= nums[left]
            left += 1
    return min_len if (min_len != (len(nums) + 1)) else 0
```

- 시간복잡도: O(n)
  - 각 원소는 포인터 i에 의해 한번 그리고 포인터 left에 의해 한번 거의 두 번씩 방문되어짐.

724. Find Pivot Index

724. Find Pivot Index

Easy 2028 327 Add to List Share

Given an array of integers `nums`, calculate the **pivot index** of this array.

The **pivot index** is the index where the sum of all the numbers **strictly** to the left of the index is equal to the sum of all the numbers **strictly** to the index's right.

If the index is on the left edge of the array, then the left sum is `0` because there are no elements to the left. This also applies to the right edge of the array.

Return *the **leftmost pivot index***. If no such index exists, return -1.

Example 1:

```
Input: nums = [1,7,3,6,5,6]
Output: 3
Explanation:
The pivot index is 3.
Left sum = nums[0] + nums[1] + nums[2] = 1 + 7 + 3 = 11
Right sum = nums[4] + nums[5] = 5 + 6 = 11
```

Example 2:

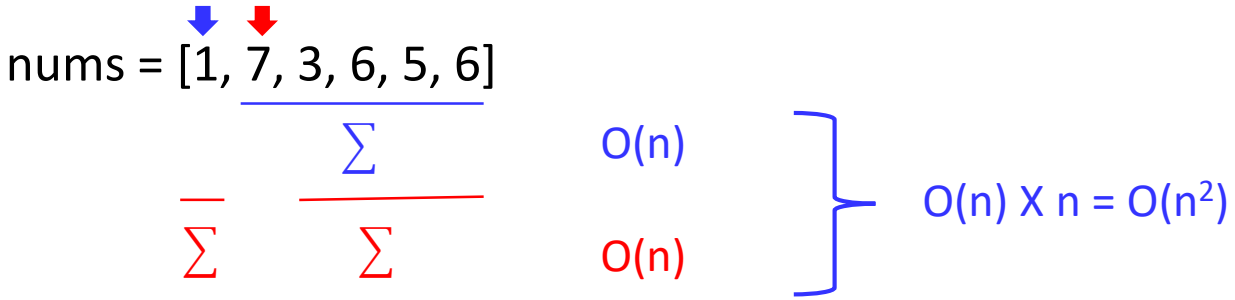
```
Input: nums = [1,2,3]
Output: -1
Explanation:
There is no index that satisfies the conditions in the problem statement.
```

Example 3:

```
Input: nums = [2,1,-1]
Output: 0
Explanation:
The pivot index is 0.
Left sum = 0 (no elements to the left of index 0)
Right sum = nums[1] + nums[2] = 1 + -1 = 0
```

관찰 1 : Brute force

- Pivot을 순서대로 하나씩 잡고 왼쪽과 오른쪽의 합을 구함



관찰 2 : Prefix sum algorithm

- 전체 부분합을 구한 후 Pivot을 순서대로 하나씩 잡고 왼쪽과 오른쪽의 합을 구함

왼쪽 오른쪽

∑ 1이 pivot일 때

1 ∑ - 1 7이 pivot일 때

7+1 ∑ - 1-7 3이 pivot일 때

전체 부분합: O(n)
왼쪽과 오른쪽의 합: O(1) X n = O(n)
→ O(n) + O(n) = O(n)

풀이

```
def pivotIndex(nums):
    s = sum(nums)
    leftsum = 0
    for i, x in enumerate(nums):
        if leftsum == (s - leftsum - x):
            return i
        leftsum += x
    return -1
```

Prefix sum algorithm

- Prefix sum
  - 부분합(partial sum): 0~k까지의 합
  - 구간합(prefix sum): a~b까지의 합
- 왜 prefix sum algorithm인가?
  - (예) A=[6,3,−2,4,−1,0,−5]에서 a에서 b의 구간합을 요구하는 query가 천 만개 들어온다 어떻게 해결할 것인가?
  - (예) A에서 [2,5]사이의 합을 구하라. → n

```
A=[6,3,-2,4,-1,0,-5]
sum=0

for i in range(2,6):
    sum+=A[i]

print(sum)
```

4

```
A=[6,3,-2,4,-1,0,-5,...]
          n
for j in range(m):
    sum=0

    for i in range(start,end):
        sum+=A[i]

    print(sum)
```

n

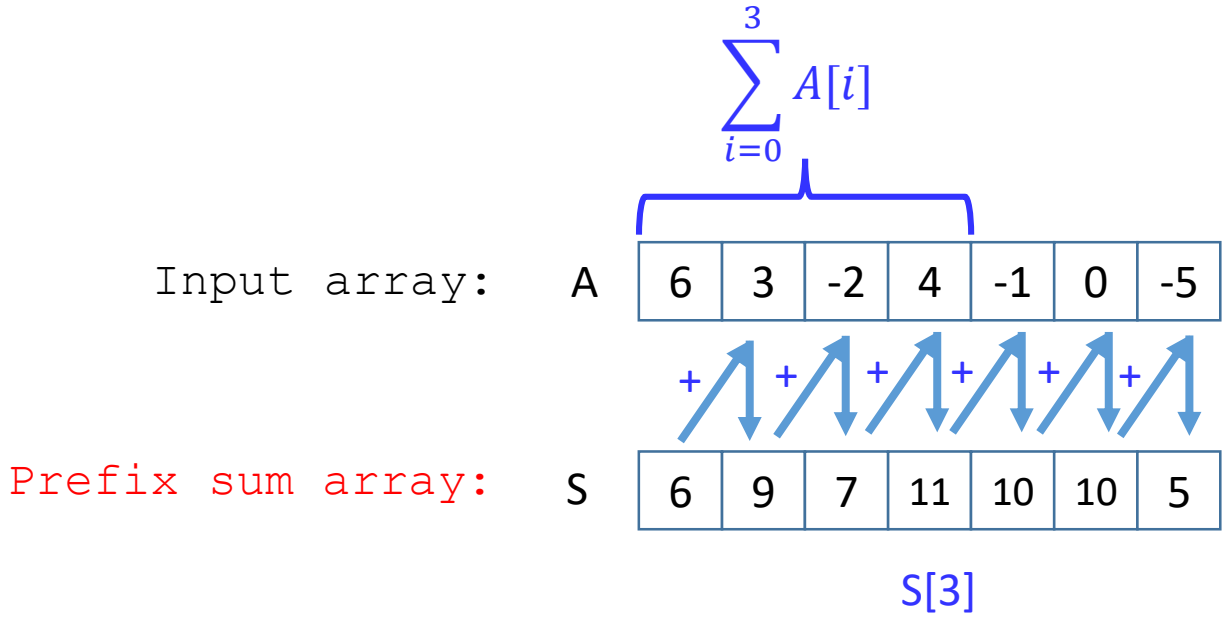
#m번의 query

m

시간복잡도: O(mn)

Prefix sum algorithm

A=[6,3,-2,4,-1,0,-5]



$$S[n] = \sum_{i=0}^{n} A[i]$$

Example:

i =	0	1	2	3	4	5	6
A =	6	3	-2	4	-1	0	-5

Prefix Sum Array:

i =	0	1	2	3	4	5	6
S =	6	9	7	11	10	10	5

Calculate the sum between range[0,4]?

Answer : 10(=S[4]) } O(1)

Calculate the sum between range[0,6]?

Answer : 5(=S[6]) } O(1)

Calculate the sum between range[2,6]?

sum between range[0,6]

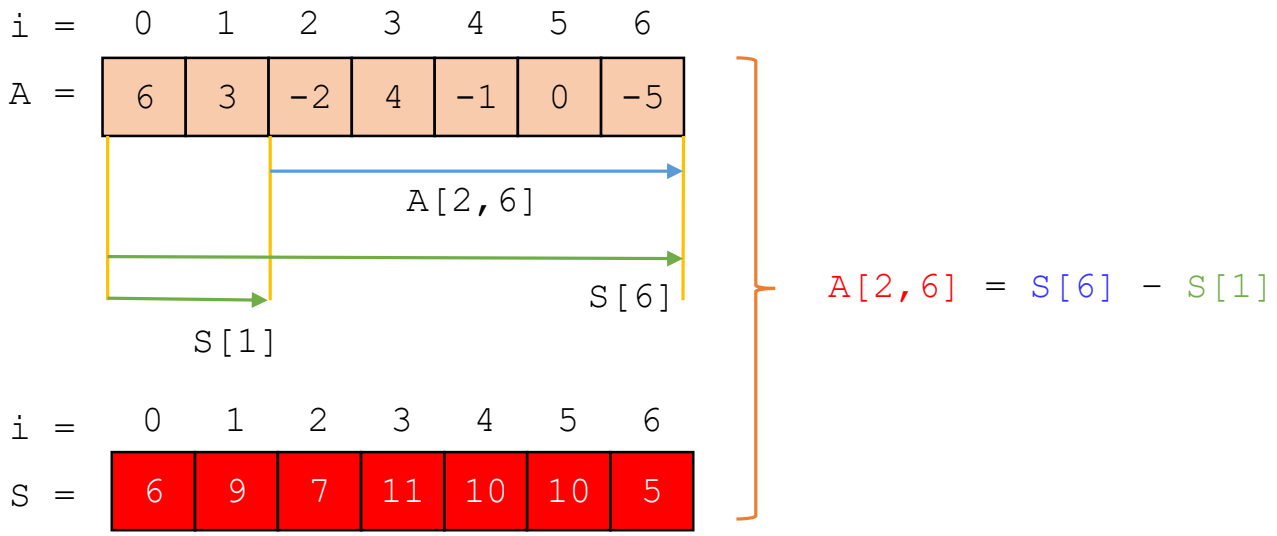
sum between range[0,6] =
  sum between range[0,1] +
  sum between range[2,6]

S[6] = S[1] + sum between range[2,6]

S[6] - S[1] = sum between range[2,6]

sum between range[2,6] = S[6] - S[1]
= 5 - 9
= - 4

Generalization:



Prefix Sum Array:

To calculate the sum between range [i,j]

Formula:

A[i,j] = S[j] - S[i-1]

calculate the sum between range [3,5]?

```
A[3,5] = S[5] - S[3-1]
       = S[5] - S[2]
       = 10 - 7
       = 3
```

O(1)

- **시간복잡도: O(n)=O(n)+O(1)**
  - (전처리) 크기 n인 배열의 prefix sum array 계산: O(n)
  - 구간합 query를 수행: O(1)

감사합니다!

부산대학교
PUSAN NATIONAL UNIVERSITY