소수 (Prime Number)

- **소수**란 1보다 큰 자연수 중에서 1과 자기 자신을 제외한 자연수로는 나누어떨어지지 않는 자연수입니다.
  - 6은 1, 2, 3, 6으로 나누어떨어지므로 소수가 아닙니다.
  - 7은 1과 7을 제외하고는 나누어떨어지지 않으므로 소수입니다.
- 코딩 테스트에서는 어떠한 자연수가 소수인지 아닌지 판별해야 하는 문제가 자주 출제됩니다.

소수의 판별: 기본적인 알고리즘 (Python)

```python
# 소수 판별 함수(2이상의 자연수에 대하여)
def is_prime_number(x):
    # 2부터 (x - 1)까지의 모든 수를 확인하며
    for i in range(2, x):
        # x가 해당 수로 나누어떨어진다면
        if x % i == 0:
            return False # 소수가 아님
    return True # 소수임

print(is_prime_number(4))
print(is_prime_number(7))
```

실행 결과

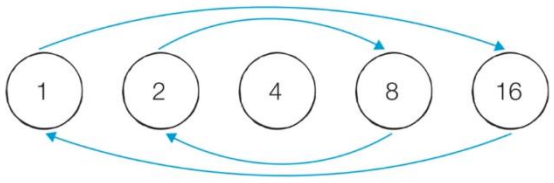
```
False
True
```

소수의 판별: 기본적인 알고리즘 성능 분석

- 2부터 $X-1$까지의 모든 자연수에 대하여 연산을 수행해야 합니다.
  - 모든 수를 하나씩 확인한다는 점에서 시간 복잡도는 $O(X)$입니다.

약수의 성질

- **모든 약수가 가운데 약수를 기준으로 곱셈 연산에 대해 대칭**을 이루는 것을 알 수 있습니다.
  - 예를 들어 16의 약수는 1, 2, 4, 8, 16입니다.
  - 이때 2 X 8 = 16은 8 X 2 = 16과 대칭입니다.
- 따라서 우리는 특정한 자연수의 모든 약수를 찾을 때 가운데 약수(제곱근)까지만 확인하면 됩니다.
  - 예를 들어 16이 2로 나누어떨어진다는 것은 8로도 나누어떨어진다는 것을 의미합니다.



소수의 판별: 개선된 알고리즘 (Python)

```python
import math

# 소수 판별 함수 (2이상의 자연수에 대하여)
def is_prime_number(x):
    # 2부터 x의 제곱근까지의 모든 수를 확인하며
    for i in range(2, int(math.sqrt(x)) + 1):
        # x가 해당 수로 나누어떨어진다면
        if x % i == 0:
            return False # 소수가 아님
    return True # 소수임

print(is_prime_number(4))
print(is_prime_number(7))
```

실행 결과

```
False
True
```


소수의 판별: 개선된 알고리즘 성능 분석

- 2부터 X의 제곱근(소수점 이하 무시)까지의 모든 자연수에 대하여 연산을 수행해야 합니다.
  - 시간 복잡도는 $O(N^{\frac{1}{2}})$입니다.

다수의 소수 판별

- 하나의 수에 대해서 소수인지 아닌지 판별하는 방법을 알아보았습니다.
- 하지만 특정한 수의 범위 안에 존재하는 모든 소수를 찾아야 할 때는 어떻게 할까요?
  - **에라토스테네스의 체 알고리즘**을 사용할 수 있습니다.

에라토스테네스의 체 알고리즘

- 다수의 자연수에 대하여 소수 여부를 판별할 때 사용하는 대표적인 알고리즘입니다.
- 에라토스테네스의 체는 N보다 작거나 같은 모든 소수를 찾을 때 사용할 수 있습니다.
- 에라토스테네스의 체 알고리즘의 **구체적인 동작 과정**은 다음과 같습니다.
  1. 2부터 N까지의 모든 자연수를 나열한다.
  2. 남은 수 중에서 아직 처리하지 않은 가장 작은 수 i를 찾는다.
  3. 남은 수 중에서 i의 배수를 모두 제거한다 (i는 제거하지 않는다).
  4. 더 이상 반복할 수 없을 때까지 2번과 3번의 과정을 반복한다.

에라토스테네스의 체 알고리즘 동작 예시

- **[초기 단계]** 2부터 26까지의 모든 자연수를 나열합니다. $(N = 26)$

2	3	4	5	6
7	8	9	10	11
12	13	14	15	16
17	18	19	20	21
22	23	24	25	26

에라토스테네스의 체 알고리즘 동작 예시

- **[Step 1]** 아직 처리하지 않은 가장 작은 수 2를 제외한 2의 배수는 모두 제거합니다.

2	3	4	5	6
7	8	9	10	11
12	13	14	15	16
17	18	19	20	21
22	23	24	25	26

에라토스테네스의 체 알고리즘 동작 예시

- **[Step 2]** 아직 처리하지 않은 가장 작은 수 3을 제외한 3의 배수는 모두 제거합니다.

2	3	4	5	6
7	8	9	10	11
12	13	14	15	16
17	18	19	20	21
22	23	24	25	26

에라토스테네스의 체 알고리즘 동작 예시

- **[Step 3]** 아직 처리하지 않은 가장 작은 수 5를 제외한 5의 배수는 모두 제거합니다.

2	3	4	5	6
7	8	9	10	11
12	13	14	15	16
17	18	19	20	21
22	23	24	25	26

에라토스테네스의 체 알고리즘 동작 예시

- **[Step 4]** 마찬가지의 과정을 반복했을 때 최종적인 결과는 다음과 같습니다.

2	3	4	5	6
7	8	9	10	11
12	13	14	15	16
17	18	19	20	21
22	23	24	25	26

에라토스테네스의 체 알고리즘 (Python)

```python
import math

n = 1000 # 2부터 1,000까지의 모든 수에 대하여 소수 판별
# 처음엔 모든 수가 소수(True)인 것으로 초기화(0과 1은 제외)
array = [True for i in range(n + 1)]

# 에라토스테네스의 체 알고리즘 수행
# 2부터 n의 제곱근까지의 모든 수를 확인하며
for i in range(2, int(math.sqrt(n)) + 1):
    if array[i] == True: # i가 소수인 경우(남은 수인 경우)
        # i를 제외한 i의 모든 배수를 지우기
        j = 2
        while i * j <= n:
            array[i * j] = False
            j += 1

# 모든 소수 출력
for i in range(2, n + 1):
    if array[i]:
        print(i, end=' ')
```

에라토스테네스의 체 알고리즘 성능 분석

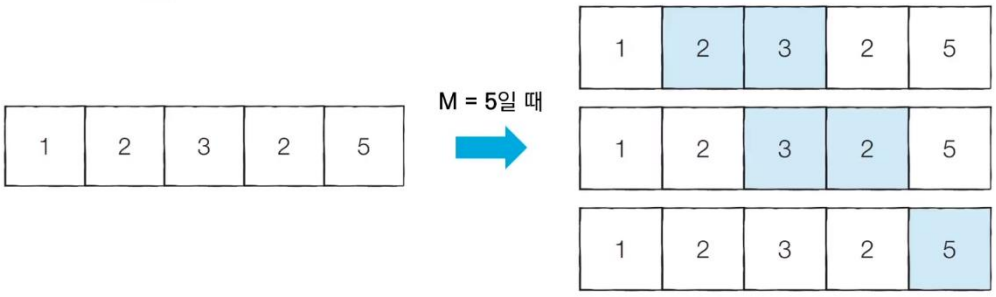
- 에라토스테네스의 체 알고리즘의 시간 복잡도는 사실상 선형 시간에 가까울 정도로 매우 빠릅니다.
  - 시간 복잡도는 $O(NloglogN)$입니다.
- 에라토스테네스의 체 알고리즘은 다수의 소수를 찾아야 하는 문제에서 효과적으로 사용될 수 있습니다.
  - 하지만 각 자연수에 대한 소수 여부를 저장해야 하므로 **메모리가 많이 필요**합니다.
  - **10억**이 소수인지 아닌지 판별해야 할 때 에라토스테네스의 체를 사용할 수 있을까요?

투 포인터 (Two Pointers)

- **투 포인터 알고리즘**은 리스트에 순차적으로 접근해야 할 때 두 개의 점의 위치를 기록하면서 처리하는 알고리즘을 의미합니다.
- 흔히 2, 3, 4, 5, 6, 7번 학생을 지목해야 할 때 간단히 '2번부터 7번까지의 학생'이라고 부르곤 합니다.
- 리스트에 담긴 데이터에 순차적으로 접근해야 할 때는 **시작점**과 **끝점** 2개의 점으로 접근할 데이터의 범위를 표현할 수 있습니다.

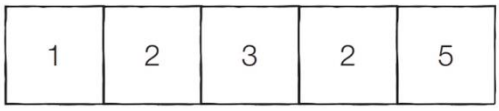
특정한 합을 가지는 부분 연속 수열 찾기: 문제 설명

- N개의 자연수로 구성된 수열이 있습니다.
- **합이 M인 부분 연속 수열의 개수**를 구해보세요.
- 수행 시간 제한은 **O(N)**입니다.



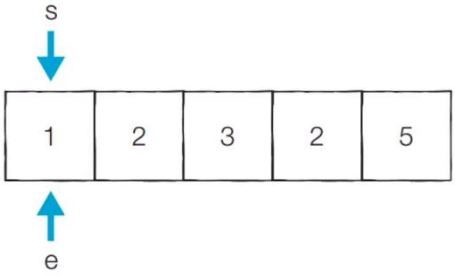
특정한 합을 가지는 부분 연속 수열 찾기: 문제 해결 아이디어

- 투 포인터를 활용하여 다음과 같은 **알고리즘**으로 문제를 해결할 수 있습니다.
  1. 시작점(start)과 끝점(end)이 첫 번째 원소의 인덱스(0)를 가리키도록 한다.
  2. 현재 부분 합이 M과 같다면, 카운트한다.
  3. 현재 부분 합이 M보다 작다면, end를 1 증가시킨다.
  4. 현재 부분 합이 M보다 크거나 같다면, start를 1 증가시킨다.
  5. 모든 경우를 확인할 때까지 2번부터 4번까지의 과정을 반복한다.



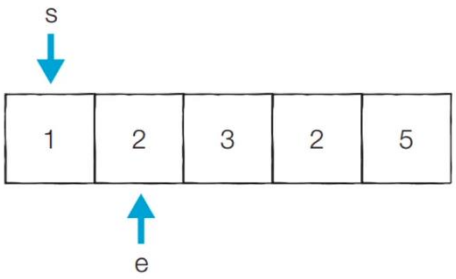
특정한 합을 가지는 부분 연속 수열 찾기: 문제 해결 아이디어

- $M = 5$
- **[초기 단계]** 시작점과 끝점이 첫 번째 원소의 인덱스를 가리키도록 합니다.
  - 현재의 부분합은 1이므로 무시합니다.
  - **현재 카운트**: 0



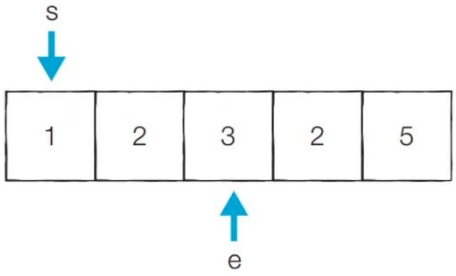
특정한 합을 가지는 부분 연속 수열 찾기: 문제 해결 아이디어

- $M = 5$
- **[Step 1]** 이전 단계에서의 부분합이 1이었기 때문에 end를 1 증가시킵니다.
  - 현재의 부분합은 3이므로 무시합니다.
  - **현재 카운트**: 0



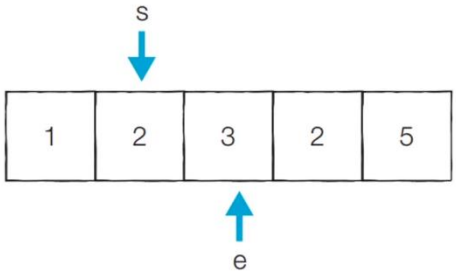
특정한 합을 가지는 부분 연속 수열 찾기: 문제 해결 아이디어

- $M = 5$
- **[Step 2]** 이전 단계에서의 부분합이 3이었기 때문에 end를 1 증가시킵니다.
  - 현재의 부분합은 6이므로 무시합니다.
  - **현재 카운트**: 0



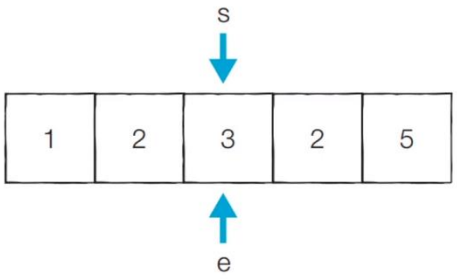
특정한 합을 가지는 부분 연속 수열 찾기: 문제 해결 아이디어

- $M = 5$
- **[Step 3]** 이전 단계에서의 부분합이 6이었기 때문에 start를 1 증가시킵니다.
  - 현재의 부분합은 5이므로 카운트를 증가시킵니다.
  - **현재 카운트**: 1



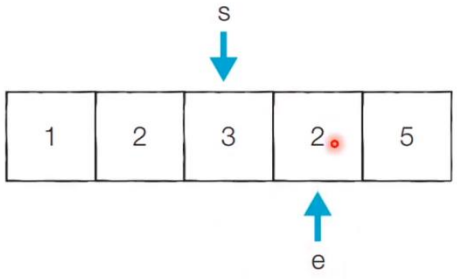
특정한 합을 가지는 부분 연속 수열 찾기: 문제 해결 아이디어

- $M = 5$
- **[Step 4]** 이전 단계에서의 부분합이 5이었기 때문에 start를 1 증가시킵니다.
  - 현재의 부분합은 3이므로 무시합니다.
  - **현재 카운트**: 1



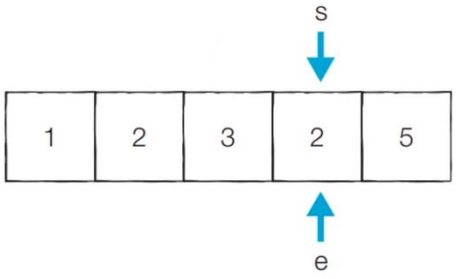
특정한 합을 가지는 부분 연속 수열 찾기: 문제 해결 아이디어

- $M = 5$
- **[Step 5]** 이전 단계에서의 부분합이 3이었기 때문에 end를 1 증가시킵니다.
  - 현재의 부분합은 5이므로 카운트를 증가시킵니다.
  - **현재 카운트**: 2



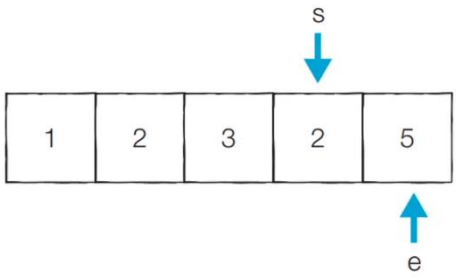
특정한 합을 가지는 부분 연속 수열 찾기: 문제 해결 아이디어

- $M = 5$
- **[Step 6]** 이전 단계에서의 부분합이 5이었기 때문에 start를 1 증가시킵니다.
  - 현재의 부분합은 2이므로 무시합니다.
  - **현재 카운트**: 2



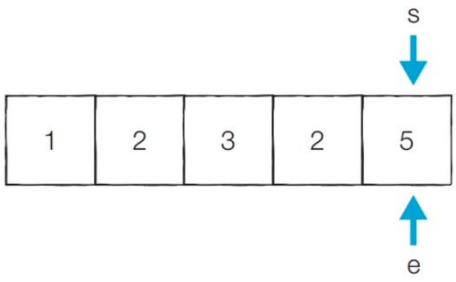
특정한 합을 가지는 부분 연속 수열 찾기: 문제 해결 아이디어

- $M = 5$
- **[Step 7]** 이전 단계에서의 부분합이 2였기 때문에 end를 1 증가시킵니다.
  - 현재의 부분합은 7이므로 무시합니다.
  - **현재 카운트**: 2



특정한 합을 가지는 부분 연속 수열 찾기: 문제 해결 아이디어

- $M = 5$
- **[Step 8]** 이전 단계에서의 부분합이 7이었기 때문에 start를 1 증가시킵니다.
  - 현재의 부분합은 5이므로 카운트를 증가시킵니다.
  - **현재 카운트**: 3



특정한 합을 가지는 부분 연속 수열 찾기: 코드 예시 (Python)

```python
n = 5 # 데이터의 개수 N
m = 5 # 찾고자 하는 부분합 M
data = [1, 2, 3, 2, 5] # 전체 수열

count = 0
interval_sum = 0
end = 0

# start를 차례대로 증가시키며 반복
for start in range(n):
    # end를 가능한 만큼 이동시키기
    while interval_sum < m and end < n:
        interval_sum += data[end]
        end += 1
    # 부분합이 m일 때 카운트 증가
    if interval_sum == m:
        count += 1
    interval_sum -= data[start]

print(count)
```

실행 결과

```
3
```

구간 합 (Interval Sum)

- **구간 합 문제**: 연속적으로 나열된 N개의 수가 있을 때 특정 구간의 모든 수를 합한 값을 계산하는 문제
- 예를 들어 5개의 데이터로 구성된 수열 {10, 20, 30, 40, 50}이 있다고 가정합시다.
  - 두 번째 수부터 네 번째 수까지의 합은 20 + 30 + 40 = 90입니다.

구간 합 빠르게 계산하기: 문제 설명

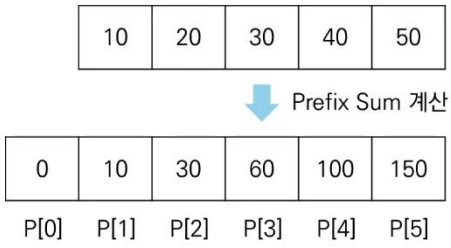
- N개의 정수로 구성된 수열이 있습니다.
- M개의 쿼리(Query) 정보가 주어집니다.
  - 각 쿼리는 $Left$와 $Right$으로 구성됩니다.
  - 각 쿼리에 대하여 $[Left, Right]$ 구간에 포함된 데이터들의 합을 출력해야 합니다.
- 수행 시간 제한은 $\boldsymbol{O(N+M)}$입니다.

구간 합 (Interval Sum)

- **구간 합 문제**: 연속적으로 나열된 N개의 수가 있을 때 특정 구간의 모든 수를 합한 값을 계산하는 문제
- 예를 들어 5개의 데이터로 구성된 수열 {10, 20, 30, 40, 50}이 있다고 가정합시다.
  - 두 번째 수부터 네 번째 수까지의 합은 20 + 30 + 40 = 90입니다.

구간 합 빠르게 계산하기: 문제 해결 아이디어

- **접두사 합(Prefix Sum)**: 배열의 맨 앞부터 특정 위치까지의 합을 미리 구해 놓은 것
- 접두사 합을 활용한 **알고리즘**은 다음과 같습니다.
  - N개의 수 위치 각각에 대하여 접두사 합을 계산하여 P에 저장합니다.
  - 매 M개의 쿼리 정보를 확인할 때 구간 합은 $P[Right] - P[Left - 1]$입니다.



1) $Left$ = 1, $Right$ = 3 ➡ P[3] − P[0] = **60**

2) $Left$ = 2, $Right$ = 5 ➡ P[5] − P[1] = **140**

...

M) $Left$ = 3, $Right$ = 4 ➡ P[4] − P[2] = **70**

구간 합 빠르게 계산하기: 코드 예시 (Python)

```python
# 데이터의 개수 N과 데이터 입력받기
n = 5
data = [10, 20, 30, 40, 50]

# 접두사 합(Prefix Sum) 배열 계산
sum_value = 0
prefix_sum = [0]
for i in data:
    sum_value += i
    prefix_sum.append(sum_value)

# 구간 합 계산(세 번째 수부터 네 번째 수까지)
left = 3
right = 4
print(prefix_sum[right] - prefix_sum[left - 1])
```

실행 결과

```
70
```