



부산대학교

PUSAN NATIONAL UNIVERSITY

Algorithm Analysis

Measuring Cost

Timing Runs and Counting Operations

DS3600082-140

자료구조

2026학년도 2학기

오용경

데이터사이언스전공

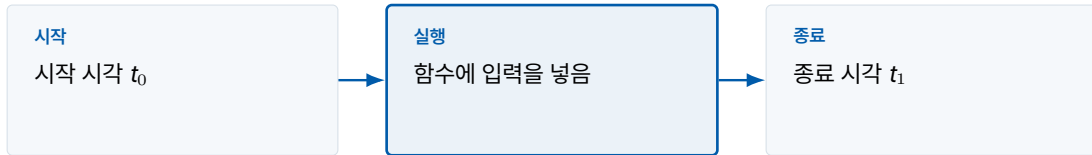
yongkyung.oh@pusan.ac.kr

Timing a Run

같은 입력과 측정 조건에서 실행 시간을 비교한다

실행 시간을 재어 비교한다

실행 시간(running time)은 프로그램이 실행을 시작해 끝낼 때까지 걸린 시간이다



$$\text{실행 시간} = t_1 - t_0 \quad (\text{초})$$

실측(benchmark)은 코드를 실행해 걸린 시간을 재는 방법이며, 같은 결과를 내는 두 구현도 실행 시간은 다를 수 있다.

같은 합을 구하는 세 함수

세 함수 모두 $1 + 2 + \dots + n$ 을 정수로 반환한다 ($n \geq 0$)

반복 합산

```
1 def sum_loop(n):
2     total = 0
3     for i in range(1, n + 1):
4         total = total + i
5     return total
```

대입을 하나 더 한 합산

```
1 def sum_extra(n):
2     total = 0
3     for i in range(1, n + 1):
4         value = i
5         total = total + value
6     return total
```

합 공식

```
1 def sum_formula(n):
2     return n * (n + 1) // 2
```

$n = 10$ 이면 세 함수 모두 55를 반환한다. $//$ 2는 정수 결과를 유지한다.

세 함수는 같은 값을 구하지만 반복 안에서 하는 일이 다르다

timeit으로 여러 번 실행한다

한 번의 짧은 실행보다 여러 호출을 묶어 재면 측정하기 쉽다

```
1 from timeit import Timer
2 from statistics import median
3
4 timer = Timer(lambda: sum_loop(10000))
5 times = timer.repeat(repeat=5, number=10)
6 us = median(times) / 10 * 1_000_000
7 print(us)
```

코드	뜻
number=10	한 묶음에 10회 호출
repeat=5	같은 측정을 5묶음 반복
/ 10	호출당 시간으로 환산
* 1_000_000	초를 μ s로

묶음 시간을 호출 수로 나누고 단위를 바꾸면 호출당 μ s가 된다

같은 코드를 다시 실행해도 측정값은 달라진다

합산 함수에 같은 $n = 10,000$ 을 넣고 다섯 번 측정했다

측정 묶음	전체 시간(ms)	호출당 시간(μ s)
1	2.970	296.990
2	2.824	282.390
3	2.685	268.530
4	3.257	325.670
5	2.863	286.310

함수를 10회 호출한 시간을 한 묶음으로 하여 다섯 묶음을 측정했다.

다른 프로그램과 운영체제의 영향으로 측정값이 달라지므로 다섯 묶음의 중앙값을 사용한다

입력을 키워 세 함수를 측정한다

<i>n</i>	반복 합산	대입 추가	합 공식
1,000	34.535	32.437	0.131
10,000	286.310	392.160	0.133
100,000	3946.950	4062.650	0.153

호출당 시간을 마이크로초로 환산한 뒤 다섯 묶음의 중앙값을 사용했다.

이 입력 범위에서는 합 공식이 가장 빨랐지만 합산 함수끼리의 작은 차이는 측정 변동과 구분해 해석해야 한다

준비 시간과 연산 시간을 분리한다

목록을 만드는 시간까지 재면 원소 하나를 읽는 비용과 섞인다

```
1 from timeit import Timer
2
3 timer = Timer(
4     "a[500]", setup="a = list(range(10000))"
5 )
6 times = timer.repeat(repeat=5, number=1000)
```

부분	하는 일	측정 시간에 포함
setup 측정문	길이 10,000인 목록 생성 a[500]을 1,000번 읽기	아니오 예

setup에 둔 코드는 묶음마다 실행되지만 측정 시간에는 들어가지 않는다

Source: Miller and Ranum (2013), Algorithm Analysis.

먼저 결과가 같은지 확인한다

틀린 답을 빨리 내는 코드는 올바른 구현과 성능을 비교할 대상이 아니다

```
1 for n in (0, 1, 10, 1000):
2     expected = n * (n + 1) // 2
3     assert sum_loop(n) == expected
4     assert sum_extra(n) == expected
5     assert sum_formula(n) == expected
```

n	expected
0	0 (빈 입력)
10	55
1000	500500

assert는 조건이 거짓이면 오류를 낸다. 이 검사는 시간 측정 전에 실행한다.

Caution 몇 개 입력을 통과했다고 모든 입력에서 옳다는 증명은 아니다

비교할 조건을 맞춘다

두 측정 사이에 코드 말고 다른 것이 달라지면 시간 차이를 해석할 수 없다

비교	확인할 조건
노트북의 함수 A와 서버의 함수 B A는 목록을 만들고 읽음, B는 읽기만 함 A는 원소 100개, B는 원소 10,000개	같은 컴퓨터에서 두 함수를 다시 측정한다. 생성 시간을 포함할지 두 쪽에 똑같이 정한다. 같은 크기의 입력에서 먼저 비교한다.

컴퓨터, 측정 범위, 입력 크기를 맞춘 뒤에야 코드끼리 비교한다

Checkpoint

Timing a Run

- 10회 호출에 걸린 초를 호출당 시간으로 바꾸려면 어떻게 계산하는가
- 목록 생성 시간을 제외하고 조회만 재려면 코드를 어디에 두는가
- 같은 코드를 여러 번 측정하는 이유는 무엇인가

Counting Operations

입력 크기와 횟수를 셀 연산을 정하고 연산 수를 식으로 적는다

문제별 입력 크기

입력 크기(input size)는 처리할 자료의 양을 나타내는 기준이다

문제	입력 크기
목록의 모든 값 더하기	원소 수 $n = \text{len}(a)$
문자열의 각 문자 확인	문자 수 $n = \text{len}(\text{text})$
두 목록의 모든 원소 쌍 비교	첫 목록의 크기 m , 둘째 목록의 크기 n
1부터 주어진 정수까지 더하기	이 예제에서는 더하는 정수의 개수 n

정수 입력의 저장 길이를 셀 때는 자릿수나 비트 수를 기준으로 삼기도 한다.

횟수를 셀 연산을 먼저 정한다

연산 수(operation count)는 지정한 연산이 실행되는 횟수다

```
def sum_loop(n):  
    total = 0  
    for i in range(1, n + 1):  
        total = total + i  
    return total
```

여기서 세는 것

- total에 값을 대입하는 연산
- 처음에 한 번, 반복문 안에서 n 번
- 입력 크기 n 에서의 연산 수를 $T(n)$ 으로 적는다: $T(n) = 1 + n$ (반복 제어와 반환은 제외)

한 줄씩 연산 수를 적는다

$n = 4$ 일 때의 실행을 식 $T(n)$ 과 대응시킨다

대입 위치	$n = 4$	일반적인 n
<code>total = 0</code>	1회	1회
<code>total = total + i</code>	4회	n 회
합계	5회	$n + 1$ 회

$0 \longrightarrow 1 \longrightarrow 3 \longrightarrow 6 \longrightarrow 10$

처음의 0은 초기 대입 결과, 화살표 하나는 반복문 안의 대입 한 번이다.

추가한 대입 한 줄의 실행 횟수

반복문 안의 대입이 하나 늘어나면 n 번의 대입이 추가된다

```
1 def sum_extra(n):  
2     total = 0  
3     for i in range(1, n + 1):  
4         value = i  
5         total = total + value  
6     return total
```

함수	초기화	반복 안의 대입	합계
sum_loop	1	n	$n + 1$
sum_extra	1	$2n$	$2n + 1$

$n = 40$ 이면 대입이 5회와 9회로 다르지만 두 함수의 반환값은 같다

연산 수는 계산 모형에 따라 달라진다

계산 모형(model of computation)은 어떤 연산을 한 번으로 셀지 정한 약속이다

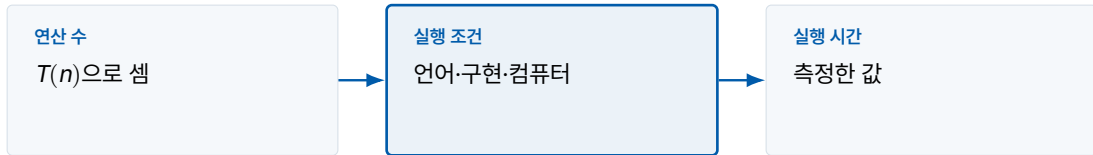
```
1 def sum_formula(n):  
2     return n * (n + 1) // 2
```

$$n + 1 : 1\text{회} \quad n(n + 1) : 1\text{회} \quad n(n + 1) // 2 : 1\text{회}$$

함수 호출과 반환은 세지 않았다. 같은 모형에서 `sum_loop`의 산술 `total + i`는 n 회다. Python의 큰 정수는 자릿수에 따라 산술 연산 자체의 시간이 달라질 수 있다.

합 공식은 n 과 관계없이 산술 연산 3회로 끝난다

연산 수와 실행 시간은 다른 양이다



연산 수는 정한 계산 모형에서 코드의 구조를 설명하고, 실행 시간은 같은 입력과 환경에서 실제 실행을 관찰한 값이다.

sum_loop와 sum_extra의 대입 수는 계산할 수 있지만, 실행 시간의 비율은 반복 측정으로 확인해야 한다

두 변수를 갱신하는 대입 횟수

반복 제어를 제외하고 명시적인 대입만 세면 반복 한 번마다 두 번이 추가된다

```
1 total = 0
2 count = 0
3 for x in a:
4     total = total + x
5     count = count + 1
```

위치	대입 수 ($n = \text{len}(a)$)
반복문 밖	2
반복문 안	한 바퀴마다 2
합계	$T(n) = 2 + 2n$

$n = 30$ 이면 대입 8회, 빈 목록이면 반복문 밖의 2회만 실행된다

Checkpoint

Counting Operations

- $T(n)$ 을 적기 전에 입력 크기와 어떤 연산을 셀지 정해야 하는 이유는 무엇인가
- 반복문 안에 대입을 하나 추가하면 연산 수는 얼마나 늘어나는가
- 연산 수가 같아도 실행 시간이 다를 수 있는가

Loops and Input Size

반복 범위에 따라 달라지는 연산 수를 계산한다

한 번의 반복은 원소 하나를 처리한다

```
count = 0
for x in [4, 7, 2, 9, 1]:
    count += 1
```

반복	x	count
1	4	1
2	7	2
...	...	...
5	1	5

입력 크기 $n = 5$ 에서 5번 실행되므로 일반적으로 $T(n) = n$ 이다

연속된 반복문은 각각 센 뒤 더한다

```
for x in a:
    count += 1
for y in b:
    count += 1
```

실행 블록

연산 수 ($m = \text{len}(a), n = \text{len}(b)$)

첫 반복문

m

둘째 반복문

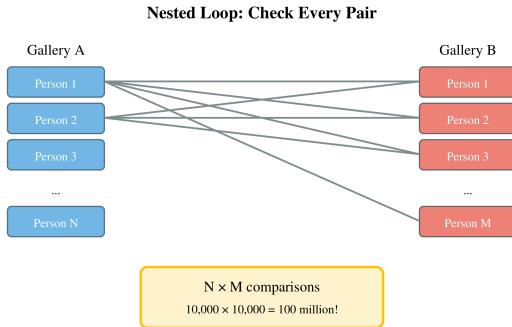
n

전체

$m + n$

두 반복문이 나란히 있으면 $m + n$ 번이며 mn 번이 아니다

두 집단의 모든 쌍을 비교한다



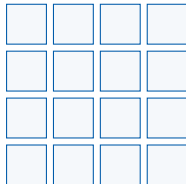
한 사람당 n 명을 비교하는 작업을 m 명에 대해 반복하므로 비교는 mn 번이다.

Source: UBC DSCI 221 (n.d.), Nested Loop: Check Every Pair.

독립적인 중첩 반복은 곱한다

안쪽 반복이 바깥 반복의 매 단계에서 처음부터 다시 시작한다

```
1 count = 0
2 for i in range(n):
3     for j in range(n):
4         count += 1
```



$$4 \times 4 = 16\text{회}$$

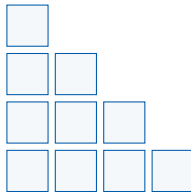
$$T(n) = \underbrace{n + \cdots + n}_{n\text{개}} = n^2$$

안쪽 반복의 길이가 달라지면 더한다

안쪽 범위가 `range(i)`이면 모든 줄의 길이가 같지 않다

```
1 count = 0
2 for i in range(n):
3     for j in range(i):
4         count += 1
```

$$n = 5 : 0 + 1 + 2 + 3 + 4 = 10$$



$$T(n) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

Source: Miller and Ranum (2013), Algorithm Analysis.

두 입력을 하나의 n 으로 합치지 않는다

길이가 다른 두 목록이면 두 크기를 따로 둔다

```
1 a = [2, 4, 6]
2 b = [1, 3, 5, 7]
3 count = 0
4 for x in a:
5     for y in b:
6         count += 1
7 print(count) # 12
```

첫 목록의 크기	둘째 목록의 크기	안쪽 문장의 실행 횟수
$m = \text{len}(a) = 3$	$n = \text{len}(b) = 4$	$T(m, n) = mn = 12$

첫 목록만 두 배로 늘리면 24회, 두 목록을 각각 두 배로 늘리면 48회다

반복문의 모양보다 실행 범위를 읽는다

두 겹으로 보인다는 이유만으로 항상 n^2 이라고 결론 내릴 수 없다

반복 범위	연산 수	$n = 5$
<code>range(0, n, 2)</code>	$\lceil n/2 \rceil$	3회
바깥 <code>range(n)</code> , 안쪽 <code>range(3)</code>	$3n$	15회
바깥 <code>range(n)</code> , 안쪽 <code>range(i)</code>	$n(n-1)/2$	10회

안쪽 범위가 상수인지, n 인지, 바깥 인덱스 i 인지 먼저 확인한다

Checkpoint

Loops and Input Size

- 연속된 두 반복문과 중첩된 두 반복문의 연산 수는 어떻게 다른가
- 안쪽 범위가 i 에 따라 바뀌는 반복문에서 n 곱하기 n 을 쓸 수 없는 이유는 무엇인가
- 두 목록의 길이가 각각 m 과 n 일 때 모든 쌍은 몇 개인가

Python Operations

같은 결과를 내는 Python 코드의 실행 시간을 비교한다

목록을 늘리는 두 방법

두 함수는 같은 목록(Python list)을 반환하지만 반복 안에서 하는 작업이 다르다

```
1 def build_append(n):  
2     a = []  
3     for i in range(n):  
4         a.append(i)  
5     return a
```

```
1 def build_concat(n):  
2     a = []  
3     for i in range(n):  
4         a = a + [i]  
5     return a
```

append는 기존 목록을 바꾸고 $a + [i]$ 는 반복마다 새 목록을 만든다

목록 생성 함수 전체를 측정한다

n	append	+
1,000	36.367	1114.133
4,000	178.933	20615.433
16,000	742.000	326253.900

한 묶음에서 함수를 3회 호출하여 5묶음의 호출당 중앙값(μs)을 사용했다. n 이 4배가 될 때 append는 약 4배에서 5배, +는 약 16배에서 19배로 늘었다.

표의 값은 목록 생성부터 반환까지 걸린 시간이며, append 한 번의 시간이 아니다

삭제 연산의 측정 조건을 맞춘다

삭제한 뒤 끝에 다시 넣어 매 반복의 목록 길이를 유지한다

```
1 end = Timer("a.append(a.pop())",
2             setup="a = list(range(n))", globals={"n": 10000})
3 front = Timer("a.append(a.pop(0))",
4               setup="a = list(range(n))", globals={"n": 10000})
5 end_times = end.repeat(repeat=5, number=1000)
6 front_times = front.repeat(repeat=5, number=1000)
```

각 묶음 시작 전에 목록을 새로 만들고 pop과 append 한 쌍만 측정한다.

두 측정문의 차이는 pop()과 pop(0) 하나뿐이다

앞에서 삭제할 때와 끝에서 삭제할 때

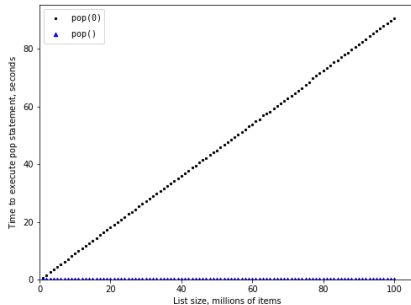
목록 길이만 바꾸어 동일한 측정을 반복했다

n	끝에서 삭제·복구	앞에서 삭제·복구
1,000	0.062	0.123
10,000	0.043	28.049
100,000	0.049	305.687

삭제·복구를 한 묶음에서 1,000쌍 실행하여 5묶음의 중앙값(μs)을 사용했다.

측정 범위가 같은데 앞에서 삭제하는 쪽의 시간만 목록 길이에 따라 크게 늘었다

더 넓은 범위에서 관찰한 삭제 시간



앞의 표와 컴퓨터·입력 크기·측정 방법이 다른 별도 실험이므로 시간 수치를 직접 비교하지 않는다.

Source: Miller and Ranum (2013), Lists, Figure 3.

같은 값이 들어 있는지 확인한다

목록의 원소와 dict의 키에 같은 정수를 넣고 없는 값 -1을 찾는다

```
1 n = 10000
2 values = list(range(n))
3 keys = dict.fromkeys(range(n))
4 list_timer = Timer("-1 in a", globals={"a": values})
5 dict_timer = Timer("-1 in a", globals={"a": keys})
6 list_times = list_timer.repeat(repeat=5, number=100)
7 dict_times = dict_timer.repeat(repeat=5, number=100)
```

두 식은 모두 False를 반환하며 목록과 dict를 만드는 시간은 측정에서 제외한다.

찾는 값과 원소 수가 같으므로 두 측정은 자료구조만 다르다

list와 dict의 in을 측정한다

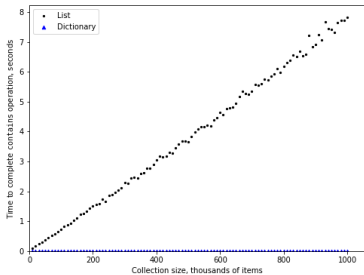
n	list	dict
1,000	7.563	0.029
10,000	89.548	0.030
100,000	983.916	0.033

없는 정수를 한 묶음에서 100회 조회하여 5묶음의 호출당 중앙값(μ s)을 사용했다.

이 입력에서는 목록 조회 시간이 입력 크기와 함께 늘었고, dict의 작은 차이는 측정 변동과 구분하기 어렵다

찾는 입력을 바꾸면 실험도 달라진다

임의의 정수를 찾으면 list의 시간은 원소 수에 비례해 늘고 dict의 시간은 거의 일정하다



앞의 표는 항상 없는 값을 찾았으므로 어떤 값을 찾았는지까지 확인해야 결과를 해석할 수 있다.

Source: Miller and Ranum (2013), Dictionaries, Figure 4.

Checkpoint

Python Operations

- 목록 생성 함수의 시간을 append 한 번의 시간으로 읽으면 안 되는 이유는 무엇인가
- 삭제 연산을 반복해서 쟈일 때 목록 길이를 유지한 이유는 무엇인가
- list와 dict의 조회 시간을 비교할 때 찾을 값을 똑같이 정해야 하는 이유는 무엇인가

How Do We Describe Growth Without Fixing a Machine?

Next Lecture: Asymptotic Notation

- 입력이 커질 때 함수가 늘어나는 증가율
- Big-O로 나타내는 상한
- 최악의 경우와 공간 복잡도