논문 18-43-08-08 The Journal of Korean Institute of Communications and Information Sciences '18-08 Vol.43 No.08
https://doi.org/10.7840/kics.2018.43.8.1282

카운터 형식을 가진 파이썬 프로그램의 고속 구현 방법과 응용

김 원 태◆, 박 호 중*, 염 용 진**, 강 주 성°

High-Speed Implementation and Applications of Python Programs with Counter Type

Wontae Kim◆, Hojoong Park*, Yongjin Yeom**, Ju-Sung Kang°

요 약

본 논문은 카운터(Counter) 형식이 활용된 파이썬(Python) 프로그램에 대한 고속화 방법으로 함수의 표현 방법을 효율적으로 개선하는 방법과 리스트(list) 형식으로 변경 구현하는 방법, 두 가지를 제시한다. 각 방법에 따라 고속화 결과를 정량적으로 제시한다. 나아가 분석한 결과를 활용하여 기존의 고속화 연구의 속도 향상 결과에 대해 함수 단위로 분석한다. 이에 더해 카운터 형식과 리스트 형식 각각을 활용하여 히스토그램 생성 프로그램을 구현한 후, 구동 속도를 비교해 제안하는 고속화 방법의 적합성을 확인한다. 논문의 주요 결과로 카운터 형식의 갱신 함수는 표현 방법을 개선해 5배, 리스트 형식으로 구현하여 30배, 정렬 함수는 리스트 형식으로 구현하여 2배, 최빈값 함수는 리스트로 구현하여 1.5배의 속도 향상이 있다. 이에 더해 리스트 형식의 경우 C와의 속도 비교를 통해 추가 고속화 가능성을 보인다. 본 논문의 분석은 파이썬 프로그램의 데이터 형식 선정의 근거가 될 수 있으며, 카운터 형식이 사용된 프로그램의 고속화에 사용될 수 있을 것으로 기대한다.

Key Words : High-Speed Implement, Python, Data Type, Counter Type, list Type, Code Improvement

ABSTRACT

We propose two ways that are efficiently changing the expression of the Counter type functions and replacing the type to list type and show quantitatively the results of the speed-up according to each function. Using this results, we analyze an existing speed-up research that has 700 times speed improvement result. We confirmed the adequacy of the proposed high-speed implementation method by comparing running times of generating histogram program using Counter type or list type. Our main results are that the update functions of Counter type are 5 times improved the speed by changing the expression, 30 times by changing to list type, the sort function of Counter is 1.5 times by changing to list type. In addition, we present the comparison results of run-times of list type functions in C and Python, and show the possibility for further speed-up. The analysis can be used as a basis for selecting data type and it can be used to speed up Python programs that use counter type.

※ 본 연구는 2017년도 정부(과학기술정보통신부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임 (No.2014-6-00908, 난수발생기 및 임베디드 기기 안전성 연구)

◆ First Author : (ORCID:0000-0002-3817-2452)Kookmin University Department of Financial Information Security, kwt123@kookmin.ac.kr, 학생회원

° Corresponding Author : (ORCID:0000-0002-0846-389X)Kookmin University Department of Information Security, Cryptology, and Mathematics, jskang@kookmin.ac.kr, 정회원

* (ORCID:0000-0003-1179-876X)Kookmin University Department of Financial Information Security, ruokay@kookmin.ac.kr, 학생회원

** (ORCID:0000-0002-8240-8661)Kookmin University Department of Information Security, Cryptology, and Mathematics, salt@kookmin.ac.kr, 종신회원

논문번호 : 201806-D-145-RN, Received April 25, 2018; Revised June 29, 2018; Accepted July 16, 2018

Ⅰ. 서 론

1.1 연구 배경

파이썬(Python)은 프로그램 언어 중 4번째로 많이 사용되는 언어[2]로, 코드의 직관적 이해가 쉽도록 만들어져 가독성이 좋으며 상대적으로 다른 프로그램 언어들보다 풍부한 라이브러리를 제공하여 개발의 편의를 제공[1]하는 특징이 있다. 하지만 구현의 편의성을 위하여 다양한 기능을 갖춘 파이썬 함수들을 무분별하게 사용할 경우 필요 이상의 연산이 소요되어 속도 지연의 원인이 될 수 있다.

한편, 구현에 있어 데이터 형식은 그 구동시간에 직접적인 영향을 줄 수 있다. 데이터 형식이란 다양한 의미로 해석되고 있지만 여기서는 정수, 수열, 문자 등에 대한 값의 공간과 그 값을 사용하는 함수에 대해 정의한 분류를 의미한다. 파이썬은 기본 데이터 형식으로 사전(dict), 리스트(list), 튜플(tuple), 셋(set) 등을 제공하고 있으며, 기본 데이터 형식을 활용해 보다 유용한 기능을 제공하는 컬렉션(collections)과 같은 형식을 제공하고 있다. 파이썬에서 제공하는 데이터 형식의 계층 구조[3,4,5]는 그림 1과 같으며, 형식에 따른 기능[4,5]은 표 1에 요약 제시하였다.

표 1에서 굵게 표시한 카운터(Counter)는 컬렉션에 포함되는 데이터 형식으로, 사전의 하위 클래스다. 카운터 형식은 입력 값인 키(key)와 키의 입력 횟수인 카운트(count)를 편리하게 기록하는 것을 목적으로 구성 되었다. 카운터는 입력 값들에 대하여 입력 순서를 고려하지 않으며, 키를 사전의 키로, 카운트는 사전의 값(value)로 기록한다. Python 3.2.x 이상의 버전에서 카운터가 제공하는 기능[5]은 사전에서 제공하는 모든 기능을 포함하고 키와 그에 대응하는 카운트를 기록하고 관리하는데 편리한 기능을 제공하기 위해 elements, most_common, subtract 세 가지의 함수를 추가 제공하며 그 기능은 표 2와 같아 통계적 프로그램 구현에 유용할 것으로 보인다. 카운터 형식이 활용된 프로그램으로는 머신러닝(machine learning)에 주로 사용되는 프로그램인 텐서플로우(TensorFlow)[6]와 NIST(National Institute of Standards and Technology)가 제공하는 엔트로피 측정 도구[7] 등이 있다.

1.2 연구 결과

본 논문에서 제안하는 고속화 방법은 총 두 가지로 카운터 형식을 유지하고 함수의 표현을 효율적으로 변경하는 방법과 카운터 형식을 리스트 형식으로 변경 구현하는 방법이 있다. 각 함수에 따른 속도 향상 결과는 그림 2와 같으며 보다 자세한 결과는 결론의 표 16에 제시한다.

1.3 논문 구성

본 논문의 구성은 다음과 같다. Ⅱ 장에서는 관련 연구로 NIST SP 800-90B[8]의 엔트로피 평가 도구에 대한 고속화 연구 결과[9]를 요약한다. Ⅲ 장에서는 카

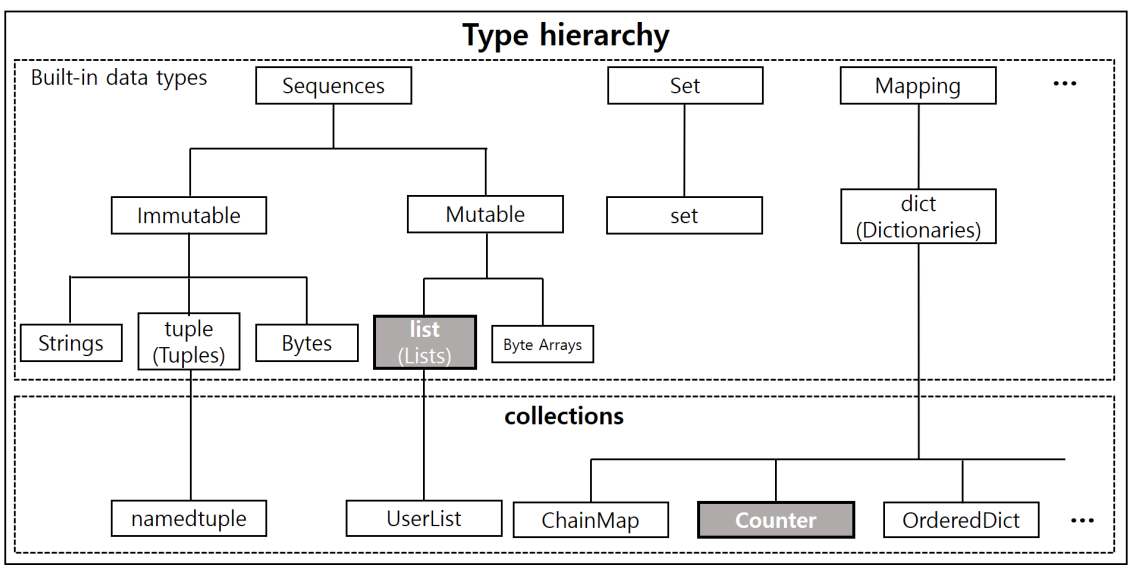

그림 1. 파이썬의 데이터 타입에 따른 계층 구조
Fig. 1. The hierarchy of data types in Python

표 1. 데이터 형식에 따른 기능
Table 1. Role of data types

Data type	dict	list	tuple	set
Role	A mapping object maps hashable values to arbitrary objects. Mappings are mutable objects.	The operations in the following table are supported by most sequence types, both mutable and immutable.	Tuples are immutable sequences, typically used to store collections of heterogeneous data.	A set object is an unordered collection of distinct hashable objects.
Data type	collections	collections.namedtuple	collections.OrderedDict	collections.Counter
Role	A collections implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, dict, list, set, and tuple. It has subtypes such that OrderedDict, Counter etc.	Returns a new tuple subclass named typename. The new subclass is used to create tuple-like objects that have fields accessible by attribute lookup as well as being indexable and iterable.	Ordered dictionaries are just like regular dictionaries but they remember the order that items were inserted. When iterating over an ordered dictionary, the items are returned in the order their keys were first added.	A Counter is a dict subclass for counting hashable objects. It is an unordered collection where elements are stored as dictionary keys and their counts are stored as dictionary values.

표 2. 카운터 타입 제공 함수와 그 기능
Table 2. The methods of Counter type

Method	*elements*	*most_common*	*subtract*
Role	Return an iterator over elements repeating each as many times as its count. Elements are returned in arbitrary order.	Return a list of the n most common elements and their counts from the most common to the least.	Elements are subtracted from an iterable or from another mapping (or counter). Like dict.update() but subtracts counts instead of replacing them.

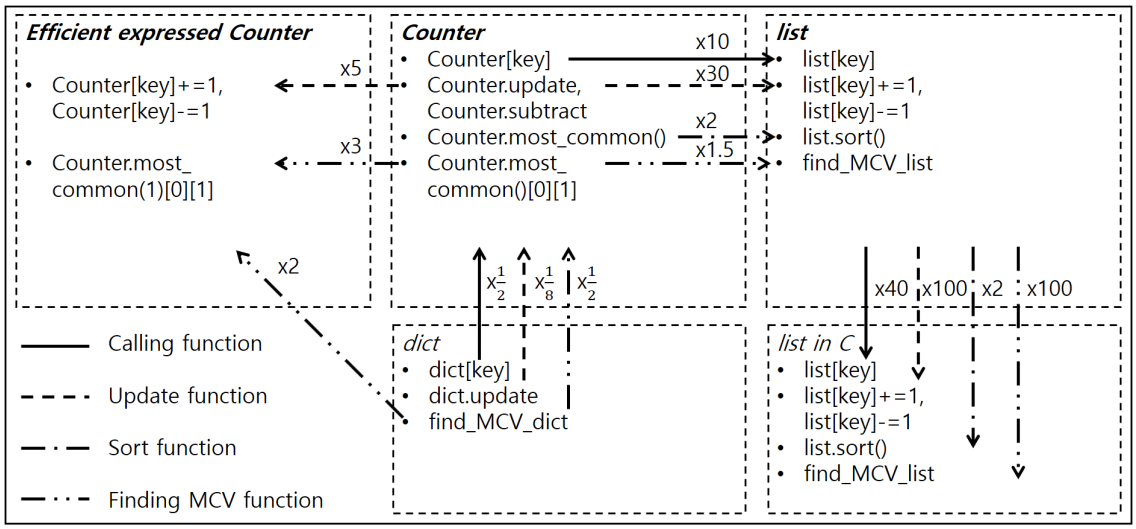

그림 2. 고속화 결과 요약
Fig. 2. Summary of Speed-ups

운터 형식이 사용된 파이썬 프로그램에 대한 고속화 방법 두 가지를 제시하고, 각 방법에 대해 적용 가능 조건, 고속화 원인 그리고 속도 향상의 정량적 결과를 제시한다. 추가로 카운터 형식과 상위 클래스인 사전 형식의 속도를 비교하며, 리스트 형식의 경우 C 와의 속도 비교를 통해 추가 고속화 가능성 또한 제시한다. Ⅳ 장에서는 논문의 분석 결과를 활용하여 기존 고속화 연구인 NIST의 엔트로피 측정 도구의 multiMCW에 대한 고속화 결과[10]에 대한 속도 향상의 원인을 함수 단위로 분석한다. 이에 더해 카운터 형식과 리스트 형식 각각을 활용해 구현한 히스토그램 생성 프로그램에 대한 속도 측정 결과를 제시하여 제안하는 고속화 방법의 적합성을 보인다. 마지막으로 Ⅴ 장에서는 논문의 결론을 맺고 해당 연구가 가지는 기대효과에 대해 언급한다.

Ⅱ. 관련 연구

2.1 NIST의 엔트로피 평가 도구의 multiMCW에 대한 고속화 연구[10]

기존 multiMCW에 대한 고속 구현 연구는 본 논문에서 제안하는 고속화 방법의 모티브가 된 연구로, NIST의 엔트로피 평가 도구를 구성하는 10 개의 엔트로피 측정 함수 중 가장 느린 함수인 multiMCW에 대한 고속 구현 방법을 제안하고 그 타당성을 실험적으로 제시하였다. 고속화 방법은 크게 세 가지로 다음과 같다.

1) 데이터 형식을 카운터에서 리스트로 변경 구현
2) NIST의 코드에서 함수 중복 사용 등 비효율적으로 구현된 부분 개선
3) C를 이용한 추가 고속화

제안하는 방법의 고속화 효과를 확인하기 위하여 multiMCW에 대해 기존 NIST 코드, C로 단순 변환 코드, 개선한 파이썬 코드, 개선한 C 코드, 총 네 가지 코드의 구동시간을 측정한 결과를 비교했고, 제안하는 방법을 적용한 C 코드는 NIST 코드 대비 700 배의 속도 향상이 확인되어 제안하는 방식이 고속화에 충분한 효과를 갖는 것을 보여준다. 그러나 기존의 연구에서는 속도향상의 원인 분석이 명확히 제시되지 못했다는 아쉬움을 갖는다. 본 연구에서는 카운터와 리스트 형식에 대해 함수 단위로 고속화 효과를 정량적으로 측정하였으며, 이 분석 결과를 기반으로 기존 연구의 속도향상에 대한 원인을 보다 상세히 분석한다.

2.2 C++를 이용한 NIST의 엔트로피 평가 도구에 대한 고속화 연구[9]

SP 800-90B 엔트로피 평가 도구의 고속 및 메모리 경량 구현 연구는 평가 도구를 구성하는 10 개의 엔트로피 측정 함수에 대한 기존 20 분의 구동시간 문제와 5.5 GB의 메모리 사용 문제[11]를 해결하기 위해 메모리 효율적 사용 및 고속 구현 방법을 제시한다. 고속화를 위하여 먼저 10개의 엔트로피 측정함수를 C++로 단순 변환 구현하였으며, 변환 후에도 구동 속도가 느린 3 개의 측정 함수인 multiMCW, multiMMC, LZ78Y에 대해 고속화 방법을 제시하였다. 여기서 기존 multiMCW의 경우 기존 multiMCW 고속 구현 연구[9]와 동일한 방법이 사용되었다. multiMMC와 LZ78Y에 대해서는 데이터 기록 방식을 트리(tree) 구조[12]로 개선하여 기존 메모리 문제를 해결하였으며, 추가 고속화 효과를 얻었다. 결과적으로 제안된 C++ 코드의 구동시간은 88 초, 사용 메모리는 440 MB로 기존 NIST의 코드 대비 약 14 배의 속도가 향상되었으며, 메모리 사용량은 1/13 로 감소하였다. 연구의 주요 결과인 multiMMC와 LZ78Y에 대한 개선 방법 또한 카운터 형식을 리스트 형식으로 변경 구현한 결과로 제안하는 고속화 방법이 사용 메모리 개선에도 활용될 수 있을 것으로 기대된다. 그러나 이에 대한 연구는 본 연구의 범위를 벗어난다.

Ⅲ. 카운터 형식이 활용되는 프로그램에 대한 고속화 방법

3.1 카운터 형식의 효율적 표현 방법

제안하는 방법은 카운터 형식의 갱신 함수(Counter.update, Counter.subtract), 정렬 함수(Counter.most_common)가 활용되는 코드에 적용 가능하다. 같은 데이터 형식에 동일한 기능을 수행하더라도 함수의 표현방법에 따라 구동 속도에 큰 차이를 가질 수 있다. 카운터 함수의 효율적 표현 방법을 확

표 3. 카운터 함수의 구동시간(초) 비교
Table 3. Run-time(seconds) comparison of Counter functions

Function	*Counter.update[(key)], Counter.subtract[(key)]*		*Counter[key]+=1, Counter[key]-=1*
Run-time	1.60		0.28
Function	*Counter.most_common()[0][1]*	*Counter.most_common(1)[0][1]*	*find_MCV_counter*
Run-time	47.26	14.02	26.13

인하기 위하여 호출, 갱신, 정렬, 그리고 자체 정의한 최빈값 도출 함수를 대상으로, 각 함수의 구동시간을 측정 비교하였다. 추가로 상위 클래스인 dict 형식의 함수들과 비교하여 카운터 형식 함수의 효율성을 평가하도록 하겠다. 카운터 형식의 함수에 대한 표현 방법에 따른 구동시간 비교 결과는 표 3과 같다. 제시한 실험 결과는 카운터에 256 개의 키와 카운트 쌍이 기록되었을 때, 100 만회 반복 구동시간을 제시한다.

3.1.1. 카운터 형식 호출 및 갱신 함수의 효율적 표현

키(*key*)에 대응하는 카운트를 출력하는 함수를 호출 함수(calling function)라 할 때, *Counter[key]*로 표현할 수 있으며, 횟수를 더하거나 빼는 함수를 갱신 함수(update function)라 할 때, *Counter.update([key])*와 *Counter.subtract([key])*로 표현할 수 있다. 여기서 *Counter.update*와 *Counter.subtract*는 입력으로 하나 이상의 키를 받아 기록할 수 있는데, 만약 하나의 키를 갱신할 경우 *Counter[key]*+=1 또는 *Counter[key]*-=1과 같이 호출 함수만으로 표현 가능하다. *Counter[key]*+=1 또는 *Counter[key]*-=1로 표현할 경우 기존 갱신 함수에서 수행되던 다양한 비교 연산 들을 피할 수 있게 되고, 그 결과 구동시간을 5 배 단축할 수 있었다. 즉, 하나의 키를 카운터에서 갱신하는 경우 *update* 함수보다 호출 함수를 이용하는 것이 효율적이다.

3.1.2카운터 형식 정렬 함수의 효율적 표현

카운트에 따라 키를 정렬하는 함수를 정렬 함수(sort function)라 할 때, 카운터 형식의 정렬 함수는 *Counter.most_common(n)*으로 표현된다. *Counter.most_common(n)* 함수는 n의 입력 여부에 따라 내부 구동 함수가 결정되게 되며 이 경우 구동시간에도 차이를 갖게 된다. 만약 n에 아무 입력을 하지 않는다면 Python 기본 함수인 *sorted* 함수를 사용해 카운터의 모든 기록된 키에 대하여 카운트를 기준으로 정렬한 결과를 반환하며, 만약 n이 1 이상의 정수가 입력되면 heapq 클래스의 *_heapq.nlargest* 함수를 사용해 기록된 카운트의 크기를 기준으로 n개의 키와 카운트 쌍을 정렬한 결과를 반환한다. 두 함수의 차이로 n에 정수 값을 입력한 경우가 아무 값도 입력하지 않은 경우보다 3 배까지 더 빠르게 구동하는 것을 확인할 수 있다. 또한 정수 n이 작을수록 구동시간이 더 적게 소요된다.

Pseudo-code of finding MCV function

```
max = 0
index = 0
for i in (1~#data)
   if storage[i] > max
      max = storage[i]
      index = i
return index
```

※ "#data n" is denotes that the recording data number is n

그림 3. 최빈값 함수의 의사코드
Fig. 3. The pseudo-code of finding MCV function

3.1.3 카운터 형식 최빈값 함수의 효율적 표현

기록된 키 중 입력 횟수가 가장 높은 값을 찾는 함수를 최빈값 함수(finding MCV function)라 하자. 이 함수는 기록된 모든 값을 한번 씩 비교하여 가장 큰 값의 순서를 출력하는 방식으로 그림 3의 의사 코드와 같이 구현된다. 여기서 *find_MCV_counter*는 카운터의 정렬 함수를 사용해 *Counter.most_common(1)[0][1]*로도 표현할 수 있는데, 이 경우 *find_MCV_counter*보다 약 3 배 더 빠르다. 이 결과는 카운터의 정렬 함수인 *most_common* 함수는 최적화가 잘 되어 있으며, 카운터 형식으로 기록된 데이터에 대해서는 최빈값을 도출할 때 *Counter.most_common(1)[0][1]*로 표현하는 것이 효율적이라는 것을 의미한다. 여기서 주의할 점은 *Counter()most_common()[0][1]*과 같이 *most_common*에 아무 값도 입력하지 않을 경우 내부에서 구동되는 함수가 *_heapq.nlargest*에서 *sorted*가 사용 되, 오히려 *find_MCV_counter* 보다 느려지게 된다는 것이다.

3.1.4 카운터 형식과 사전 형식의 성능 비교

추가로 카운터 형식의 성능 확인을 위하여 사전 형식과 속도를 비교했으며, 그 결과는 표 4와 같다. 카운터 형식의 호출 함수와 갱신 함수는 상위 클래스인 사전 보다 느렸다. 최빈값 함수인 *find_MCV_counter*, *find_MCV_dict* 함수를 각각 카운터, 사전 형식을 이용해 그림 3 함수를 구현한 함수라 할 때, 호출 함수의 구동시간 차이로 *find_MCV_counter*는 *find_MCV_dict*보다 느린 것을 확인할 수 있다. 최빈값을 도출하는 카운터 형식의 *Counter.most_common(1)[0][1]*함수는 *find_MCV_dict* 보다도 약 2 배 더 빠른 것을 확인할 수 있다.

표 4. 사전 함수와 카운터 함수 구동시간(초) 비교
Table 4. Run-time(seconds) comparison of Counter functions and dict functions

Function	*Counter[key]*	*dict[key]*
Run-time	0.14	0.08
Function	*Counter.update ([key])*	*dict.update ({key:value})*
Run-time	1.60	0.23
Function	*Counter[key]+=1, Counter[key]-=1*	*dict[key]+=1, dictc[key]-=1*
Run-time	0.28	0.14
Function	*Counter.most_common(1)[0][1]*	*find_MCV_dict*
Run-time	14.02	26.05

3.2 카운터 형식을 리스트 형식으로 변경 구현하여 고속화하는 방법

카운터 형식과 리스트 형식은 데이터를 기록하는 방식이 다르며, 이에 따라 활용되는 용도도 달라진다. 단순 데이터 입출력의 경우 리스트 형식이 값의 호출이 빠르기 때문에 카운터 형식보다 빠르고, 카운터 형식은 해시테이블의 사용에 의해 상대적으로 많은 메모리를 소요해 리스트 형식을 사용하는 것이 적절하다. 그러나 입력된 값의 검색, 중복성 확인, 비교 등을 통한 데이터 재정렬 등의 기능이 필요한 경우 해시테이블을 사용하는 카운터 형식이 더 빠르며 구현 방법에 따라 메모리 측면에서도 더 높은 효율성을 가질 수 있다. 간단한 예로 0 에서 255 사이의 임의의 정수를 10000 개 입력할 경우 카운터는 중복을 체크해 256 개의 데이터를 기록해 5192 바이트, 리스트의 경우 10000 개를 모두 입력하여 43816 바이트를 소요하여 리스트에서 더 많은 메모리가 소요된다. 이처럼 용도에 따라 데이터 형식의 선택은 중요하지만 실제 구현에서는 그 용도를 명확하게 구분지어 형식을 선택하는 것은 어려운 경우가 많다. 여기서는 한 예로 임의의 데이터가 입력되지만 데이터의 표본공간은 한정될 때의 카운터 형식을 리스트 형식으로 변경 구현하여 고속화 하는 방법과 그 효과를 정량적으로 제시한다.

Counter

key	'c'	'a'	'ba'	1	4
count	1	2	1	3	1

list

order	0	1	2	3	4	5	6	7
value	'c'	'a'	'ba'	1	4	1	1	'a'

그림 4. 카운터 형식과 리스트 형식의 데이터 기록
Fig. 4. Data recording of Counter type and list type

이러한 예는 통계적 데이터 등에서 빈번히 발생할 수 있을 것으로 보인다.

카운터 형식은 사전을 이용해 구현되어 입력 값인 키(key)와 키의 입력 횟수인 카운트를 연결한 쌍을 기록하며, 키의 순서 또는 대소를 구분하지 않는다. 예를 들어 카운터 형식에 'c', 'a', 'ba', 1, 4, 1, 1, 'a' 이 순서대로 입력된다면 그림 4의 카운터(Counter)와 같이 기록된다. 본 논문에서는 카운터 형식의 함수 중 키에 대응하는 카운트를 호출하는 함수(Counter[key])와 카운트를 갱신하는 함수(Counter.update([key]). Counter.subtract([key])), 그리고 키를 카운트의 크기 기준으로 정렬하는 함수(Counter.most_common(n))에 대해 다룰 것이다. 리스트 형식은 입력 순서(order)대로 값(value)을 기록한다. 예를 들어 리스트 형식에 'c', 'a', 'ba', 1, 4, 1, 1, 'a' 이 순서대로 입력된다면 그림 4의 리스트(list)와 같이 기록된다.

그러나 카운터에 입력되는 키의 범위가 고정될 경우 카운터의 키를 0 에서 시작하는 양의 정수로 치환할 수 있다. 그림 4의 예에 적용할 경우 'c'를 0, 'a'를 1, 'ba'를 2, 1 을 3, 4 를 4 로 치환할 수 있다. 이 경우 그림 5와 같이 리스트를 크기 5 로 할당하여 순서(order)를 카운터의 키(key)로, 값(value)를 카운트(count)로 대응시키는 것으로 리스트 형식으로 카운터 형식과 동일한 데이터를 기록할 수 있으며, 카운터 함

Counter

key	'c'	'a'	'ba'	1	4
count	1	2	1	3	1

list

order =key*	0	1	2	3	4
value	1	2	1	3	1

※ key* denotes the order according to key of Counter

그림 5. 카운터 형식을 리스트 형식으로 변경한 경우의 데이터 기록
Fig. 5. Data recording of list type that is changed from Counter type

수의 기능에 따라 Counter[key]는 list[key*]로, Counter.update([key])를 list[key*]+=1로, Counter.subtract([key])를 list[key*]-=1로 대체할 수 있다. 여기서 key*는 카운터의 키를 양의 정수로 치환한 값을 의미한다. Counter.most_common(n)함수의 경우 완전히 동일하게 리스트로 표현되진 않지만 sorted(list)를 통해 유사하게 그 역할이 충족될 수 있다. 제시한 방법으로 카운터 형식을 리스트 형식으로 변경 구현할 경우 0 에서 255 사이의 임의의 정수를 10000 개 입력할 경우 256 개의 데이터를 기록해 1112 바이트를 소요하여 메모리 측면에서는 약 4 배의 개선을 확인할 수 있다.

여기서는 카운터 함수를 리스트 함수로 대체한 각 함수와 이를 이용해 구성한 통계 값 중 하나인 최빈값을 구하는 함수에 대해 구동시간을 비교한 결과를 제시한다. 나아가 추가 고속화 가능성을 확인하기 위하여 C에서 리스트 함수의 구동시간을 제시한다. 리스트 형식의 경우 Python과 C의 코드가 유사하여 따로 C 코드를 제시하진 않는다. 표 5는 기록된 데이터 수가 256 개 일 때, 단일 함수를 100 만 번 반복 구동한 시간을 제시한다. 주의할 것은 제시하는 것과 같이 카운터 형식을 리스트 형식으로 변경 구현하기 위해서는 키를 0 에서 시작하는 유한개의 양의 정수로 사전 치환할 수 있어야만 한다. 그렇지 못할 경우 카운터 형식을 리스트로 대체하기 위해 리스트에서 키를 검색하는 등의 오버헤드가 추가 발생하여 오히려 효율이 떨어질 수 있다.

표 5. 카운터와 리스트 형식 함수의 구동시간(초) 비교
Table 5. Run-time(seconds) comparison of Counter and list type functions

Data type	Counter in Python	list in Python	list in C
Function	*Counter[key]*	*list[key*]*	*list[key*]*
Run-time	0.14	0.01	0.0003
Function	*Counter.update([key]), Counter.subtract([key])*	*list[key*]+=1, list[key*]-=1*	*list[key*]+=1, list[key*]-=1*
Run-time	1.60	0.049	0.0005
Function	*Counter.most_common()*	*sorted(list)*	*sort(list, list+255)*
Run-time	47.26	26.72	8.32
Function	*find_MCV_counter*	*find_MCV_list*	*find_MCV_list_C*
Run-time	26.13	18.35	0.02

3.2.1 리스트 형식으로 변환을 통한 카운터 형식 호출 및 갱신 함수의 고속화

키(*key*)에 따른 카운트를 출력하는 함수를 '호출 함수', 그 횟수를 더하거나 빼는 함수를 '갱신 함수'라 하자. 카운터와 리스트 형식에 따른 호출 함수는 각각 *Counter[key]*와 *list[key*]*이며, 갱신 함수는 *Couner.update([key])*, *Counter.subtract([key])*와 *list[key*]+=1*, *list[key*]-=1*이다. 호출 함수, 갱신 함수의 경우 구동시간이 입력된 데이터의 크기에 영향을 받지 않았다. 호출 함수의 경우 데이터 형식 변환으로 10 배, 갱신 함수의 경우 데이터 형식 변환으로 30 배의 속도가 향상 되었으며 C로 변환하는 것으로 각각 30 배, 100 배의 추가 속도 향상을 확인했다.

3.2.2 리스트 형식으로 변환을 통한 카운터 형식 정렬 함수의 고속화

키를 카운트에 따라 정렬하는 함수를 '정렬 함수'라 하자. 카운터와 리스트(list) 형식에 따른 정렬 함수는 각각 Counter.most_common, sorted(list)로 표현된다. 리스트의 경우 Python과 C++에서 각각 제공하는 기본 정렬 함수인 sorted와 sort를 이용하였는데, 이 함수는 기록된 값을 크기 순서로 재정렬하는 함수로 most_common과는 달리 카운트에 대응하는 키를 찾을 수 없다는 단점이 있다. 따라서 카운트에 따라 정렬한 킷값이 필요한 경우 카운터 형식으로 구현해야만 한다. 정렬 함수의 경우 기록된 값을 최소 1회 이상 비교해야 하기 때문에 기록된 데이터의 수에 따라

표 6. 정렬 함수에 대한 구동시간(초) 비교
Table 6. Run-time(seconds) comparison of sort functions

Program language	Function	Run-time with #data 32	Run-time with #data 64	Run-time with #data 128	Run-time with #data 256
Python	*Counter.most_common()*	3.31	14.08	22.18	47.26
	sorted(list)	2.08	4.92	11.43	26.72
C	*sort(list, list + #data - 1)*	0.39	1.30	3.42	8.32

※ "#data n" is denotes that the recording data number is n

구동시간이 결정될 것으로 예상된다. 표 6은 데이터 수를 32, 64, 128, 256 으로 변형하면서 100 만회 반복 구동시간을 측정한 결과를 제시한다.

실험 결과 데이터 크기 증가에 따라 정렬 함수의 구동시간도 증가되었으며, Counter.most_common(), sorted(list), sort(list, list + #data －1) 각각의 구동시간 증가량은 0.187, 0.111, 0.035 이다. 정렬 함수의 경우 데이터 형식 변환으로 2 배의 속도가 향상되었으며, C 변환으로 5 배의 속도가 향상되었다.

3.2.3 리스트 형식으로 변환을 통한 카운터 형식 최빈값 함수의 고속화

카운트가 가장 높은 키를 찾는 함수를 '최빈값 함수'라 하자. 이 함수는 기록된 모든 값을 한번 씩 비교하여 가장 큰 값의 순서를 출력하는 방식으로 그림 3의 의사 코드와 같이 구현된다. 카운터와 리스트 형식으로 따른 그림 3과 같이 함수를 구현한 것을 각각 *find_MCV_counter*, *find_MCV_list*라 하겠으며, 동일하게 리스트 형식을 C에서 구현한 것을 *find_MCV_list_C*라 하겠다. 최빈값 함수의 정렬 함수와 마찬가지로 기록된 데이터의 수에 따라 구동시간이 결정될 것으로 예상된다. 표 7은 데이터 수를 32, 64, 128, 256 으로 변형하면서 100 만회 반복 구동시간을 측정한 결과를 제시한다.

실험 결과 최빈값 함수의 경우 데이터 형식 변환으로 1.5배의 속도가 향상되었으며, C 변환으로 700 배 이상의 속도가 추가 향상되었다. 최빈값 함수의 경우 find_MCV_Counter 보다 정렬 함수인 *Counter.most_common(1)[0][1]*로 표현하는 것이 더 효율적이다.

Ⅳ. 카운터 형식에 대한 고속화 방법의 응용

4.1 NIST의 multiMCW 엔트로피 추정 함수에 대한 고속화 연구 결과 분석

4.1.1 분석 개요

multiMCW는 NIST의 엔트로피 추정 도구 내부의 10 개 엔트로피 추정 함수 중 가장 많은 시간이 소요되어 기존 연구에서 700 배의 속도를 향상 시켰다. 그러나 기존 연구에서는 속도향상의 원인이 명확히 제시되지 못한 아쉬움이 있어, 제안한 방법을 활용하여 기존의 파이썬 프로그램에 대한 고속화 연구인 multiMCW 엔트로피 추정 함수에 대한 속도 향상의 원인을 함수 단위로 분석한다. 먼저 multiMCW의 구동속도에 직접적인 영향을 주는 5 개 함수에 대해 밝히고, 각 함수의 고속 구현 코드를 제시한다. 다음으로 위의 고속화 분석 결과를 이용하여 고속화 전•후의 구동 속도를 제시하였으며, 이를 바탕으로 multiMCW의 구동시간을 예상한 결과와 실제 구동시간을 비교하여 분석의 적절성을 확인하였다.

한편, 실험에 사용된 C는 디버그(debug) 모드와 릴리즈(release) 모드로 나누어 실행될 수 있는데, 그 차이는 최적화 여부로 나눌 수 있다. 릴리즈 모드는 최적화를 통해 상대적으로 디버그 모드보다 빠르고 효율적으로 구동[13]되어 실제 프로그램 배포에는 릴리즈 모드로 배포되는 것이 일반적이다. 이에 따라 Ⅲ장 2절의 C 실험 결과는 릴리즈 모드를 기준으로 측정한 결과를 제시하였다. 그러나 릴리즈 모드의 경우 최적화로 인해 구동 환경과 데이터 특성에 따라 그 실험 결과의 변동이 크기 때문에 보다 정확한 실험을 위하여 Ⅳ장에서의 C를 이용한 실험은 디버그 모드로 실험하였다. 이 결과 기존 연구[9]에서 제시한 실험 결과와 Ⅳ장의 실험 결과에 있어 다소 차이가 발생하였지만 기존 연구를 보다 정확히 분석한 결과를 제시할 수 있었다.

4.1.2 multiMCW 프로세스 분석

multiMCW의 의사코드[8]는 그림 6과 같으며, 이 중 반복적으로 연산이 소요되는 그림 6의 3.a.i.에서 실질적인 구동시간을 결정한다. 3.a.i.에는 총 4 개의 카운터 함수와 1개의 카운터 함수를 이용한 사용자정의 함수로 이루어진다. 5 개의 함수와 multiMCW에서의 역할은 표 8과 같다.

표 7. 최빈값 함수에 대한 구동시간(초) 비교
Table 7. Run-time(seconds) comparison table of finding MCV functions

Program language	Function	Run-time with #data 32	Run-time with #data 64	Run-time with #data 128	Run-time with #data 256
Python	*find_MCV_counter*	5.11	10.02	18.75	26.13
	find_MCV_list	2.71	4.61	9.42	18.35
C	*find_MCV_list_C*	0.005	0.006	0.01	0.02

※ "#data n" is denotes that the recording data number is n

Pseudo-code of multiMCW

Given the input $S=(s_1,...,s_L)$, where $s_i \in A=\{x_1,...,x_k\}$,

1. Let window size be $w_1=63, w_2=255, w_3=1023, w_3=1023, w_4=4095$, and $N=L-w_1$. Let correct be an array of N Boolean values, each initialized to 0.
2. Let scoreboard be a list of four counters, each initialized to 0. Let frequent be a list of four values, each initialized to Null. Let winner = 1
3. For $i=w_1+1$ to L
   a. For $j=1$ to 4,
      i. If $i>w_j$, let frequent be the most frequent value in $(S_{i-w_j}, S_{i-w_j}+1, ..., S_{i-1})$. If there is a tie, then $frequent_i$ is assigned to the most frequent value that has appeard most recently.
      ii. Else, let $frequent_i$=NULL
   b. Let prediction = frequent_winner.
   c. If(prediction = s_i), let $correct_j = scoreboard_j + 1$
   d. Update the scoreboard. For $j=1$ to 4 if
      i. If $(frequent_j = s_{i)}$
         1) Let $scorboard_j += 1$
         2) If $scoreboard_j \geqq scoreboard_{winner}$ $winner=j$

After step 4, the entropy is calculated using $correct$.

그림 6. multiMCW 함수의 의사코드
Fig. 6. The pseudo-code of multiMCW

4.1.3 multiMCW 고속화 분석

여기서는 표 8에서 제시된 multiMCW의 주요 함수

표 8. multiMCW의 주요 함수
Table 8. The key functions of multiMCW

Function	Role of the function
Counter[key]	Call the count of the key.
Counter.update([key])	Add the count of the key by 1.
Counter.subtract([key])	Subtract the count of the key by 1.
Counter.most_common()[0][1]	Find the most frequent value.
mostCommon(Counter, data)	If the most common value is tied, then return the mot frequent value that has appeared most recently.

5 개의 사용 위치, 사용 목적을 언급하며 속도 지연 요소를 제시하고 이에 대한 고속화 방법 및 그 효과를 정량적으로 제시한다. 나아가 정량화된 값으로 계산한 multiMCW의 예상 구동시간과 실제 구동시간과의 차이를 비교하여 분석의 적절성을 확인한다.

multiMCW의 실질적 구동시간을 결정하는 그림 6의 3.a.i.를 구현한 NIST의 코드와 이를 개선한 코드는 각각 그림 7의 왼쪽, 오른쪽과 같이 구현된다. SP 800-90B는 입력 데이터의 수와 크기(1~8 bits)를 지정할 수 있다. 권장 데이터 수는 100 만 개로, 이 경우 그림 7의 코드를 약 400 만 회를 반복하게 된다. 이 코드에서는 총 5 개의 함수가 사용되며 이 함수들의 역할은 표 8과 같다.

1) 호출 함수와 갱신 함수에 대한 고속화 분석

*counters[j], counter[j].subtract*와 *counters[j].update*는 카운터의 호출 함수와 갱신 함수로 NIST 코드의 2, 3, 4 번째 줄에 사용되었으며, 개선 코드는 동

Line	NIST's multiMCW	Improved multiMCW
1	if i>w[j]+1:	if i>w[j]+1:
2	counters[j].subtract([S[i-w[j]-2]])	list[j][S[i-w[j]-2]] -= 1
3	counters[j].update([S[i-2]])	list[j][S[i-2]] += 1
4	if counters[j][S[i-2]] == counters[j].most_common()[0][1]:	maxcnt = 0
5	frequent[j] = S[i-2]	for t in range(k):
6	else:	if list[j][t] > maxcnt:
7	frequent[j] = mostCommon(S[i-w[j]-1:i-1],counters[j])	maxcnt = window[j][t]
8		frequent[j] = mostCommon(list[j],S[i-w[j]-1:i-1],maxcnt)

※ i denotes the order of inputted data and j denotes the index of window

그림 7. NIST의 multiMCW 코드와 개선한 multiMCW 코드
Fig. 7. The NIST's multiMCW code and its improved code

일한 기능을 수행하도록 각각 2, 3, 6 번째 줄과 같이 구현하였다. Python에서 카운터 형식과 리스트 형식의 갱신 함수와 호출 함수의 구동시간은 Ⅱ 장과 동일하게 적용하였지만 C의 경우 실험 환경을 릴리즈 모드에서 디버그 모드로 변경함에 따라 측정 결과에 차이를 갖는다. 디버그 모드에서 갱신 함수와 호출 함수는 각각의 100 만회 구동시간은 0.004, 0.002 초의 구동시간을 갖는다. 분석 결과 카운터에서 리스트로 변경함으로 호출 함수의 경우 10 배, 갱신 함수의 경우 68 배의 속도가 향상되었으며, 파이썬의 리스트 형식을 C로 구현할 경우 릴리즈 모드의 경우 호출 함수의 경우 46 배, 갱신 함수의 경우 97 배 속도가 향상되는 반면, 디버그 모드의 경우 3 배, 18 배의 속도가 향상되었다.

2) 최빈값 함수에 대한 고속화 분석

*Counter.most_common*은 정렬 함수로 기록된 값을 크기 순서로 정렬하는 기능을 갖는 함수로, NIST 코드의 4 번째 줄에 사용되었다. 그러나 multiMCW에서는 *Counter.most_common* 함수를 최빈값을 찾는데 사용하여 그림 3의 최빈값을 찾는 함수 코드 보다 구동시간이 1.3 배 소요된다. 그림 7의 개선된 코드는 리스트를 이용해 find_MCV_list로 4~7 번째 줄과 같이 구현하였다. 표 9는 Ⅲ 장 2 절 c 항에서 제시한 *Counter.most_common*, *find_MCV_list*, *find_MCV_list_C*에 대해 샘플 크기 1~8 bits, 샘플 개수 100 만 개를 기준으로 구동시간을 측정한 결과다. 샘플 크기에 따라 각각 2, 4, 8, ..., 256 개의 데이터가 기록되어 그 구동시간도 샘플 크기의 증가와 동일한 비율로 증가하였다. 최빈값 도출 함수의 100 만회 반복 구동시간은 "구동시간 = 증가비율 × (기록된 데이터 수) + 고정 소요 시간"과 같이 계산할 수 있다. *Counter.most_common*, *find_MCV_list*, *find_MCV_list_C*의 샘플 개수에 따른 100 만 회 구동시간 증가 비율은 각각 0.2469, 0.032, 0.0019 로 구동시간에 대한 식은 표 10 과 같이 산출된다.

이에 더하여 *Counter.most_common* 함수는 *mostCommon* 함수 내부에서 최빈값을 찾을 때 사용되는데, 이 때 계산되는 최빈값은 *mostCommon* 함수의 호출 전에 기 계산된 값이다. 개선된 코드에서는 기 계산된 최빈값을 *most_Common* 함수에 입력하여 함수의 중복 구동을 피했다.

mostCommon 함수는 최빈값이 두 개 이상일 경우 가장 나중에 기록된 값을 반환하는 함수로, 그림 7의 왼쪽 코드와 같이 구현되었다. 이때 *mostCommon* 함수 내부에는 *Counter.most_common*, *len(list)*, *list(list)*, *set(list)*, *list.index(count)* 함수가 있지만 *Counter.most_common* 함수의 경우 multiMCW에서 단일로 구동되는 함수로 *mostCommon* 함수의 내부 함수와는 독립적으로 다루도록 하겠다. *mostCommon* 함수 1 회 구동 간에 *len(list)*, *list(list)*, *set(list)*는 1 회 구동되며 *list.index(count)* 함수는 데이터 상태에 따라 구동 횟수가 결정되는데, 실험 데이터의 경우 평균 4 회 구동되었다. 각 함수의 호출 횟수 및 샘플 크기에 따른 단일 함수의 100 만회 반복 구동시간은 표 11과 같다. 실험 결과에 따르면 *len(list)*, *list(list)*, *set(list)*의 구동시간은 데이터의 크기와 관계없이 고정적이었으며, *list.index(count)*는 데이터 크기가 증가함에 따라 구동시간도 증가하였다.

mostCommon 함수를 개선한 함수는 그림 8의 오른쪽 코드로 외부에서 계산된 최빈값을 받아와 최빈값 도출 함수의 중복 사용을 피하였으며, 리스트의 값 호

표 10. 데이터 수에 따른 최빈값 도출 함수의 구동시간(초)
Table 10. Run-time(seconds) of the finding MCV function according to the number of data

Function	Run-Time
Counter.most_common	Run-time = 0.2469 x #data + 0.1904
find_MCV_list	Run-time = 0.032 x #data + 0.5492
find_MCV_list_C	Run-time = 0.0019 x #data - 0.0018

※ "#data n" is denotes that the recording data number is n

표 9. NIST의 multiMCW에서 최빈값을 구하는 함수와 이를 개선한 함수의 구동시간(초) 비교
Table 9. Run-times(seconds) comparison of the functions finding a most common value and its improved function.

Function	The number of samples							
	2	4	8	16	32	64	128	256
Counter.most_common()[0][1]	2.10	2.27	2.86	4.15	7.25	14.13	29.77	64.94
find_MCV_list	0.44	0.56	0.81	1.22	2.02	3.71	7.05	13.78
find_MCV_list_C	0.004	0.007	0.01	0.03	0.06	0.13	0.24	0.50

Line	NIST's *mostCommon* function code	Improved *mostCommon* function code
1	def mostCommon(S,c):	def mostCommon(list, S, maxcnt):
2	maxcount = *c.most_common()[0][1]*	for i in *S[::-1]*:
3	maxsymb = None	if *list[i]* == maxcnt:
4	lastindex = *len(S)*	return i
5	reverse = *S[::-1]*	
6	for s in *set(S)*:	
7	if (*c[s]* == maxcount) and (*reverse.index(s)* < lastindex):	
8	lastindex = *reverse.index(s)*	
9	maxsymb = s	
10	return maxsymb	

그림 8. NIST의 mostCommon 함수와 개선한 mostCommon 함수의 코드
Fig. 8. The codes of NIST's mostCommon function and its improved function

출과 호출된 값이 최빈값인지 확인하는 함수로 간략히 구현하여 기존 *mostCommon* 함수에서 사용하는 다양한 내부함수의 사용을 피했다. *mostCommon* 함수와 개선한 *mostCommon* 함수에 대한 샘플 크기 1~8 bits, 샘플 개수 100 만 개를 기준으로 구동시간을 측정한 결과는 표 11과 같다. 샘플 크기에 따라 각각 2, 4, 8, ..., 256 개의 데이터가 기록되어 그 구동시간도 샘플 크기의 증가와 동일한 비율로 증가하였다. *mostCommon* 함수의 100 만회 반복 구동시간은 "구동시간 = 증가비율 × (기록된 데이터 수) + 고정 소요 시간"과 같이 계산할 수 있다. 기존 *mostCommon*, 개선한 *mostCommon*, 개선한 *mostCommon*함수를 C로 구현한 함수에 대한 샘플 개수에 따른 100 만 회 구동시간 증가 비율은 각각 0.152, 0.026, 0.0005 로 표 12과 같이 산출된다. *mostCommon* 함수의 최빈값 도출 함수와 차이가 나는 특징은 고정 시간이 구동시간에 큰 영향을 준다는 것이다.

4.1.4 고속화 분석 결과와 실제 실험 결과 비교

표 14는 분석한 5 개 함수의 단일 함수 구동시간을 기반으로 multiMCW의 구동시간을 계산한 시간(Expected run-time)과 실제 구동시간(multiMCW)을 비교한 결과를 보여준다. 계산된 구동시간은 입력 샘플 수 256 개를 기준으로 5 개 함수에 대해 (각 함수의 구동 예상 시간)×(각 함수의 multiMCW에서의 실제 구동 횟수) 값을 다 더한 값이므로 표 14, 그림 9와 같다. 기존, 개선, C에서 개선 multiMCW 함수 모두 예상 구동시간과 실제 구동시간과의 차이가 크지 않은

표 11. mostCommon 내부 함수의 호출 횟수 및 구동시간(초) 측정
Table 11. Run-time(second) and the number of calls for the inner functions of mostCommon

Function	The number of calls	The number of samples							
		2	4	8	16	32	64	128	256
len(list)	1	0.07	0.06	0.06	0.06	0.06	0.07	0.06	0.06
list[::-1]	1	3.29	2.90	2.90	2.86	2.83	2.85	2.86	2.85
set(list)	1	27.43	27.50	27.60	28.68	28.05	28.42	29.59	30.41
list.index (count)	≈4	0.26	0.27	0.31	0.40	0.55	0.83	1.57	2.97

표 12. mostCommon 함수와 개선한 mostCommon 함수의 구동시간(초) 비교
Table 12. Run-times(seconds) comparison of the NIST's mostCommon function and improved mostCommon function.

Function	The number of samples							
	2	4	8	16	32	64	128	256
mostCommon	35.53	35.06	35.82	37.07	39.80	44.66	56.70	72.98
Improved mostCommon	6.80	6.08	6.10	6.30	6.73	7.63	9.38	12.82
Improved mostCommon_C	0.009	0.01	0.01	0.01	0.03	0.05	0.10	0.20

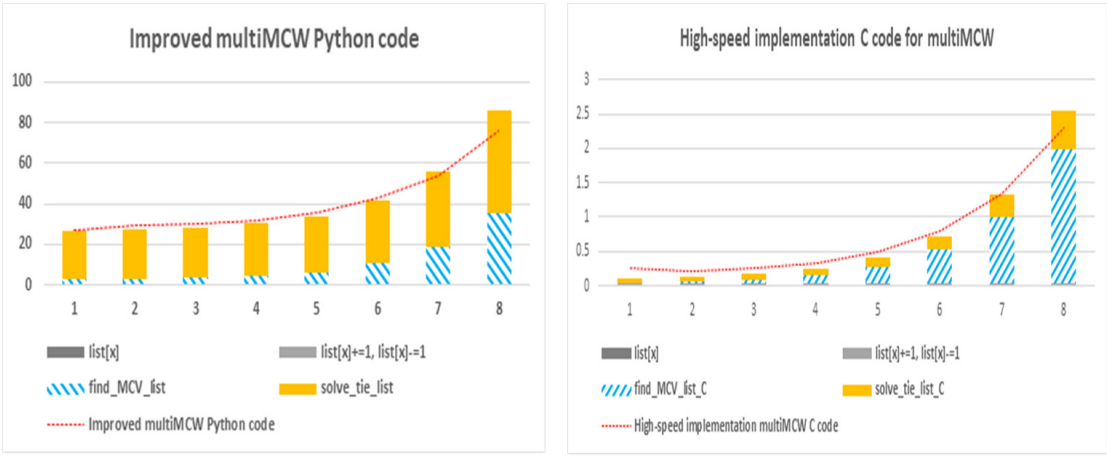

그림 9. 고속화 분석 결과를 기반으로 도출한 구동시간과 실 구현물의 구동시간 비교 결과(x 축 : 데이터 크기, y 축 : 초)
Fig. 9. Comparison of the run-time(seconds) derived from the analysis results and the implemented multiMCW(x-axis : data size, y-axis : seconds)

것을 확인할 수 있다. 이 결과는 고속화 분석이 적절히 이루어졌음을 의미한다.

표 13. mostCommon 함수와 개선한 mostCommon 함수의 구동시간(초)
Table 13. Run-time(seconds) of the mostCommon and the improved mostCommon functions according to the number of data

Functions	Run-time(seconds)
mostCommon	Run-time = 0.1525 x #data + 34.984
Improved mostCommon	Run-time = 0.0264 x #data + 8.4863
Improved mostCommon in C	Run-time = 0.0005 x #data + 0.0163

※ "#data n" is denotes that the recording data number is n

표 14. 기존, 개선, C에서 개선 multiMCW 함수의 분석 결과 기반 구동시간과 실제 구동시간 비교(초)
Table 14. Run-time(seconds) comparison of the NIST's, improved and improved in C multiMCW functions

	NIST's multiMCW	Improved multiMCW	Improved multiMCW in C
Expected	800.41	86.39	2.57
Implemented multiMCW	749.78	76.41	2.31

4.2 히스토그램 생성 프로그램 속도 비교

히스토그램이란 도수 분포표의 하나로 가로축에 계급을, 세로축에 계급에 따른 빈도를 나타내는 막대(bar) 그래프로 표현된다. 히스토그램의 계급은 카운터의 키로, 빈도는 카운트로 자연스럽게 표현된다. 제안하는 고속화 방법의 효과를 확인하기 위하여 카운터 형식과 리스트 형식 각각을 활용하여 히스토그램 생성 프로그램을 구현하였다. 구현된 히스토그램 생성 프로그램의 프로세스는 다음과 같다. 먼저 n 개의 실수를 입력 받는다. 다음으로 정수 단위로 계급을 구성하고, 계급에 따른 입력 횟수를 측정한다. 계급에 따른 입력 횟수 측정이 끝나면, 측정된 계급과 그에 따른 분포를 이용해 히스토그램을 그린다. 여기서 계급의 구성 및 그에 따른 입력 횟수 측정하는 코드에서 각각 리스트 형식과 카운터 형식으로 계급을 나누고 기록하였다. 히스토그램 생성 프로그램의 프로세스는 그림 10 과 같다.

제안하는 고속화 방법의 효율성을 보기위하여 그림

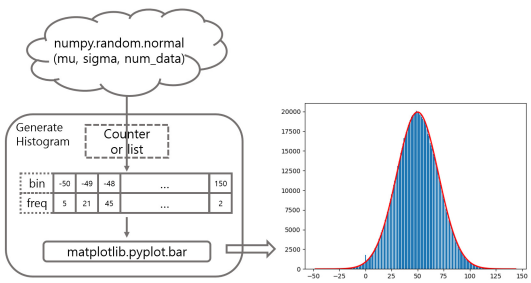

그림 10. 히스토그램 생성 프로그램 프로세스
Fig. 10. Generating histogram process

Line	Counter	list
1	c=Counter()	min_data = int(min(raw_data)) # overhead
2	for i in raw_data:	max_data = int(max(raw_data)) # overhead
3	c.update([int(i)])	l = [0 for i in range(max_data - min_data+1)]
4		for i in raw_data:
5		l[int(i)-min_data]+=1 # overhead

그림 11. 데이터 형식에 따른 계급 구성 및 입력 횟수 측정 코드
Fig. 11. Determining number of bins and counting its frequency codes according to the data types

11의 코드로 표현되는 계급의 구성과 계급의 입력 횟수 측정 부분의 속도를 측정해 비교하였다. 입력 받은 데이터는 정규분포를 따르는 난수를 출력하는 numpy의 random.normal 모듈의 출력으로, 이때 평균은 50으로, 표준편차는 10, 20, 30으로 설정하였으며, 그 결과 표준 편차가 증가함에 따라 계급의 수(bins) 또한 약 100, 200, 300으로 증가하였다. 입력한 데이터의 수는 각각 데이터 크기 10^4, 10^5, 10^6으로 변경하며 실험을 진행하였다. 카운터 형식 또는 리스트 형식으로 구현한 히스토그램 생성 프로그램에 정규분포를 따르는 출력 데이터를 입력하여 계급 구성 및 입력 횟수 측정 코드의 구동시간을 측정한 결과, 표 15와 같이 모든 데이터에 대해 4 배의 속도 차이가 발생하였다. 각 형식에서 사용된 함수는 갱신함수로 분석 결과에 따르면 *Counter.update([key])*를 *list[key*]+=1*로 변경 구현할 경우 10 배의 속도 향상이 예상되었지만, 여기서는 결과적으로 4 배의 속도 향상으로 상대적으로 낮은 비율로 속도가 향상되었다. 이 결과는 리스트 형식으로 구현할 경우 계급(bin)의 범위 지정에 따라 1, 2, 5 번째 줄과 같은 오버헤드에 의한 것으로 보인다. 결론적으로 몇몇 오버헤드에도 불구하고 카운터 형식의 프로그램을 리스트 형식으로 구현하는 것으로 충분한 속도 향상이 이루어지는 것을 확인할 수 있었다.

표 15. 카운터 형식 또는 리스트 형식으로 구현한 히스토그램 생성 프로그램의 구동시간(초) 비교 (카운터 형식 → 리스트 형식)
Table 15. Run-time(seconds) comparison of the histogram generating programs using Counter type or list type (Counter type → list type)

The num of bins / The num_data	100	200	300
10,000	0.026 → 0.006	0.025 → 0.006	0.025 → 0.006
100,000	0.269 → 0.064	0.263 → 0.062	0.258 → 0.064
1,000,000	2.615 → 0.594	2.614 → 0.602	2.668 → 0.587

V. 결 론

본 논문에서는 파이썬에 대한 고속화 방안으로 컬렉션의 카운터 형식이 사용되는 프로그램에 적용 가능한 두 가지 방법을 제시한다. 제시한 방법은 카운터

표 16. 카운터 와 리스트 형식 함수의 구동시간(초) 비교
Table 16. Run-time(seconds) comparison of Counter and list type functions

Data type	dict	Counter		Efficient expressed Counter	list	list in C
Function	*dict[key]*	*Counter[key]*			*list[key*]*	*list[key*]*
Run-time	0.08	0.14			0.01	0.0003
Function	*dict.update ({key:value})*	*Counter.update([key]), Counter.subtract([key])*		*Counter[key]+=1, Counter[key]-=1*	*list[key*]+=1, list[key*]-=1*	*list[key*]+=1, list[key*]-=1*
Run-time	0.23	1.60		0.30	0.05	0.0005
Function	-	*Counter.most_common()*			*sorted(list)*	*sort(list, list+255)*
Run-time	-	47.26			26.72	8.32
Function	*find_MCV_dict*	*Counter.most_common()[0][1]*	*find_MCV_Counter*	*Counter.most_common(1)[0][1]*	*find_MCV_list*	*find_MCV_list_C*
Run-time	26.05	47.26	26.13	14.02	18.35	0.02

형식을 사용하는 모든 코드에 대한 방법을 제시한 것이 아닌 함수의 사용 목적 또는 적용 데이터의 특징에 따라 개선 가능한 방법을 제시하고 이를 정량적으로 분석한 결과이다. 첫 번째 방법은 카운터 형식 함수의 필요 기능에 따른 효과적 표현법을 제시했다. 두 번째 방법은 임의의 데이터가 입력되지만 데이터의 경우의 수는 한정될 때, 카운터 형식을 리스트 형식으로 변경 구현하여 고속화하는 방법을 제시하며 그 효율성을 보였다. 각 방안에 대해 고속화 원인을 분석하였으며, 실험을 통해 그 효과를 정량적으로 제시하였다. 본문의 분석 결과를 요약하면 표 16과 같다. 나아가 제안하는 고속화 방법을 기반으로 기존 NIST의 엔트로피 측정 도구에 대한 고속 구현[9] 연구 결과인 multiMCW에 대한 700 배 속도 향상의 원인을 함수 단위로 분석하였으며, 카운터 형식과 리스트 형식을 각각 사용하여 구현한 히스토그램 생성 프로그램을 통해 제안한 고속화 방법의 적절성을 확인하였다. 본 논문의 분석 결과는 파이썬 프로그램을 구현할 때 데이터 형식 선정에 근거가 될 수 있을 것이며, 카운터 형식이 사용된 파이썬 프로그램의 고속화에 활용될 수 있을 것으로 기대한다.

본 논문의 모든 실험은 다음과 같은 환경에서 이루어졌다.

(1) 프로세서(Processor) : inter(R) Core(TM) i7-6700
(2) CPU : 3.40 GHz
(3) OS : Windows 10 / 64-bit
(4) 사용 프로그램 언어 : Python 3.6 32-bit, Visual Studio 2015 C

References

[1] H. C. Shin, S. J. Woo, and D. J, Choi, *Python3.2 programming*, 3rd Ed., Wikibook, Jun. 2015.

[2] *The ranking table for program language*, Retrieved Apr. 25, 2018. from https://www.tiobe.com/tiobe-index/

[3] *Data model in Python*, Retrieved Apr. 25, 2018. https://docs.python.org/3/reference/datamodel.html

[4] *Standard data type in Python*, Retrieved Apr. 25, 2018. https://docs.python.org/3/library/stdtypes.html

[5] *Collections type in Python*, Retrieved Apr. 25, 2018. https://docs.python.org/3/library/collections.html

[6] *Tensorflow tool*, Retrieved Apr. 25, 2018. https://github.com/tensorflow/tensorflow

[7] *NIST's entropy estimation tool*, Retrieved Apr. 20, 2018. https://github.com/usnistgov/SP800-90B_EntropyAssessment

[8] M. S. Turan, et al., "Recommendation for the entropy sources used for random bit generation," (2nd Draft) NIST SP 800-90B, Apr. 2016.

[9] W. Kim, Y. Yeom, and J. S. Kang, "High-speed implementation and efficient memory usage of min-entropy estimation algorithms in NIST SP 800-90B," *KIISC*, vol. 28, no. 1, pp. 25-39, Feb. 2018.

[10] W. Kim, H. Park, Y. Yeom, and J. S. Kang, "High-speed implementation of MultiMCW entropy estimation method," *2017 IEEE Conf. AINS*, pp. 60-63, Miri, Malaysia, Nov. 2017.

[11] Issues #31 : noniid_main.py Memory leak?, Retrieved Apr. 25, 2018. "https://github.com/usnistgov/SP800-90B_EntropyAssessment/issues/31"

[12] *Tree graph theory*, Retrieved Apr. 25, 2018. https://en.wikipedia.org/wiki/Tree_(graph_theory)

[13] *Release mode in C*, Retrieved Apr. 25, 2018. https://msdn.microsoft.com/en-us/library/wx0123s5.aspx

김 원 태 (Wontae Kim)

2017년 2월 : 국민대학교 수학과 학사
2017년 3월~현재 : 국민대학교 금융정보보안학과 석사과정
<관심분야> 난수성 분석 및 평가, 프로그램 고속구현, 대칭키 암호 분석

박 호 중 (Hojoong Park)

2015년 2월 : 국민대학교 수학과 학사
2017년 2월 : 국민대학교 금융정보보안학과 석사
2017년 3월~현재 : 국민대학교 금융정보보안학과 박사과정

<관심분야> 암호이론, 정보보호 알고리즘 및 프로토콜, 난수성 분석

염 용 진 (Yongjin Yeom)

1991년 2월 : 서울대학교 수학과 학사
1994년 2월 : 서울대학교 수학과 석사
1999년 2월 : 서울대학교 수학과 박사
2000년 4월~2012년 2월 : ETRI 부설연구소 책임연구원/팀장
2006년 12월~2007년 12월 : Columbia 대학교 방문연구원
2012년 3월~현재 : 국민대학교 정보보안암호수학과 부교수
2013년~현재 : 국민대학교 BK21+ 미래 금융정보보안 인력양성사업단 교수

<관심분야> 암호구현 및 분석, 보안시스템 평가

강 주 성 (Ju-Sung Kang)

1989년 2월 : 고려대학교 수학과 학사
1991년 2월 : 고려대학교 일반대학원 수학과 석사
1996년 2월 : 고려대학교 일반대학원 수학과 박사
1997년~2004년 : 한국전자통신연구원 선임연구원/팀장
2001년~2002년, 2010년 : 벨기에 루벤대학 COSIC 방문 연구원
2004년~현재 : 국민대학교 정보보안암호수학과 교수
2013년~현재 : 국민대학교 BK21+ 미래 금융정보보안 인력양성사업단 교수

<관심분야> 암호이론, 정보보안 프로토콜, 안전성 분석 및 평가