About Fingerprints

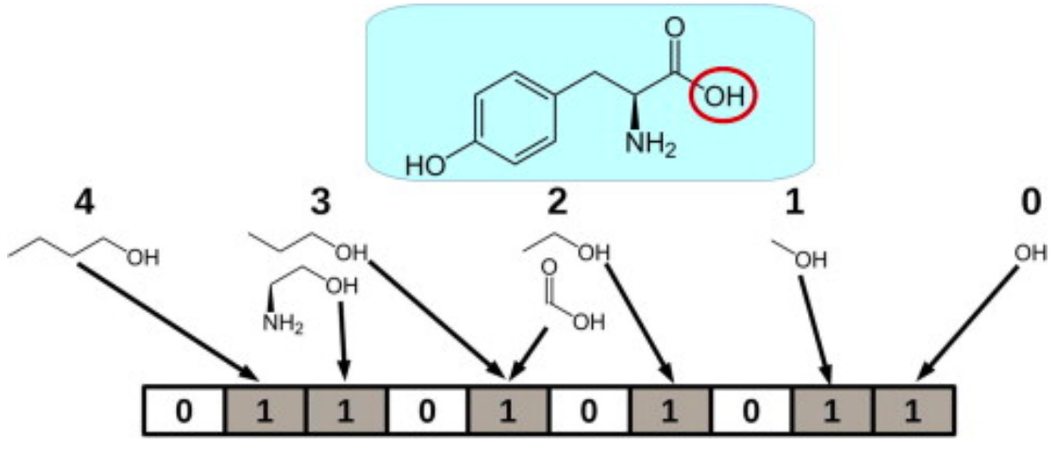
Chemoinformatics & Molecular Modeling

2020. 10. 29

Juyong Lee

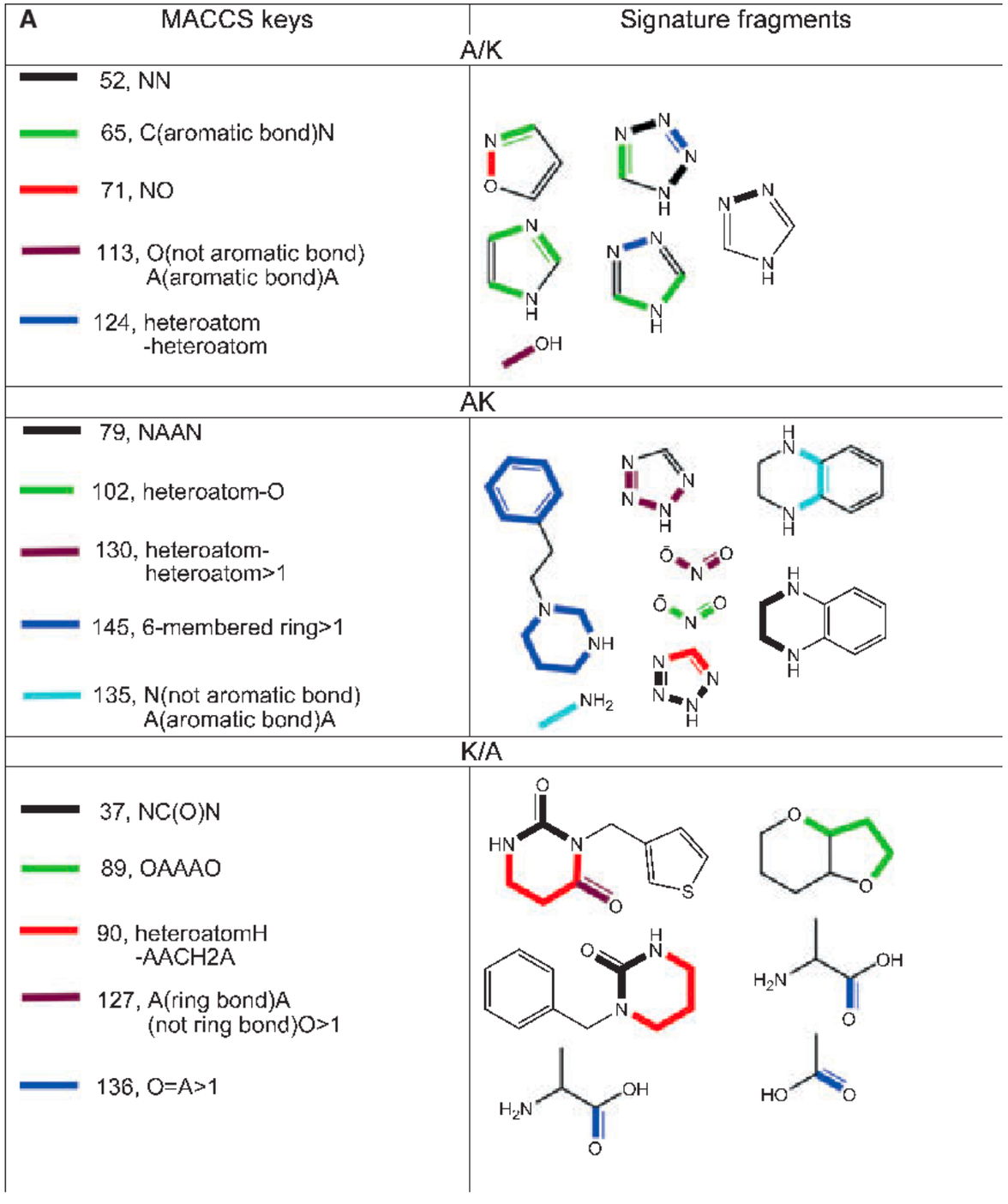
Chemical FingerPrints

- 분자 구조를 정해진 차원의 정수의 벡터로 변환하는 방법
- 목적: 동일하거나 비슷한 분자를 빠르게 검색하기 위해서 많이 사용된다.
- 컴퓨터를 이용해서 2차원의 분자 구조나 1차원 텍스트를 비교하는 것보다 1차원 정수 벡터를 비교하는 것이 빠르다.



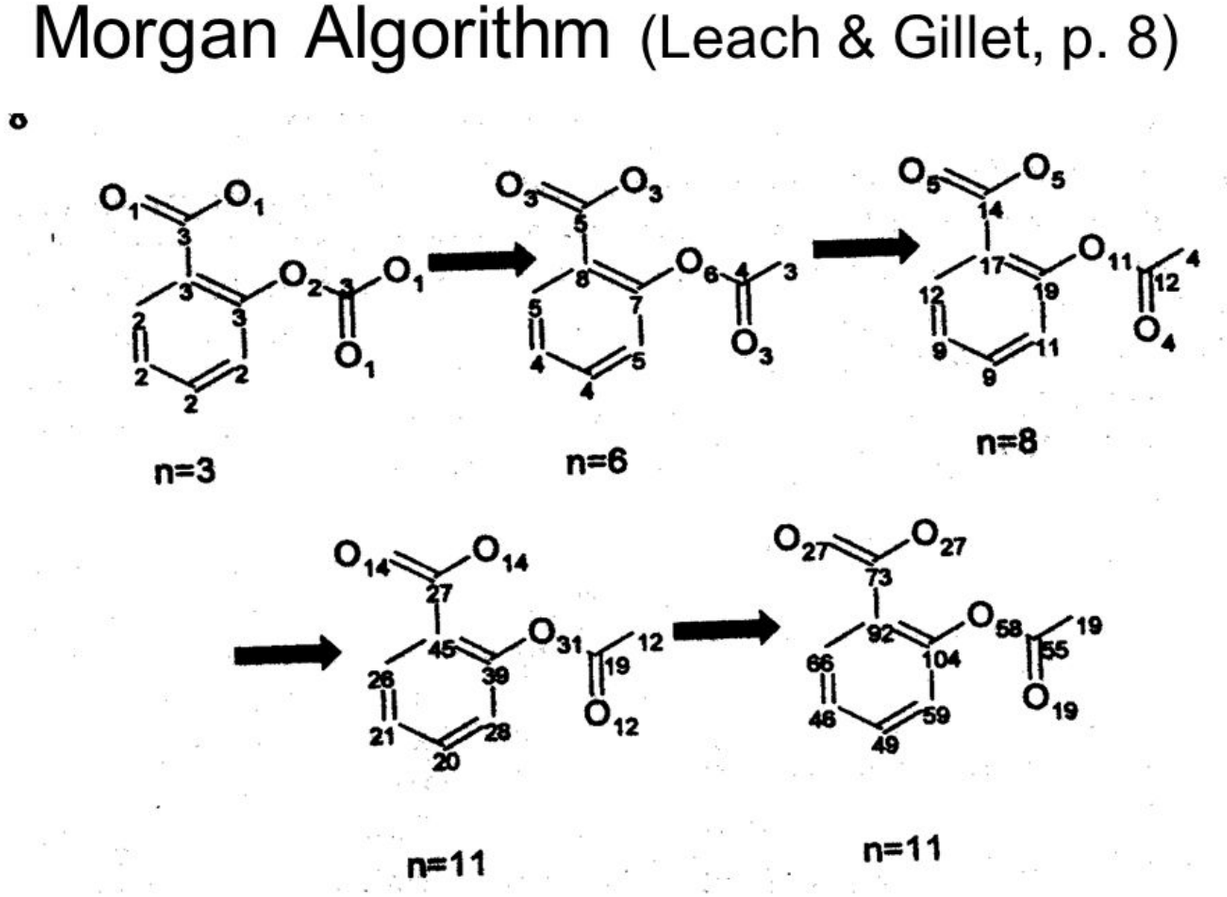
MACCS Key

- 자주 관찰되는 분자의 부분 구조를 미리 정한 후에 그 구조가 존재하는지 아닌지를 0과 1로 표현
- 현재 가장 많이 사용되는 버젼은 166개의 부분 구조가 정의된 MACCS166 임.
- 다시 말해, 분자를 166차원의 0과 1로 된 정수 벡터로 표현



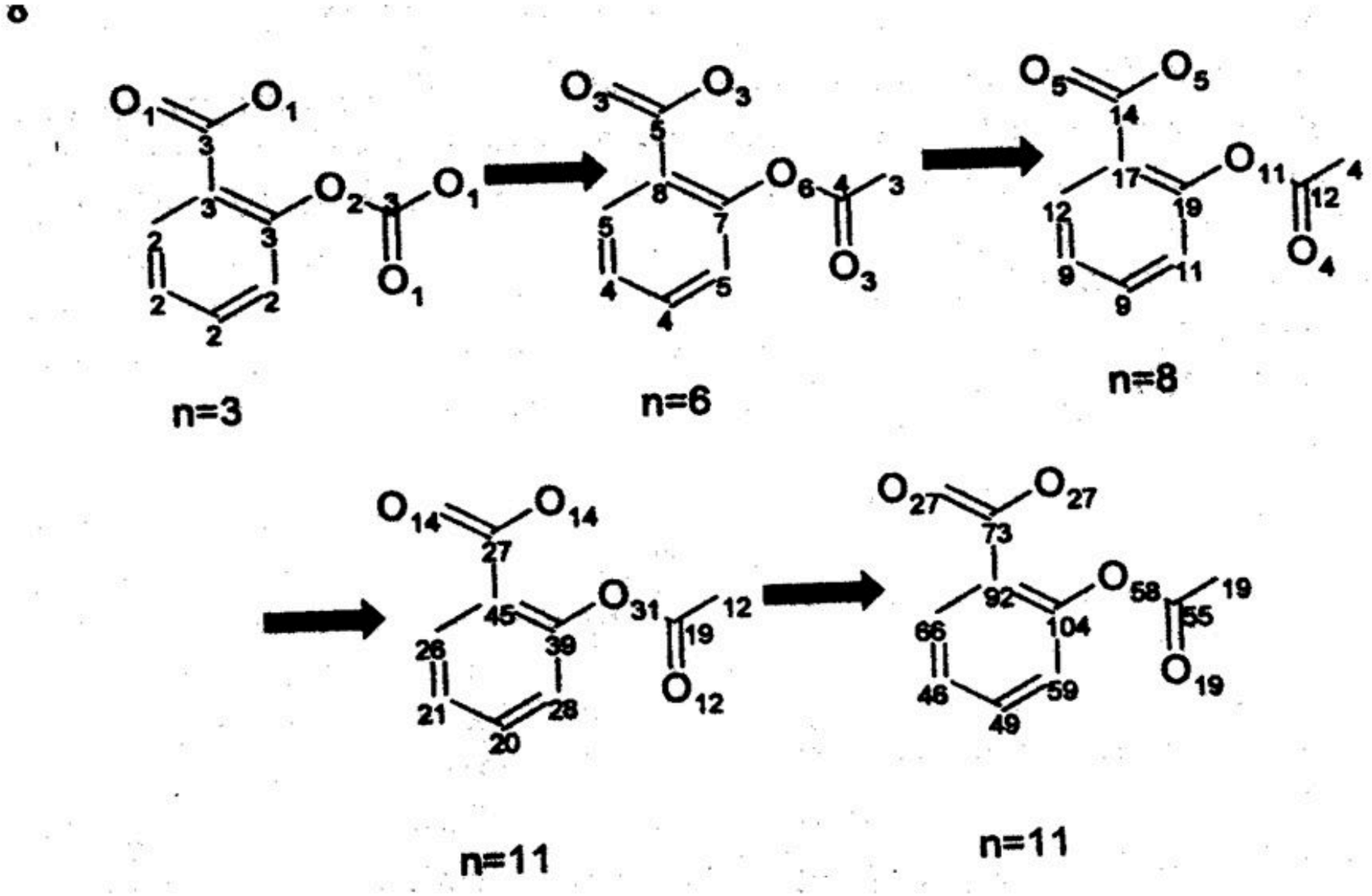
Morgan Algorithm

- 분자의 구조를 정수로 표현하기 위한 방법
- 1969년에 Morgan에 의해 제안됨.
- 1. 처음에는 각 원자에 연결된 원자의 개수를 부여한다. (connectivity)
- 2. 각 원자들에 연결된 원자의 connectivity 값을 모두 합한 값을 그 원자의 connectivity 값으로 다시 부여한다.
- 2번의 과정을 반복한다.
- 각 원자가 가지는 값이 더이상 변화하지 않을 때 반복을 멈춘다.



Morgan algorithm

Morgan Algorithm (Leach & Gillet, p. 8)



ECFP

- How to generate ECFP?
- First, assign **initial number**s to atoms based on the Daylight atomic invariants-derived rule.
  - Number of heavy atom neighbors
  - Atomic number
  - Atomic mass
  - Atomic charge
  - Number of attached hydrogens
  - Whether it is contained in a ring or not.
- Initial numbers are put into **a hash function**

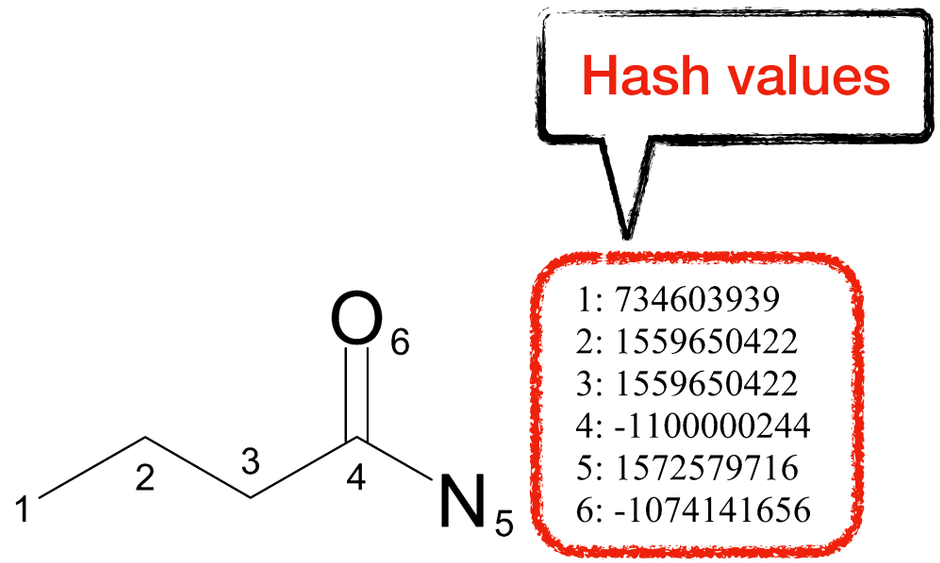


Figure 3. The initial atom identifiers for butyramide, calculated using the Daylight atomic invariants-derived rule. (Note that the hash function may return either positive or negative numbers for the identifiers.)

Hash?

- You must have heard about “hash” mostly from social media as a form of “hashtag”

- However, “hash” is a popular concept (algorithm) in computer science.

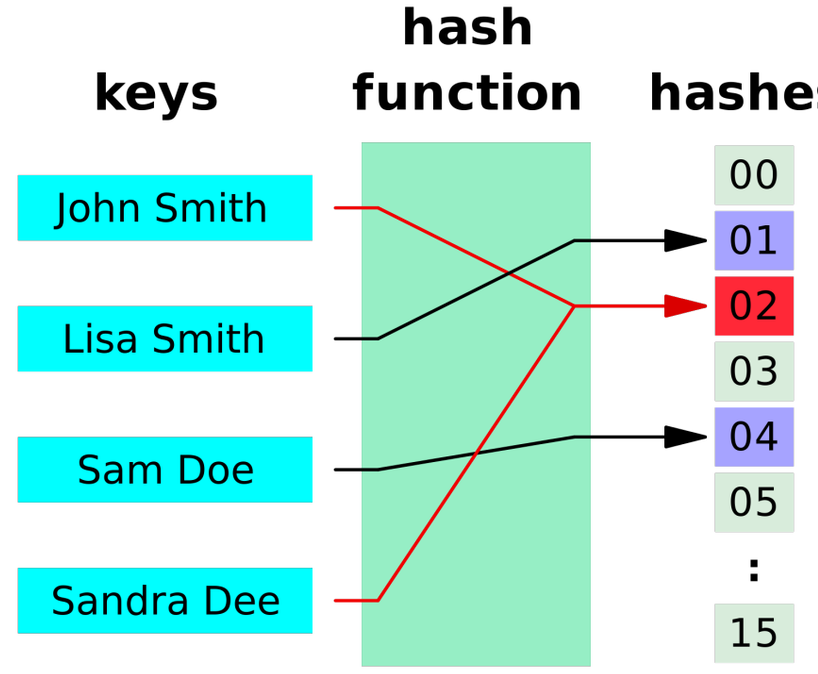
Hash function

- A hash function is any function that can be used to map data of arbitrary size to fixed-size values.

- 해시 함수(hash function)는 **임의의 길이의 데이터를 고정된 길이의 데이터로 매핑하는 함수**이다.

- The values returned by a hash function are called hash values, hash codes, digests, or simply hashes.

- 해시 함수에 의해 얻어지는 값은 해시 값, 해시 코드, 해시 체크섬 또는 간단하게 해시라고 한다.



사람의 이름을 2개의 이진 해시 값으로 변환하는 예시

Purpose of hash function

- 대용량의 데이터가 저장되어 있을 때, 특정 데이터를 찾거나 동일한지 아닌지를 매우 빠르게 판단할 수 있다.

- 해시 값이 동일하지 않으면 두 파일이 다른 파일이라는 것을 쉽게 판단할 수 있다.

- 그러나 해시 값이 같더라도 두 파일이 서로 다를 가능성이 존재한다. - 해시 충돌 (collision)

- 해시 값의 사이즈가 작으면 검색 속도는 빠르지만 충돌 확률은 증가한다.

- 반대로 해시 값의 사이즈가 크면 충돌 확률을 낮아지지만 검색의 속도는 느려진다.

Example of hash function

- 가장 간단한 해시 함수
- f(key) = key % a
  - 나머지 연산
- 임의의 정수를 a 보다 작은 정수로 변환시키는 해시 함수
- 충돌 확률이 높다!

Lets say, Hash1 (key) = key % 13

Hash2 (key) = 7 – (key % 7)

Hash1(19) = 19 % 13 = 6

Hash1(27) = 27 % 13 = 1

Hash1(36) = 36 % 13 = 10

Hash1(10) = 10 % 13 = 10

Hash2(10) = 7 – (10%7) = 4

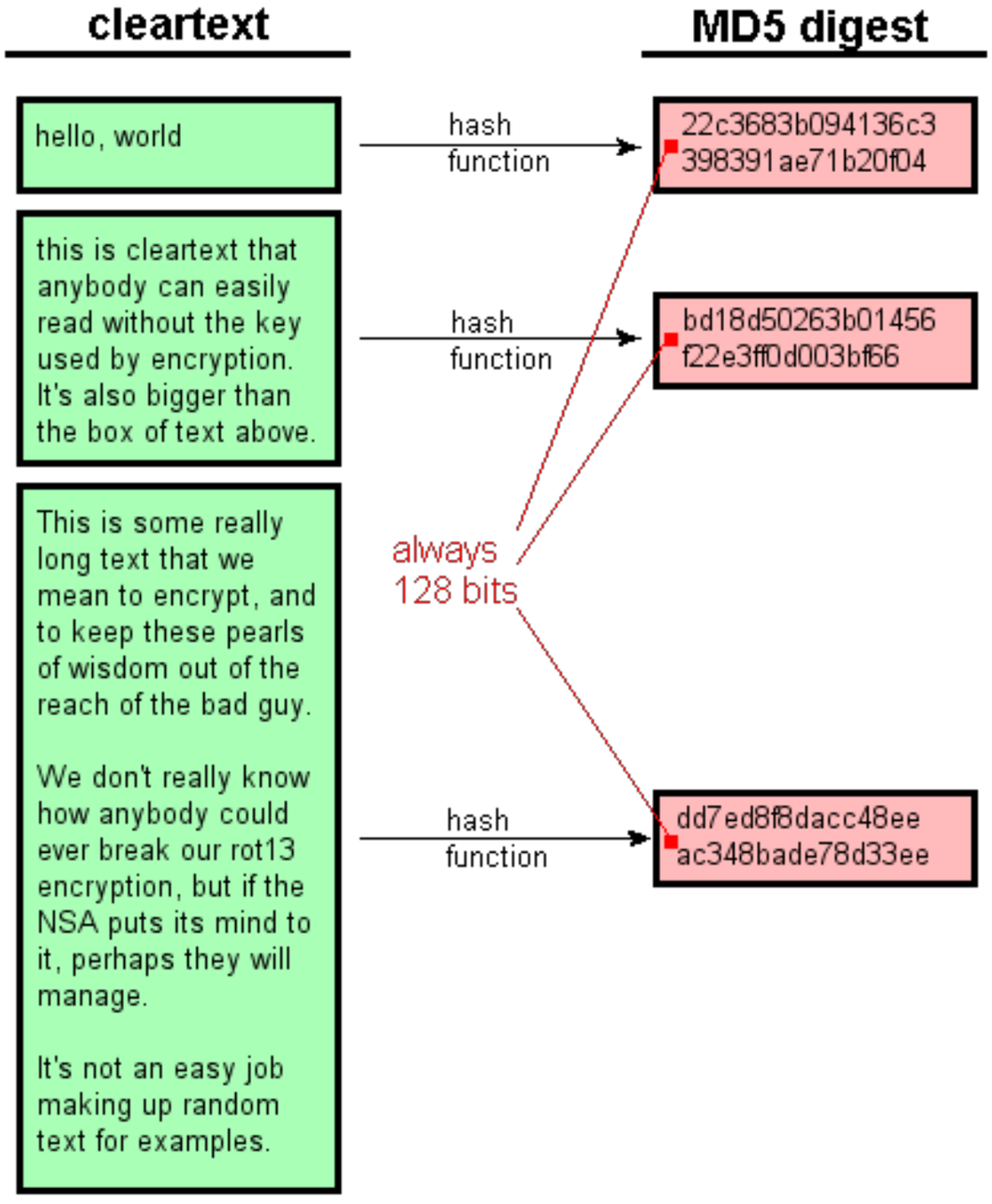
(Hash1(10) + 1*Hash2(10))%13= 1

(Hash1(10) + 2*Hash2(10))%13= 5

Collision

일상 생활에서 볼 수 있는 해시 함수의 예시

- MD5 checksum
- 임의의 크기의 파일을 128 bit로 바꾸어 준다.
- 파일이 변형되었는지 아닌지를 판단하는데 매우 유용하다.
  - torrent 프로그램의 핵심 원리
- 충돌 확률: 1.5E-25
  - 1초에 파일 하나씩 만든다면 대략 4경년 정도에 한 번 동일한 MD5 checksum을 가지는 다른 파일이 생성될 수 있다.
  - 더 큰 byte를 사용하면 확률은 더 낮아진다.



ECFP 만들기

- Atom number assignment가 끝나면 그 다음에는 다음 step에 따라서 각 atom의 정보를 리스트로 저장한다.
- 첫번째: 반복 횟수
- 두번째: 결합 차수에 따라서 결합되어 있는 원소들을 정렬
  - single, double, triple, and aromatic
- 세번째: attachment identifier와 결합차수를 저장한다.

ECFP 만들기 2

- Carboxylic acid carbond의 예를 살펴보자.
- 리스트의 첫번째 원소로는 1이 들어간다. (반복 횟수)
- 두번째는 원소의 hash value인 -1100000244가 들어감.
- 세번째 부터는 붙어있는 heavy atom의 정보가 들어감.
  - Bond order 순서대로.
- [1, -1100000244, 1, 1559650422, 1, 1572579716, 2, -1074141656].

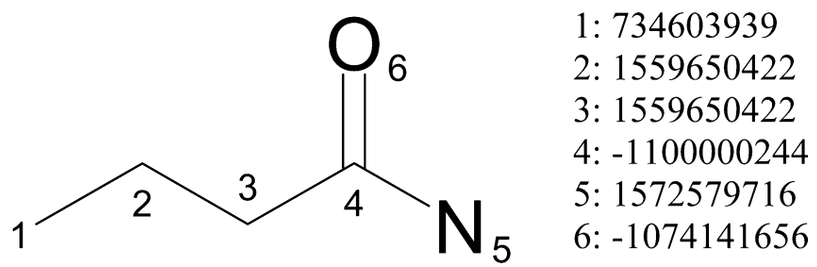


Figure 3. The initial atom identifiers for butyramide, calculated using the Daylight atomic invariants-derived rule. (Note that the hash function may return either positive or negative numbers for the identifiers.)

ECFP 만들기 3

- 위의 작업을 모든 원소에 대해서 반복한다.
- 그 뒤 생성된 list를 hash function에 넣는다.
- 결과
  - [734603939, 1559650422, 1559650422, -1100000244, 1572579716, -1074141656, 863188371, -1793471910, -1789102870, -1708545601, -932108170, 2099970318]

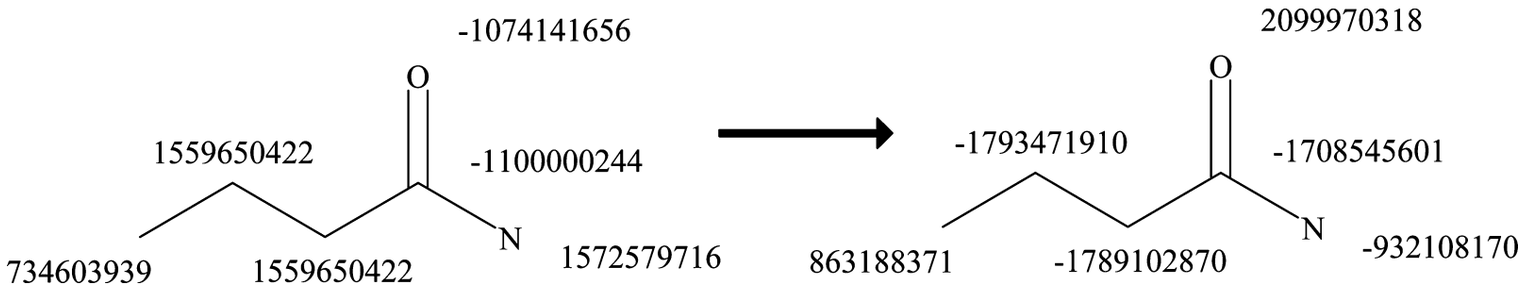


Figure 4. Generation of new identifiers by performing one iteration using butyramide. The initial atom identifiers are shown on the molecule on the left; after the updating process, each atom is given a new identifier, shown on the molecule on the right.

몇 번 정도 반복하는 것이 좋을까?

- 완벽하거나 엄밀한 rule은 없음.
- 일반적으로 similarity 혹은 identity 비교를 위해서는 2번 반복하면 충분하다고 알려져 있음.
- 성질 예측을 위해서는 3~4번 반복하는 것이 더 좋다고 알려져 있음.

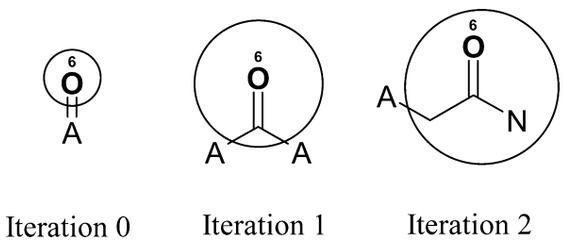


Figure 5. Effect of the iteration process on information stored in the identifier of the oxygen atom in butyramide. (The "A" atom type can map onto any atom type and is the only atom that may have connections that are not specified.) The sphere shows the *feature region* represented in the identifier after the given number of iterations. After zero iterations, only information about the atom itself and its connectivity are available. After one iteration, the identifier contains information from the core atom's immediate neighbors; in this case a carbonyl. After two iterations, atoms within two bonds of the core atom are included. At this point, it represents an aliphatic carboxylic acid amide, with no substituents on the nitrogen atom and exactly one substituent on the α carbon.

반복 횟수에 따라서 더 많은 주변 원자들의 정보를 보게 된다.

ECFP에 담기는 정보

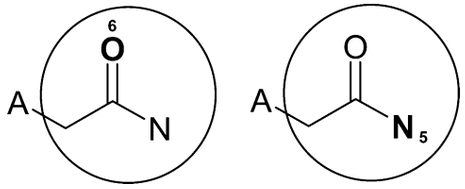


Figure 6. The feature regions after two iterations centered on the oxygen (left) and the nitrogen (right). Since both of these regions contain exactly the same atoms and bonds, they represent duplicate information, even though their hashed identifier values are different.

Hash 값은 다르지만 담겨있는 정보는 동일

- 더 많은 neighbor를 볼 수록 fragment의 종류도 다양해짐

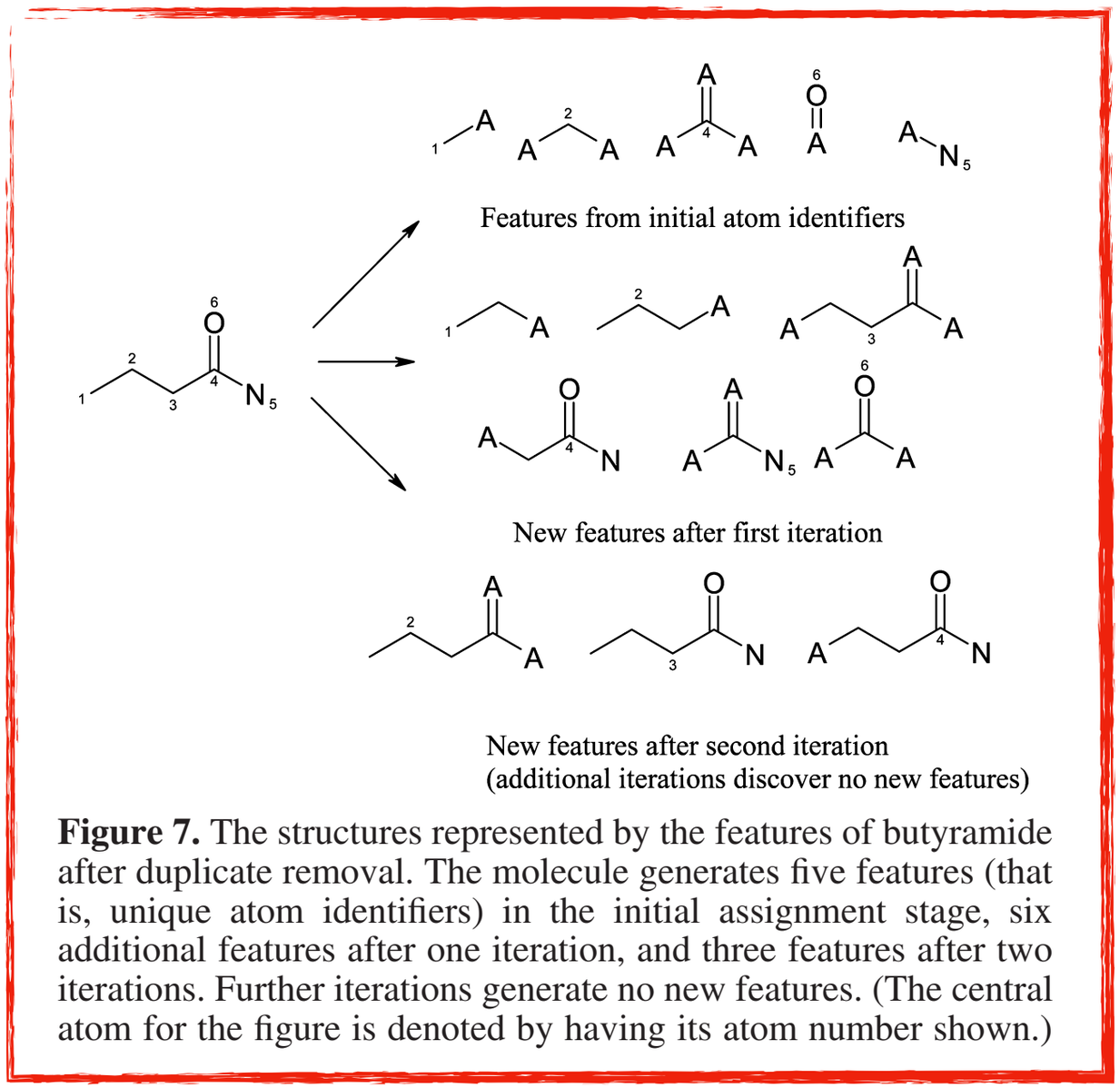
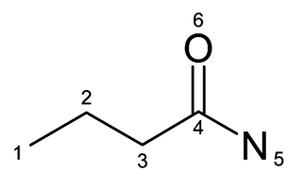


Figure 7. The structures represented by the features of butyramide after duplicate removal. The molecule generates five features (that is, unique atom identifiers) in the initial assignment stage, six additional features after one iteration, and three features after two iterations. Further iterations generate no new features. (The central atom for the figure is denoted by having its atom number shown.)

ECFP를 여러번 반복하면

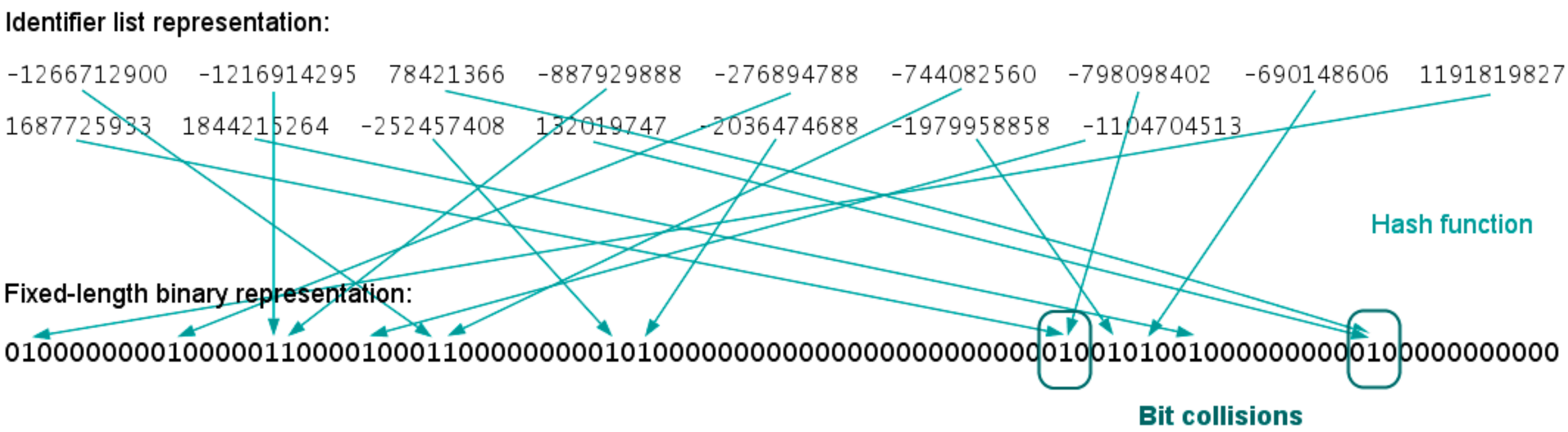


> <ECFP_0>	> <ECFP_2>	> <ECFP_4>	> <ECFP_6>
734603939	734603939	734603939	734603939
1559650422	1559650422	1559650422	1559650422
-1100000244	-1100000244	-1100000244	-1100000244
1572579716	1572579716	1572579716	1572579716
-1074141656	-1074141656	-1074141656	-1074141656
	863188371	863188371	863188371
	-1793471910	-1793471910	-1793471910
	-1789102870	-1789102870	-1789102870
	-1708545601	-1708545601	-1708545601
	-932108170	-932108170	-932108170
	2099970318	2099970318	2099970318
		-87618679	-87618679
		1112638790	1112638790
		-627599602	-627599602

- 앞에서 설명한 방식을 1번 적용한 fingerprint를 ECFP_2라고 부른다.
- 2번 반복하면 ECFP4, 3번 반복하면 ECFP6
- ECFP4와 ECFP6의 fingerprint가 동일한 이유는 새로운 fragment가 나오지 않기 때문

Folding to bits

- 실제로 ECFP를 사용할 때는 검색의 속도를 높이기 위해서 정수의 리스트를 정해진 수의 bit (0또는 1)로 변환한다.
- 이 때도 hash function을 사용한다.
- 일반적으로 1024 bit를 많이 사용한다.



실제 Rdkit에서 사용 예시

Morgan Fingerprints (Circular Fingerprints)

This family of fingerprints, better known as circular fingerprints [5], is built by applying the Morgan algorithm to a set of user-supplied atom invariants. When generating Morgan fingerprints, the radius of the fingerprint must also be provided :

```
>>> from rdkit.Chem import AllChem
>>> m1 = Chem.MolFromSmiles('Cc1ccccc1')
>>> fp1 = AllChem.GetMorganFingerprint(m1,2)
>>> fp1
<rdkit.DataStructs.cDataStructs.UIntSparseIntVect object at 0x...>
>>> m2 = Chem.MolFromSmiles('Cc1nccccc1')
>>> fp2 = AllChem.GetMorganFingerprint(m2,2)
>>> DataStructs.DiceSimilarity(fp1,fp2)
0.55...
```

Morgan fingerprints, like atom pairs and topological torsions, use counts by default, but it's also possible to calculate them as bit vectors:

```
>>> fp1 = AllChem.GetMorganFingerprintAsBitVect(m1,2,nBits=1024)
>>> fp1
<rdkit.DataStructs.cDataStructs.ExplicitBitVect object at 0x...>
>>> fp2 = AllChem.GetMorganFingerprintAsBitVect(m2,2,nBits=1024)
>>> DataStructs.DiceSimilarity(fp1,fp2)
0.51...
```

Radius = 2 (ECFP4)
# of bits = 1024

The default atom invariants use connectivity information similar to those used for the well known ECFP family of fingerprints. Feature-based invariants, similar to those used for the FCFP fingerprints, can also be used. The feature definitions used are defined in the section Feature Definitions Used in the Morgan Fingerprints. At times this can lead to quite different similarity scores:

```
>>> m1 = Chem.MolFromSmiles('c1ccccn1')
>>> m2 = Chem.MolFromSmiles('c1cccо1')
>>> fp1 = AllChem.GetMorganFingerprint(m1,2)
>>> fp2 = AllChem.GetMorganFingerprint(m2,2)
>>> ffp1 = AllChem.GetMorganFingerprint(m1,2,useFeatures=True)
>>> ffp2 = AllChem.GetMorganFingerprint(m2,2,useFeatures=True)
>>> DataStructs.DiceSimilarity(fp1,fp2)
0.36...
>>> DataStructs.DiceSimilarity(ffp1,ffp2)
0.90...
```

Similarity measure

Tanimoto Coefficient

- 두 binary vector (0과 1로 이루어진 벡터)의 유사도를 계산하는 방법
  - a.k.a Jaccard Index
  - 0과 1사이의 값을 가진다.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}.$$

*** Can be calculated with** `rdkit.DataStructs.`**FingerprintSimilarity**

Dice similarity

- Also known as, Sørensen–Dice index, Sørensen index and Dice's coefficient
- $DSC = \frac{2|X \cap Y|}{|X| + |Y|}$
- Can be calculated with `DataStructs.DiceSimilarity` in RDKit