Reservoir Sampling

구현 방법

복원 추출되는 reservoir sampling은 기존에 수업 시간에 구현하였던 reservoir class를 이용하여 구현하였습니다.

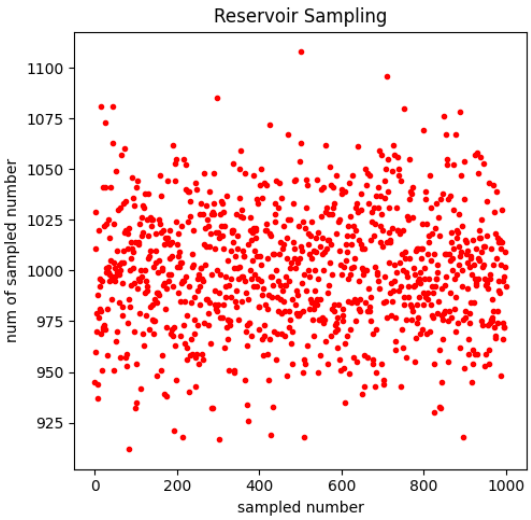
```
class Reservoir:
    def __init__(self, k):
        self.k = k
        self.sampled = []
        self.idx = 0

    def put(self, item):

        if self.idx < self.k:
            self.sampled.append(item)
            self.idx += 1
            return item # 추가

        else:
            r = random.randint(0, self.idx)

            if r < self.k:
                self.sampled[r] = item
                self.idx += 1
                return item # 추가

        self.idx += 1
        return None # 추가
```

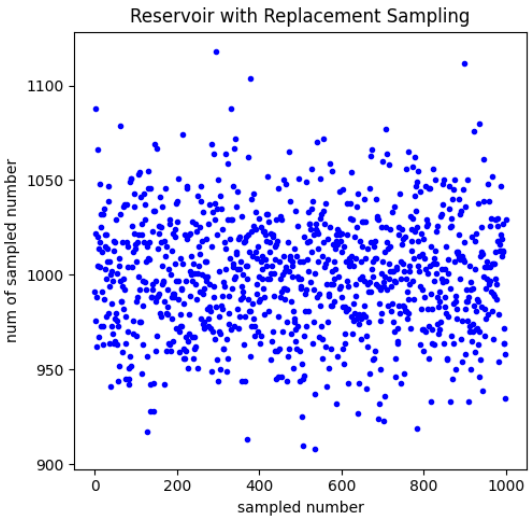
```
class ReservoirWithReplacement:
    def __init__(self, k):
        self.k = k
        self.sampled = [-1 for _ in range(self.k)]
        self.res = [Reservoir(1) for _ in range(self.k)]

    def put(self, item):

        for i in range(self.k): # k=100
            replace = self.res[i].put(item)

            if replace != None: # random을 통해 교체된 경우
                self.sampled[i] = replace
```

복원 추출의 경우 크기가 1인 비복원 추출 reservoir sampling을 k개 진행하여 각각에서 나온 결과를 합쳐 구합니다.

따라서 저는 생성자에 k개의 Reservoir(1) 객체를 가진 res 배열을 만들어 주었고, 각각의 Reservoir(1) 객체에 put(item)을 해주어 해당 item이 샘플링 된 경우, item을 return 하도록 하여 ReservoirWithReplacement의 sampled 배열에서 바뀐 자리의 값만 바꿀 수 있도록 하였습니다.

결과





두 알고리즘 모두, 모든 값이 균등하게 샘플링 되었습니다.

DGIM Algorithm

구현 방법

```
class IntStreamDGIM1:
    def __init__(self):
        self.dgims = [DGIM() for _ in range(4)] # dgims[0]이 2^3자리, dgims[3]이 2^0 자리.

    def put(self, integer):
        bin = '{0:04b}'.format(integer) # 입력 받은 10진수 정수를 4자리 이진수로 바꿔줌(왜냐하면 입력이 0~15)
        for i in range(len(self.dgims)):
            self.dgims[i].put(int(bin[i])) # 이진수의 각 자리에 기존에 구현한 bit stream DGIM 알고리즘 적용

    def count(self, k):
        cnt = 0

        for i in range(len(self.dgims)):
            cnt += self.dgims[i].count(k) * (1 << (3-i)) # 이진수의 각 자리별 1의 개수를 예측하고, 자리 값을 곱하여 줌.

        return cnt
```

첫 번째 방법의 경우, 비트 스트림을 DGIM을 이용하므로, 수업 시간에 구현한 DGIM 클래스를 이용하여 구현하였습니다.

우선 입력이 0이상 15이하의 수이므로, 4개의 비트가 필요하다고 생각하여 생성자에 4개의 DGIM 객체가 들어있는 배열을 만들어주었습니다.

put() 함수에서는 비트 스트림 DGIM을 적용하기 위해 정수를 4자리 이진수로 변환해주었고, 각 자릿수를 기존에 구현한 DGIM.put()을 이용하여 bucket_tower에 넣어주었습니다.

이후 count() 함수에서는 자릿수별 1의 개수를 기존의 함수를 이용하여 구하고, 자릿수 값을 곱해주어 최근 입력된 k개의 정수의 합을 구해주었습니다.

```python
class IntStreamDGIM2:
    def __init__(self):
        self.bucket_tower = [[]] # 버킷의 time stamp를 저장할 배열
        self.bucket_sum = [[]] # 버킷의 부분합을 저장할 배열
        self.ts = 0

    def put(self, integer):
        self.bucket_tower[0].insert(0, Bucket(self.ts, self.ts)) # 입력으로 들어온 버킷 맨 앞에 넣기
        self.bucket_sum[0].insert(0, integer)

        layer = 0
        while len(self.bucket_tower[layer]) > 2:
            if len(self.bucket_tower) < layer+2: # 위 layer가 없는 경우 생성
                self.bucket_tower.append([])
                self.bucket_sum.append([])

            if self.bucket_sum[layer][-1] + self.bucket_sum[layer][-2] > (2 << layer): # 버킷이 3개
                b1 = self.bucket_tower[layer].pop()
                s1 = self.bucket_sum[layer].pop()

                self.bucket_tower[layer+1].insert(0, b1)
                self.bucket_sum[layer+1].insert(0, s1)
            else: # 두 버킷의 합이 다음 layer의 기댓값 이하인 경우 둘 다 올려줌
                b1 = self.bucket_tower[layer].pop()
                b2 = self.bucket_tower[layer].pop()
                s1 = self.bucket_sum[layer].pop()
                s2 = self.bucket_sum[layer].pop()

                b1.end = b2.end # 두 버킷의 time stamp 합쳐줌
                s = s1 + s2 # 두 버킷의 부분합을 합쳐줌


                self.bucket_tower[layer+1].insert(0, b1) # 위 layer에 넣어줌
                self.bucket_sum[layer+1].insert(0, s)
            layer += 1

        self.ts += 1
    def count(self, k):
        s = self.ts - k

        cnt = 0

        for zipped in zip(self.bucket_tower, self.bucket_sum):
            buckets = zipped[0]
            sums = zipped[1]

            for i in range(len(buckets)):
                if s <= buckets[i].start: # 해당 버킷을 다 포함하는 경우
                    cnt += sums[i]
                elif s <= buckets[i].end: # 해당 버킷의 일부만 포함하는 경우
                    cnt += round(sums[i] * (buckets[i].end - s + 1) // (buckets[i].end - buckets[i].start + 1))
                    return cnt
                else:
                    return cnt

        return cnt
```

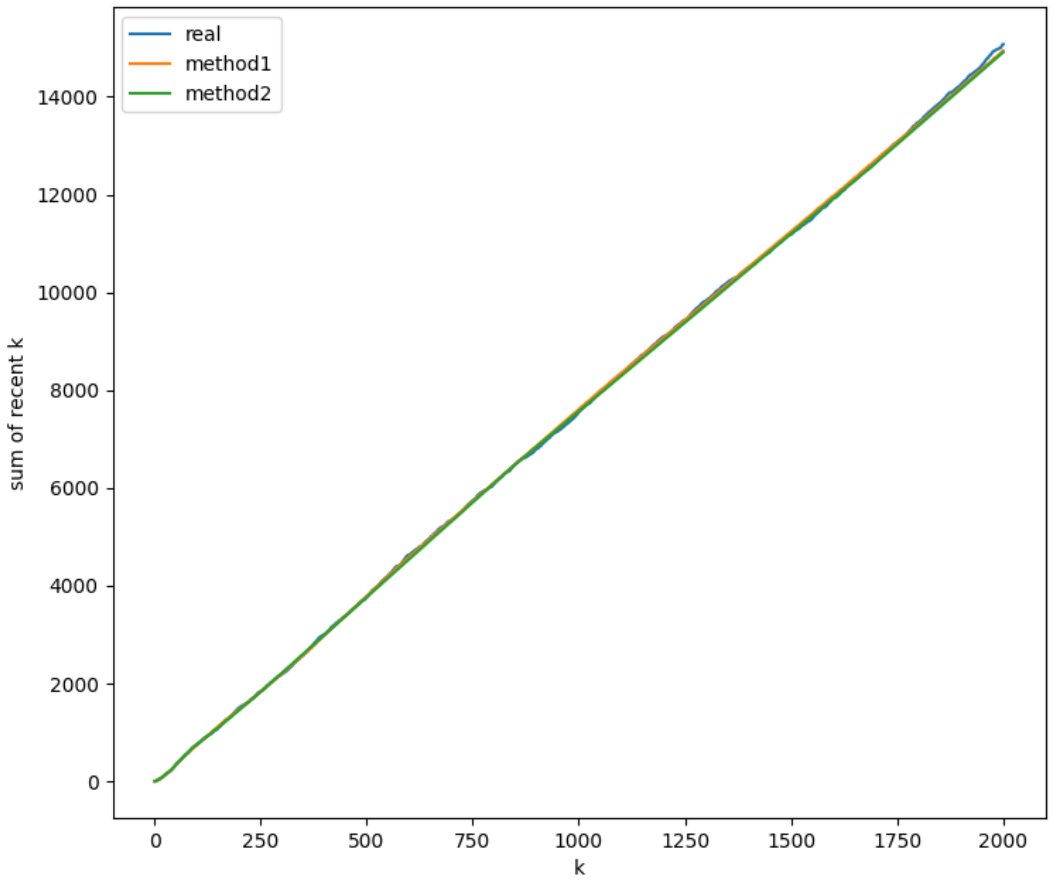
두 번째 방법의 경우, 이전과 다르게 버킷의 부분합을 저장할 배열이 필요해져 만들어주었습니다.

put() 함수에서는 기존과 비슷하지만, 각 layer 내의 값의 합이 2^layer이 되도록 유지하기 위하여 버킷이 3개가 된 layer의 뒤쪽의 두 버킷의 합이 2^(layer+1)보다 큰 경우 맨 뒤의 버킷만을 올려주고, 그렇지 않은 경우 두 버킷 모두 올려주도록 구현하였습니다.

count() 함수에서는 bucket_sum에 각 버킷의 부분합이 저장되어 있기 때문에, 해당 버킷이 k 내

에 포함되는 경우 해당 부분합을 모두 더해주었고, 버킷의 일부만이 포함되는 경우 부분합에 포함되는 비율을 곱하여 예측값 구해 더해주었습니다.

결과



방법1과 방법2 모두 잘 예측하지만, 방법1의 경우 조금 더 스트림의 영향을 받지 않고 안정적으로 예측하는 것 같습니다. 이는 모든 자리수에서 DGIM 알고리즘을 적용하기 때문에 조금 더 스트림의 분포의 영향을 덜 받는다고 생각합니다.