NV Sieve on Quantum (1)

https://youtu.be/yH0lKFV9F3c

NV Sieve 동작 과정

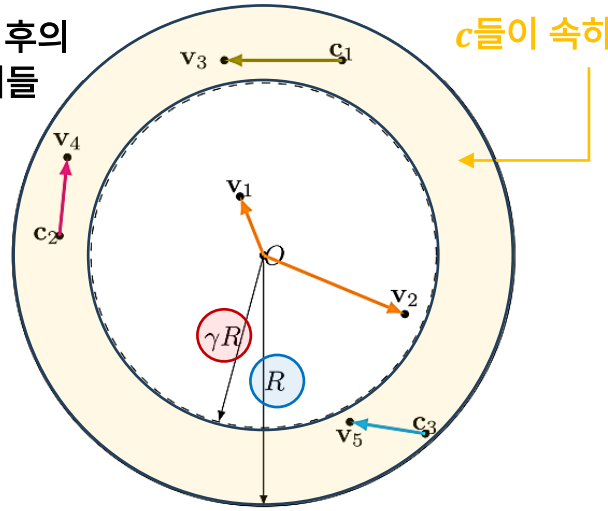
Algorithm 5 The lattice sieve

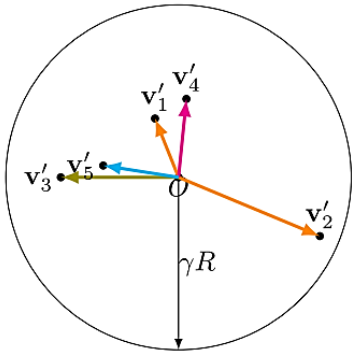
Input: A subset $S \subseteq B_n(R)$ of vectors in a lattice L and a sieve factor $2/3 < \gamma < 1$.
Output: A subset $S' \subseteq B_n(\gamma R) \cap L$.

1: $R \leftarrow \max_{\mathbf{v} \in S} \|\mathbf{v}\|$
2: $C \leftarrow \emptyset$, $S' \leftarrow \emptyset$
3: **for** $\mathbf{v} \in S$ **do**
4: **if** $\|\mathbf{v}\| \leq \gamma R$ **then**
5: $S' \leftarrow S' \cup \{\mathbf{v}\}$
6: **else**
7: **if** $\exists \mathbf{c} \in C \ \|\mathbf{v} - \mathbf{c}\| \leq \gamma R$ **then**
8: $S' \leftarrow S' \cup \{\mathbf{v} - \mathbf{c}\}$
9: **else**
10: $C \leftarrow C \cup \{\mathbf{v}\}$
11: **end if**
12: **end if**
13: **end for**
14: **return** S'

$v - c$ 후의 벡터들

c들이 속하는 공간





$v - c$ 들을 원점으로 이동해보면?
범위 안에 속함

- **목적** : 짧은 벡터에 대한 손실이 없게 하기 위해 c를 랜덤으로 선택하여 범위를 줄여 나가며 γR 보다 짧은 벡터 얻기
- **입력** : 최대 길이가 R인 격자 상의 벡터
- **출력** : v보다 짧은 격자 상의 벡터
- **용어**
  - $B_n(R)$: 원점으로부터의 길이가 R보다 작은 격자 상의 벡터
  - S' : 범위 내의 벡터들을 저장
  - c : $\gamma R \leq x \leq R$에 속하는 충분한 수의 격자 상의 점
- **동작 과정**
  1. S 에 속한 벡터들 중 최대 길이 → R
     C, S' 초기화
  2. γR보다 길이가 짧은 벡터들은 S'에 저장
  3. γR 보다 길이가 긴 벡터들은 c라는 포인트와 뺄셈 한 후,
     그 길이가 γR보다 짧으면 S'에 저장 / 길면 C에 저장
  4. S' 반환 (γR보다 길이가 짧은 벡터들)
- 해당 과정은 **알고리즘 4의 line 9에 해당되므로 반복**
  - 반복적으로 수행하여 충분히 짧은 벡터 집합들을 얻고, 그 중에서 가장 짧은 벡터를 찾아냄

NV Sieve 중요 포인트

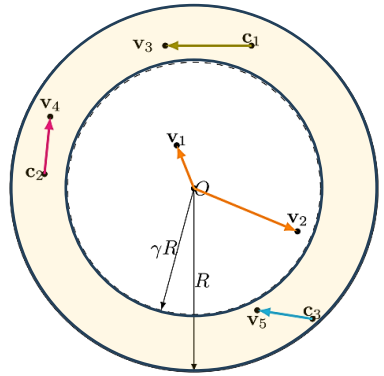
- **알고리즘 복잡도에 영향을 미치는 결정적 부분**
  - **c의 포인트 수를 측정하는 부분**
    - 충분히 많은 포인트들이 있고 이를 이용하여
      γR보다 짧은 길이의 벡터를 만들 수 있는 점을 찾아야 함
  - **처음 주어진 서브셋 S의 크기가 클수록 좋지 않음**
    - S의 rank가 커지면 c의 개수도 많아짐
      - **c도 격자 상의 벡터이고, S의 부분 집합이므로**
    - 해당 부분이 c의 포인트 수를 측정하는 것에 영향을 주므로 전체적인 Sieve 알고리즘 퀄리티에 중요

- **실제 구현**
  - 실제 구현 시, rank를 약 50으로 설정
    $(30 \leq \text{rank} \leq 48)$
  - **$\gamma = 0.97$** 사용
  - **LLL 적용** 후 축소된 **8000개의 벡터를 입력**으로 사용
  - 48 rank까지 **700 MB RAM** 사용
    (많은 메모리 사용되지 않음)
  - C++의 long type vector 사용
    (64-bit OS에서 64-bit)

  - 구현 관련 부분은 추후 더 자세히 알아볼 예정

NV Sieve on Grover

- **적용 지점**
  - 우선, 고차원에서 벡터를 걸러내는 LLL은 classical로 구현
  - **Sieve (알고리즘 5)에서 c를 활용하여 조건을 만족하는 짧은 벡터를 찾기 위한 과정에 Grover 적용**
  - **c는 격자 상의 포인트 (노랑)**, 이를 **중첩상태로 준비**하여 γR보다 짧은 길이의 벡터를 만들 수 있는 점을 찾아냄
    → 이후, 뺄셈 연산하여 조건 만족하는지 확인까지가 Grover 적용 범위



- **입력 및 오라클**
  - **입력 :** c, v (중첩 상태)
  - **오라클 :** $v - c \leq \gamma R$ **(**범위 내에 속하는지 비교)

- **기대 효과**
  - 알고리즘의 시간 복잡도 감소 (공간 복잡도 감소 없음)*
    - $\log_2$(time) : 0.415 → 0.312
    - $\log_2$(space) : 0.2075 → 0.2075

- 그러나, **이론적 계산에 의한 결과**이며, 실험에 의한 분석 및 평가는 없음

*Laarhoven, Thijs, Michele Mosca, and Joop Van De Pol. "Finding shortest lattice vectors faster using quantum search." Designs, Codes and Cryptography 77 (2015): 375-400.

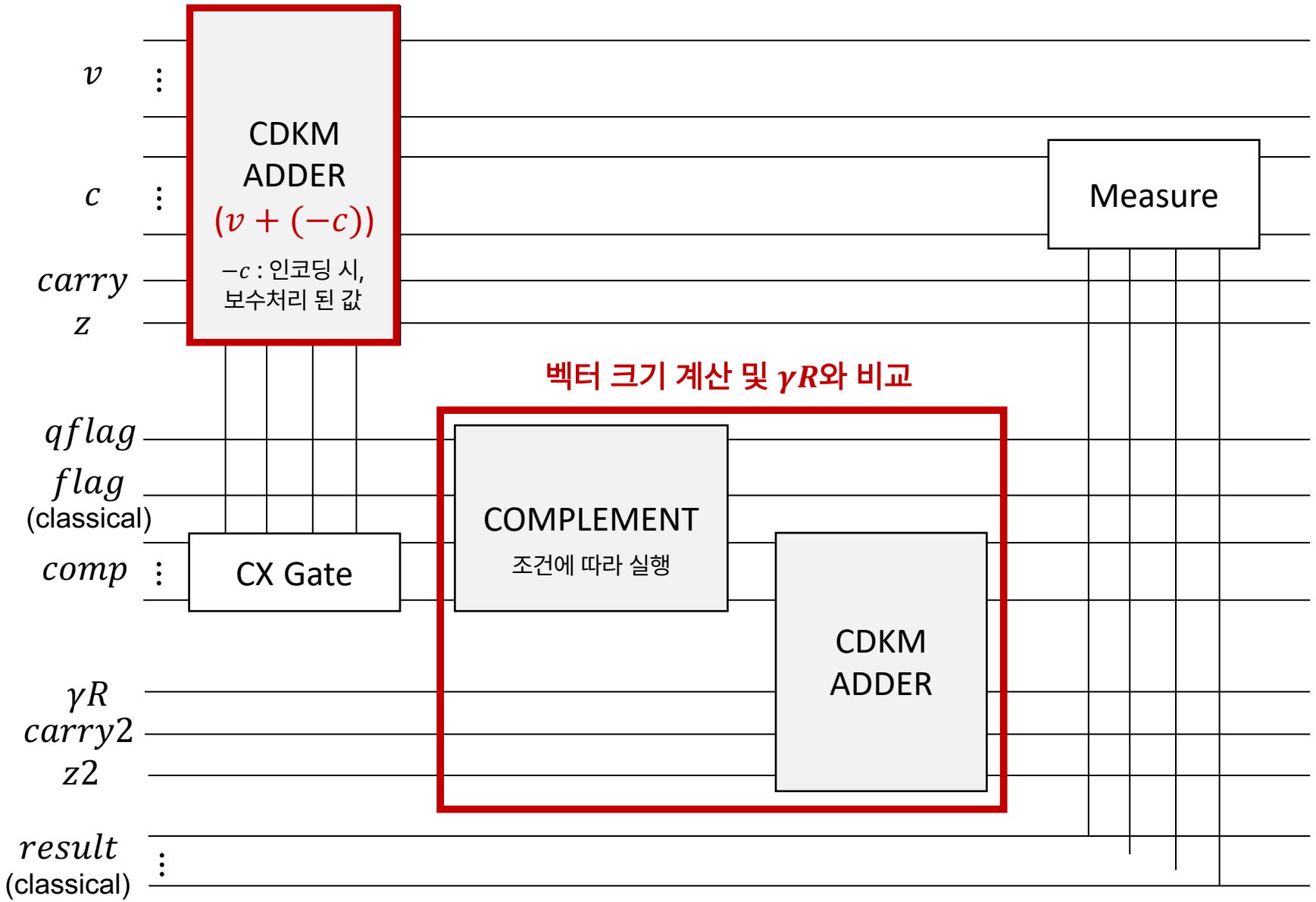
필요 함수

- Classical
  - $rR > v$
    - True : v는 짧은 벡터 리스트에 포함
    - False : $v - c$ 통해 짧은 벡터 찾음 (→ Quantum 으로 넘어감)

- Quantum (Oracle)
  - v, c : 중첩 상태로 준비
  - $v - c$: 뺄셈
  - $||v - c||$: rR과의 크기 비교를 위한 절대값 연산
  - $rR > ||v - c||$ → $rR - ||v - c|| > 0$
    - True : $v - c$는 짧은 벡터 리스트에 포함 (classical)
    - False : 버려짐

- **두 번의 뺄셈 (두 포인트 간의 뺄셈, 비교 연산), 크기 비교 위한 절대값 연산 필요**
  → 여기서 **불필요한 자원이나 연산이 생겨서, 조건문 사용하여 절대값 및 비교 연산을 합치고** 덧셈기를 그대로 사용

NV Sieve 양자 회로 (초기버전..)



CDKM 덧셈기

- Ripple carry adder 개선
- In-place 방식의 덧셈기
  - 결과 큐비트를 따로 사용하지 않고 연산된 값은 operand_b에 저장됨
- 구현물이 있어서 받아서 사용
  - Sieve 완성이 목표라서 덧셈기 자체에 대한 공부는 아직 진행하지 않았습니다.
  - 추후 최적화 필요하거나 하면 하도록 하겠습니다.

```
def CDKM(circuit1, operand_a, operand_b, operand_c, operand_z, n):
    for i in range(1, n):
        circuit1.cx(operand_a[i], operand_b[i])

    circuit1.cx(operand_a[1], operand_c)
    circuit1.ccx(operand_a[0], operand_b[0], operand_c)
    circuit1.cx(operand_a[2], operand_a[1])
    circuit1.ccx(operand_c, operand_b[1], operand_a[1])
    circuit1.cx(operand_a[3], operand_a[2])

    for i in range(2, n-2):
        circuit1.ccx(operand_a[i-1], operand_b[i], operand_a[i])
        circuit1.cx(operand_a[i+2], operand_a[i+1])

    circuit1.ccx(operand_a[n-3], operand_b[n-2], operand_a[n-2])
    circuit1.cx(operand_a[n-1], operand_z)
    circuit1.ccx(operand_a[n-2], operand_b[n-1], operand_z)

    for i in range(1, n-1):
        circuit1.x(operand_b[i])

    circuit1.cx(operand_c, operand_b[1])

    for i in range(2, n):
        circuit1.cx(operand_a[i-1], operand_b[i])

    circuit1.ccx(operand_a[n-3], operand_b[n-2], operand_a[n-2])

    for i in range(n-3, 1, -1):
        circuit1.ccx(operand_a[i-1], operand_b[i], operand_a[i])
        circuit1.cx(operand_a[i+2], operand_a[i+1])
        circuit1.x(operand_b[i+1])

    circuit1.ccx(operand_c, operand_b[1], operand_a[1])
    circuit1.cx(operand_a[3], operand_a[2])
    circuit1.x(operand_b[2])

    circuit1.ccx(operand_a[0], operand_b[0], operand_c)
    circuit1.cx(operand_a[2], operand_a[1])
    circuit1.x(operand_b[1])

    circuit1.cx(operand_a[1], operand_c)

    for i in range(0, n):
        circuit1.cx(operand_a[i], operand_b[i])

    circuit1.barrier()
```

2의 보수

- Classical에서의 2의 보수
  - **1의 보수 → LSB +1**

- **Quantum에서의 2의 보수**
  - 1의 보수 : 모든 비트에 X gate (NOT 연산)
  - LSB+1 : 마지막으로 1 더함
    - 큐비트 상태가 모두 1인 경우, Ripple carry 사용하여 LSB flip

```
def COMPLEMENT(circuit1, comp, qflag, flag, n):

    circuit1.cx(comp[n-1], qflag)
    circuit1.measure(qflag, flag)

    with circuit1.if_test((flag, 0)):
        [circuit1.x(comp[i]) for i in range(n)]

        # Add 1 to LSB
        for i in range(n-1, -1, -1):
            if i > 0:
                cnx_gate = XGate().control(i)
                circuit1.append(cnx_gate, comp[0:i] + [comp[i]])
        circuit1.x(comp[0])
```

입력 및 회로 설정

- 큐비트 및 회로, 입력 값 설정
- 실제로는 다음과 같은 **정수 벡터** 입력 (벡터의 방향성으로 인해 부호 존재)
- 현재 버전에서는 **일단 4 비트 입력 사용 (-8~+7)**
- 최적화하지 않은 코드이므로 큐비트의 수가 많이 사용되었을 수도 있음

experimental_sieve / tests / lattices / example_svp_in

```
[[-24 -31 -340 654 407 -428 113 31 112 -166 179 -185 -575 64 -121 -189 -55 44 655 57 23 ]
[-107 99 -435 414 -480 103 184 -35 110 -145 -104 436 -294 484 160 -754 22 -289 526 -271 5 ]
[-203 -672 262 688 159 -432 293 -10 86 -136 -266 -148 100 -660 42 34 260 618 293 270 6 ]
[163 -436 -206 91 80 -173 814 597 499 -633 -153 -254 -97 -184 62 -20 253 89 75 -117 -133 ]
[-106 174 100 -356 -191 -613 -300 -48 89 437 304 567 -157 560 -186 35 -356 -305 -472 427 54 ]
[213 -468 407 164 302 594 -298 182 -199 -245 27 106 668 -29 -233 23 -118 -154 219 -328 -152 ]
[-135 -187 569 31 -146 -463 425 100 68 460 -344 214 203 630 240 -344 66 -512 -130 -372 148 ]
[60 -290 916 -32 66 -570 -84 -216 233 -203 -357 88 624 162 95 926 118 -172 -243 -51 -128 ]
[-243 74 -594 -14 534 -294 364 156 -105 437 23 414 -410 -81 -431 -70 -150 -617 -214 127 -88 ]
[-366 -23 -185 157 720 -236 82 550 194 -595 -213 -107 243 527 193 -501 -281 -599 25 -140 -18 ]
[157 -408 222 -145 -544 358 146 -253 -34 560 -337 290 68 313 -141 -344 -30 -103 496 509 -458 ]
[68 -72 290 -97 -42 -121 -616 -201 196 376 -322 308 343 -212 287 463 415 88 -209 393 -840 ]
[89 10 233 57 -378 -615 108 37 -70 526 438 231 464 386 -184 71 -637 175 183 -512 396 ]
[-62 179 98 -240 -110 -589 272 784 112 1 -100 256 -116 -581 630 -705 124 -422 -227 -172 328 ]
[150 64 220 -261 796 -248 191 113 -399 160 330 -57 -538 -438 298 295 548 146 536 -371 286 ]
[276 728 78 73 280 250 -665 -262 285 -367 475 118 535 74 65 -89 -569 -1 -392 154 -628 ]
[317 154 -6 199 63 -427 -274 -146 -430 167 -35 456 74 130 335 -674 592 -144 236 495 105 ]
[-590 -345 -384 346 918 118 -19 -144 -636 144 405 -522 250 -54 180 132 162 -525 72 -213 403 ]
[-152 563 187 131 118 -156 -209 -337 354 38 81 -64 682 650 -75 -519 569 181 352 -417 21 ]
[-34 -243 235 331 663 -424 223 -339 476 -659 -719 -503 -498 -310 622 208 -182 511 -8 -317 743 ]
]
```

```
a = QuantumRegister(4) # a
b = QuantumRegister(4) # b

c = QuantumRegister(1) # carry qubit
c2 = QuantumRegister(1)

z = QuantumRegister(1)
z2 = QuantumRegister(1)

comp = QuantumRegister(4)

rR = QuantumRegister(4)

qflag = QuantumRegister(1)
flag = ClassicalRegister(1)

res_comp = ClassicalRegister(5) # abs


circuit1 = QuantumCircuit(a, b, c, c2, z, z2, comp, rR, qflag, flag, res_comp)

bit_size = 4
```

```
#circuit1.x(a[0])  # LSB
#circuit1.x(a[1])
circuit1.x(a[2])
#circuit1.x(a[3]) # MSB

#circuit1.x(b[0])  # LSB
circuit1.x(b[1])
circuit1.x(b[2])
circuit1.x(b[3])  # MSB

circuit1.x(rR[0])  # LSB
circuit1.x(rR[1])
circuit1.x(rR[2])
#circuit.x(rR[3])  # MSB
```

NV Sieve 로직 구현

- 다음과 같이 간소화
  - 원래 : 뺄셈 → 절대값 → 크기비교 (뺄셈)
  - 현재 : c의 2의 보수 (classical) → v, c 덧셈 (CDKM) → 조건문에 따라 2의 보수 취한 후, 덧셈 통한 크기 비교
- If문 사용하여 $v + (-c)$의 결과 (res)가 양수이면, 2의 보수 취함
  - rR 과의 크기 비교를 위해 rR -||res||를 해야 함
    → 여기서, **res가 음수이면 절대값 처리 필요 없음** (rR + (res) 자체가 뺄셈) ; ex : 7+(-6)
    → **res가 양수이면 벡터의 크기와 동일**하므로 뺄셈을 해야함 → 조건문 통해 최상위 비트가 0 (양수)이면 보수 처리하여 음수로 변경

```
############## ADD ############
CDKM(circuit1, a, b, c, z, bit_size)

############## Complement ##############
# 덧셈 결과 b가 양수면 보수취하고, 아니면 바로 덧셈으로 넘어가
for i in range(0, bit_size):
    circuit1.cx(b[i], comp[i])

COMPLEMENT(circuit1, comp, qflag, flag, bit_size)
############## 끝됴 ##############

############## ADD ############
CDKM(circuit1, rR, comp, c2, z2, bit_size)

circuit1.measure(comp[0], res_comp[0])
circuit1.measure(comp[1], res_comp[1])
circuit1.measure(comp[2], res_comp[2])
circuit1.measure(comp[3], res_comp[3])
circuit1.measure(z2, res_comp[4])

circuit1.measure(b[0], res[0])
circuit1.measure(b[1], res[1])
circuit1.measure(b[2], res[2])
circuit1.measure(b[3], res[3])
circuit1.measure(z, res[4])
```

```
def COMPLEMENT(circuit1, comp, qflag, flag, n):

    circuit1.cx(comp[n-1], qflag)
    circuit1.measure(qflag, flag)

    with circuit1.if_test((flag, 0)):
        [circuit1.x(comp[i]) for i in range(n)]

        # Add 1 to LSB
        for i in range(n-1, -1, -1):
            if i > 0:
                cnx_gate = XGate().control(i)
                circuit1.append(cnx_gate, comp[0:i] + [comp[i]])
        circuit1.x(comp[0])
```

Flag 큐비트 및 클래식비트를 사용하여 조건문 사용 (측정 때문)

회로 실행

- Qiskit 시뮬레이터 사용
- Qubit 21, Depth 144
- $v = 4, c = -2$ (보수 처리 된), $rR = 7$인 경우
  - 덧셈 결과 10010에서 최상위 캐리 값 무시하면, 0010 → 2 (= 4+(-2)) → 보수 취하는 조건문으로 감
  - $rR - (||v - c||)$ = 10101에서 최상위 값 무시하면, 0101 → 5 (=7+(-2))
  - 즉, **결과 값이 양수**이므로, rR가 $v - c$ 보다 더 큼 → shortest vector 리스트에 큐비트 b의 값을 측정하여 추가 `CDKM(circuit1, a, b, c, z, bit_size)`

중간 결과 확인용

```
################### Execute ####################

statevector_simulator = AerSimulator(method='statevector')

# Transpile circuit for backend
tcirc = transpile(circuit1, statevector_simulator)

# Try and run circuit
result = statevector_simulator.run(tcirc, shots=5).result()
counts = result.get_counts(circuit1)

circuit1 = circuit1.decompose()
print("Decomposed depth : ", circuit1.depth())
print(counts)

#circuit1.draw(output='mpl')
```

```
Decomposed depth :  145
{'10101 10010 0': 5}
```

덧셈 결과

$rR - (||v - c||)$

실제 결과

```
################### Execute ####################

statevector_simulator = AerSimulator(method='statevector')

# Transpile circuit for backend
tcirc = transpile(circuit1, statevector_simulator)

# Try and run circuit
result = statevector_simulator.run(tcirc, shots=5).result()
counts = result.get_counts(circuit1)

circuit1 = circuit1.decompose()
print("Decomposed depth : ", circuit1.depth())
print(counts)

#circuit1.draw(output='mpl')
```

```
Decomposed depth :  144
{'10101 0': 5}
```

$rR - (||v - c||)$

향후 계획

- 양자회로에서는 고정 소수점이 더욱 효율적이라 이 방식에 대해 공부하고 적용할 예정
  (https://arxiv.org/pdf/1805.12445.pdf)
  → 이렇게 하려면 v, c 등 다른 벡터 값들의 자료형도 바꾸어야 할 것으로 보임

- 그러나, 현재 QRAM을 사용하지 않고 입력을 고정해둔 상태
  → 즉, 그루버 돌릴 게 없음 / NV Sieve 로직만 구현 중
  → 코드 구할 수 있는지 물어볼 예정

감사합니다.