

This handout walks through two ways to compute LSH hash keys directly from the original vector, without constructing the full unary-encoded vector.

## 1 Example Setup

Consider a two-dimensional vector with maximum coordinate value  $C = 5$ :

$$p = (3, 2)$$

Its unary encoding is

$$p = (3, 2) \longrightarrow \underbrace{11100}_{\text{Coordinate 1}} \underbrace{11000}_{\text{Coordinate 2}}$$

The encoded vector contains  $dC = 2 \times 5 = 10$  bits.

Suppose an LSH hash function selects the following positions using 1-based indexing:

$$I = \{2, 4, 5, 6, 8, 10\}$$

| Position | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|----------|---|---|---|---|---|---|---|---|---|----|
| Bit      | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0  |

Therefore, the expected hash key is

$$g(p) = 100100$$

We now compute this key directly from  $p = (3, 2)$ .

## 2 Mapping Selected Positions to Coordinates

Before moving on to the optimized methods, we need to figure out which original coordinate each selected bit belongs to and where that bit is located within the coordinate.

Since each coordinate occupies  $C = 5$  bits, a global position  $j$  maps to coordinate  $i$  and local position  $t$  by

$$i = \left\lceil \frac{j}{C} \right\rceil, \quad t = ((j - 1) \bmod C) + 1$$

For example,  $j = 8$  corresponds to coordinate  $i = 2$  and local position  $t = 3$ .

Applying this mapping to get

| Coordinate | Value | Selected Local Positions |
|------------|-------|--------------------------|
| 1          | 3     | [2, 4, 5]                |
| 2          | 2     | [1, 3, 5]                |

The unary encoding contains  $p_i$  ones followed by  $C - p_i$  zeros. Thus, a selected bit can be computed directly by

$$b(i, t) = \begin{cases} 1, & t \leq p_i, \\ 0, & t > p_i. \end{cases}$$

### 3 Approach 1: Binary Search

Because the selected local positions are sorted, the binary search can locate the boundary between ones and zeros.

#### Coordinate 1

For  $p_1 = 3$ , the selected positions are [2, 4, 5].

1. Check the middle position 4:  $4 > 3$ , so the position 4 and all later positions are zeros.
2. Check position 2:  $2 \leq 3$ , so position 2 is one.

Therefore,

$$g_1(p) = 100$$

#### Coordinate 2

For  $p_2 = 2$ , the selected positions are [1, 3, 5].

1. Check the middle position 3:  $3 > 2$ , so the position 3 and all later positions are zeros.
2. Check position 1:  $1 \leq 2$ , so position 1 is one.

Therefore,

$$g_2(p) = 100$$

Combining both coordinates gives the following.

$$g(p) = g_1(p) \| g_2(p) = 100100$$

## Time Complexity

Each coordinate has at most  $C$  selected positions. The binary search takes  $O(\log C)$  comparisons per coordinate. Across  $d$  coordinates, determining the boundaries takes

$$O(d \log C)$$

## 4 Approach 2: Precomputed Lookup

Instead of performing a binary search for each vector, we precompute the number of selected ones for every possible coordinate value.

### Lookup Tables

| Value | Coordinate 1: $[2, 4, 5]$ |      | Coordinate 2: $[1, 3, 5]$ |      |
|-------|---------------------------|------|---------------------------|------|
|       | # Ones                    | Bits | # Ones                    | Bits |
| 0     | 0                         | 000  | 0                         | 000  |
| 1     | 0                         | 000  | 1                         | 100  |
| 2     | 1                         | 100  | 1                         | 100  |
| 3     | 1                         | 100  | 2                         | 110  |
| 4     | 2                         | 110  | 2                         | 110  |
| 5     | 3                         | 111  | 3                         | 111  |

For  $p = (3, 2)$ :

$$p_1 = 3 \longrightarrow 1 \text{ selected one} \longrightarrow g_1(p) = 100$$

$$p_2 = 2 \longrightarrow 1 \text{ selected one} \longrightarrow g_2(p) = 100$$

Therefore:

$$g(p) = g_1(p) \| g_2(p) = 100100$$

## Time Complexity

Each coordinate requires one constant-time lookup. Across  $d$  coordinates:

$$O(d)$$