1 Quasi-Succinct Indexing

1.1 Index Compression

- Instantaneous codes – storage of integers is proportional to size of integer
  - smaller numbers use fewer bits
- Gap encoding – turns lists of increasing numbers into lists of smaller integers
  - smaller integers being the gaps between successive values

1.2 Brief Overview of Unary-Code

Unary code represents a natural number n as

- n 1's followed by a 0 – for n *non-negative*, or
- $n-1$ 1's followed by a 0 – for n *strictly positive*.

We can exchanges 0's and 1's without loss of generality. The flipped version is often called negated unary code. Example: $5 \to$ `111110` or `000001`

1.3 Representation of a Monotone Sequence

Suppose we have the following monotone sequence for $x_i \in \mathbb{N}$:

$$x_0 \leq x_1 \leq \ldots \leq x_{n-1} \leq u,$$

where u is some upper bound. A question to consider: what is the most effective way to choose u? Two possible options are to choose u s.t. (1) $\log u \in \mathbb{N}$, or (2) $u = x_{n-1} + 1$.

- We store the sequence in a "high/low bit representation" using two *bit-arrays*.
  - The lower $l = \max\{0, \lfloor \log \frac{u}{n} \rfloor\}$ bits of each x_i are stored *explicitly* and *contiguously* in a **lower-bits array**.
  - The remaining upper-bits are stored as a sequence of *unary code gaps* in an **upper-bits array**.
    * difference in upper bits $= \lfloor \frac{x_i}{2^l} \rfloor - \lfloor \frac{x_{i-1}}{2^l} \rfloor$.
    * For the sake of completeness, let $x_{-1} = 0$.
- This representation uses at most $2 + \lceil \log \frac{u}{n} \rceil$ bits **per element** meaning that this representation is **quasi-succinct**.

Proof. Each unary encoding uses 1 stop bit, and each other written bit increases the value of the upper bits by 2^l. This cannot happen more than $\lfloor \frac{x_{n-1}}{2^l} \rfloor$ times. This leads us to the following:

$$\left\lfloor \frac{x_{n-1}}{2^l} \right\rfloor \leq \left\lfloor \frac{u}{2^l} \right\rfloor \leq \frac{u}{2^l} \leq 2n.$$

Unless $\frac{u}{n}$ is a power of 2, we have

$$\left\lceil \log \frac{u}{n} \right\rceil = \left\lfloor \log \frac{u}{n} \right\rfloor + 1.$$

If $\frac{u}{n}$ is a power of 2, then

$$\left\lceil \log \frac{u}{n} \right\rceil = \log \frac{u}{n},$$

but the previous equation is bounded by n instead of $2n$. The informational-theorectial limit is

$$\left\lceil \log \binom{u+n}{n} \right\rceil \approx n \log \left(\frac{u+n}{n} \right),$$

so we conclude that this representation is close to succinct. ■

1.4 Example

Suppose we have a list `[5, 8, 8, 15, 32]` with $u = 36$. We compute $l = \left\lfloor \log \frac{36}{5} \right\rfloor = 2$. Split the lower l bits of the binary representation of each element of the list.

```
     5             8             8            15              32
+---+----+  +----+----+  +----+----+  +----+----+  +------+----+
| 1 | 01 |  | 10 | 00 |  | 10 | 00 |  | 11 | 11 |  | 1000 | 00 |
+---+----+  +----+----+  +----+----+  +----+----+  +------+----+
```

The lower-bits array:

```
+----+----+----+----+----+
| 01 | 00 | 00 | 11 | 00 |
+----+----+----+----+----+
```

To compute the upper-bits array, we find the difference between each of the upper bits and then convert to unary code.

```
     5             8             8            15              32
+---+----+  +----+----+  +----+----+  +----+----+  +------+----+
| 1 | 01 |  | 10 | 00 |  | 10 | 00 |  | 11 | 11 |  | 1000 | 00 |
+---+----+  +----+----+  +----+----+  +----+----+  +------+----+
  |            |            |            |             |
  |            |            |            |             |
  v            v            v            v             v
  1            2            2            3             8
```

Recall that we choose $x_{-1} = 0$ for consistency. The upper-bits array:

```
  1      1    0    1       5
+----+----+---+----+--------+
| 01 | 01 | 1 | 01 | 000001 |
+----+----+---+----+--------+
```

1.5 Recovering x_i

- Perform i unary code reads in the upper-bits array to p^{th} bit position
  - The value of the upper bits is exactly $p - i$
  - Example: To read the upper bits of 15, we perform 4 unary code reads to position 7 of the upper-bits array. Thus, the value of the upper bits of 15 is $7 - 4 = 3$.
- Extract lower bits with random access at position il in the lower-bits array

1.6 Some Optimizations for Accessing

1.6.1 Forward Pointers

- Choose a *quantum* q.
- Store **forward pointers** at positions that we would reach after kq unary code reads.
  - This is the position immediately after $kq - 1$ bits.
- Retrieve x_i by simulatinging $q \left\lfloor \frac{i}{q} \right\rfloor$ reads then $i \mod q$ sequential reads.
- On average, recovering x_i takes constant time.
- Memory wise, recovery takes at most $3q$ bits.
- A smaller q takes less reads, but requires more space.

1.6.2 Skip Pointers

- Store positions reached at negated unary code bits.
  - In other words, store pointers to the positions of each `1` in the upper-bits array.

A similar analysis of time and space complexity can be done, but I don't really feel like doing that right now. This method may not even be that helpful for what I want.

1.7 Quasi-Succinct Bitstream

1.7.1 Bitstream Layout

The desired Bitstream Layout is

```
+----------+-----+-----+---------+-----+-----+--------+-------+
| metadata | p_0 | ... | p_{s-1} | l_0 | ... | l_{n-1} | u ... |
+----------+-----+-----+---------+-----+-----+--------+-------+
                        Bitstream Layout
```

- n: number of elements
- u: upper bound of x_i
- $l = \max\{0, \lfloor \log \frac{u}{n} \rfloor\}$: number of bits of each element stored in lower-bits array
- q: arbitrary quantum
- $m = n + \lfloor \frac{u}{2^l} \rfloor$: maximum length of the *upper-bits array*
- $w = \lceil \log(m+1) \rceil$: width of the pointers
  - If u is not known, then the metadata must contain data to compute w.
- $s = \left\lfloor \frac{n}{q} \right\rfloor$: number of forward pointers
  - $s \leq \left\lfloor \frac{m}{q} \right\rfloor$ for skip pointers that point to negated unary codes
- $d + sw$: beginning position of lower-bits array
- $sw + nl + d$: beginning position of upper-bits array

1.7.2 Information in the Metadata

We want to store n, l, q, and w in the metadata in some order. Order does not matter, as long as, it is consistent and clearly marked in the implementation.