Example: Fixed Density Binary Strings

- We want to encode a length $n$ binary string that we know has $k$ ones.

- What kind of redundancy is there in the data?

- How can we use that to encode the data?

Thanks to Janine Tiefenbruck.
Example: Fixed Density Binary Strings

- We want to encode a length $n$ binary string that we know has $k$ ones.

- How many bits are required?

- Can we recover the original string?
Compressing Fixed Density Strings

- Goal: Find a *better* compression algorithm for fixed-density binary strings. This should use *less than* \( n-1 \) bits.

- Idea for an algorithm: Give the positions of the ones in the string, not within the whole string, but within some smaller window.
Compressing Fixed Density Strings

A possible compression algorithm:

- Only look ahead a few positions in the string. This is a fixed window size.
- If there is a one, record its position within the window, then move up the window.
- If there are no ones, record a 0 and move up the window.
Example

- Let $n = 12$, $k = 3$.
- We will use a window size of $n/k = 4$.
- Say we want to encode the string
  
  $s = 011000000010$
Example

\[ s = \underline{011000000010} \quad \leftarrow \text{one in the window in position 1} \]

- output = 01
  
  01 – one in position 1 in the window
Example

\[ s = 011000000010 \]

- one in the window in position 0

\[ \begin{array}{c}
0 \\
1 \\
2 \\
3 \\
\end{array} \]

- output = 0100
- 01 – one in position 1 in the window
- 00 – one in position 0 in the window
Example

\[ s = 011\overline{0000000010} \quad \leftarrow \text{no ones in the window} \]

\[
\begin{array}{c|c}
\text{0} & \text{1} & \text{2} & \text{3} \\
\hline
0 & 1 & 2 & 3 \\
\end{array}
\]

- output = 01000
  - 01 – one in position 1 in the window
  - 00 – one in position 0 in the window
  - 0 – no ones in the window
Example

\[ s = \overline{011000000010} \quad \leftarrow \text{one in the window in position 3} \]

- output = 0100011
  
  01 – one in position 1 in the window
  00 – one in position 0 in the window
  0 – no ones in the window
  11 – one in position 3 in the window
Example

$s = \overline{011000000010} \quad \leftarrow \text{no ones in the window}$

- output = 01000110
  - 01 – one in position 1 in the window
  - 00 – one in position 0 in the window
  - 0 – no ones in the window
  - 11 – one in position 3 in the window
  - 0 – no ones in the window
Is this a valid compression algorithm?

- Let $n = 12$, $k = 3$. 
  
  $011000000010 \rightarrow \text{Algorithm} \rightarrow 01000110$

- The point of compression is to store data using less space.
- In this example, we are using less space.
- But can we recover our original string from the output?
Reversing the algorithm

- Try to reverse the algorithm. The problem is that we can parse the output, 01000110, in several ways. One way is
  01 – one in position 1 in the window
  00 – one in position 0 in the window
  0 – no ones in the window
  11 – one in position 3 in the window
  00 – one in position 0 in the window
- This gives $s = 0110000000010$, our original input.
Reversing the algorithm

- Another way to parse the output, 01000110, is:
  01 – one in position 1 in the window
  0 – no ones in the window
  00 – one in position 0 in the window
  11 – one in position 3 in the window
  0 – no ones in the window

- This gives \( s = 010000100010 \), a different input with the same output.
Is this a valid compression algorithm?

• Let \( n = 12 \), \( k = 3 \).

\[
\begin{align*}
011000000010 & \rightarrow \text{Algorithm} \rightarrow 01000110 \\
010000100010 & \rightarrow \text{Algorithm} \rightarrow 01000110
\end{align*}
\]

• This is not a valid compression algorithm. It is somehow throwing away some crucial information we need to reconstruct the string.

• How can we modify the algorithm so that we can always reconstruct an input from an output?
Compressing Fixed Density Strings

• Idea for modification: Use a *marker bit* to indicate when to interpret the output as a position.

• A modified compression algorithm:
  - Only look ahead a few positions in the string.
  - If there is a one, record a 1 to say to interpret the next bits as a position within the window. Then record its position within the window, and move up the window.
  - If there are no ones, record a 0 and move up the window.
Example

\[ s = \underline{011000000010} \quad \leftarrow \text{one in the window in position 1} \]

- output = 101
- 1 – there is a one in the window
- 01 – one in position 1 in the window
Example

\[ s = \overline{011000000010} \leftarrow \text{one in the window in position 0} \]

\[ 0123 \]

- output = 101100
  1 – there is a one in the window
  01 – one in position 1 in the window
  1 – there is a one in the window
  00 – one in position 0 in the window
Example

\[ s = 011000000010 \quad \leftarrow \quad \text{no ones in the window}\]

- output = 1011000
  1 – there is a one in the window
  01 – one in position 1 in the window
  1 – there is a one in the window
  00 – one in position 0 in the window
  0 – no ones in the window
Example

\[ s = 011000000010 \]

0 1 2 3

\[ \text{one in the window in position 3} \]

- output = 1011000111

... 

0 – no ones in the window

1 – there is a one in the window

11 – one in position 3 in the window
Example

s = 011000000010 ← no ones in the window

- output = 10110001110

...  

0 – no ones in the window
1 – there is a one in the window
11 – one in position 3 in the window
0 – no ones in the window
Is this a valid compression algorithm?

• Let \( n = 12, \, k = 3. \)

\[
011000000010 \rightarrow \text{Algorithm} \rightarrow 10110001110
\]

• Trade-off: It takes more space than last time, but this is necessary in order to associate outputs with unique inputs.

• Does the output of this algorithm actually include enough information to reconstruct the input?
Is this a valid compression algorithm?

- Let $n = 12$, $k = 3$.
  
  $\underline{\text{Algorithm}} \rightarrow 10110001110$

- Does the output of this algorithm actually include enough information to reconstruct the input?
Is this a valid compression algorithm?

- Let $n = 12$, $k = 3$.

011000000010 $\rightarrow$ Algorithm $\rightarrow$ 10110001110

- Are any of these output bits extraneous?
We've seen an algorithm to encode fixed density binary strings (of length n, with k ones):

- Use a sliding window of size n/k.
- Is there a 1 in the window?
  
  - **YES:** Record a 1 as a marker. Then record the position of the first one within the window.
  
  - **NO:** Record a 0.

- Move up the window and repeat until you reach the end of the string.

```
011000000010 → Algorithm → 10110001110
```
Compressing Fixed Density Strings

- Idea for modification: After we see the last one, no need to keep recording zeros to indicate empty windows.

- A modified algorithm:
  - Same as before except keep a count of the number of 1's as you record their positions
  - Stop when all 1's have had their positions recorded.
Encoding Fixed Density Strings

WindowEncode(bitstring s of length n with k ones)

\[
j \leftarrow \text{floor}(n/k) \quad \text{\\(j\) is window length.}
\]

\[
W \leftarrow s[0, \ldots, j-1] \quad \text{\\(W\) is the window to look at.}
\]

While count < k:

If W contains no ones,

Write a 0.
Update W to move up j positions.

Else if W contains some ones,

Write a 1.
count++
p \leftarrow \text{position of the first one within } W
Write p in binary.
Update W to move up p+1 positions.
Our algorithm \texttt{WindowEncode} gives a way of encoding a fixed-density binary string in a more succinct way.

Input $s \rightarrow \text{WindowEncode} \rightarrow$ Output $t$

Can we always decode in a unique way?
Decoding Fixed Density Strings

WindowDecode(bitstring \( t \), target is string of length \( n \) with \( k \) ones)

\[
\begin{align*}
  s & \leftarrow \text{empty string}, \ j \leftarrow \text{floor}(n/k), \\
  i & \leftarrow 0, \ b \leftarrow \text{floor}(\log(j)).
\end{align*}
\]

While \( i < \text{length}(t) \),

If \( t[i] = 0 \)

Write a 0 \( j \) times to \( s \).

\( i++ \)

Else if \( t[i] = 1 \)

\[
\begin{align*}
  p & \leftarrow \text{convert to decimal} \ t[i+1, \ldots, i+b] \ \text{\( \backslash \backslash \text{position of} \ 1 \)} \\
  \text{Write a 0} \ p \ \text{times, then write a 1 to} \ s.
\end{align*}
\]

\( i \leftarrow i+b+1 \)

If \( \text{length}(s) < n \), write a 0 \( (n - \text{length}(s)) \) times to \( s \).

Output \( s \).
Inverse Functions

- Let $E(s)$ denote the result of encoding string $s$ of length $n$ with $k$ ones.
- Let $D(t)$ denote the result of decoding string $t$ to create a string of length $n$ with $k$ ones.

**Theorem:** $D(E(s)) = s$

- That is, encoding a string then decoding it gives the same string back (i.e. $D$ and $E$ are inverse functions).
Theorem: $D(E(s)) = s$

- Proof: By strong induction on $n$, the length of $s$.
  - **Base Case:** String $s$ is length 1. You can check that $D(E(1)) = 1$ and $D(E(0)) = 0$.
  - **Strong Induction Hypothesis:** Suppose $D(E(u)) = u$ for all strings $u$ of length $j < n$.
  - **Need to show:** $D(E(s)) = s$ for all strings $s$ of length $n$. 


Theorem: $D(E(s)) = s$

- **CASE 1**: String $s$ has a one among the first $n/k$ bits.
  Then $s = 0...01u$ where $u$ has length $< n$.
  Then $E(s) = 1\text{(position of the first one)}E(u)$.
  $D(E(s)) = D(1\text{(position of the first one)}E(u))$
  $= 0...01D(E(u))$
  $= 0...01u$ by the induction hypothesis
  $= s$
Theorem: $D(E(s)) = s$

- **Proof:**
  - **CASE 2:** String $s$ has no ones in the first $n/k$ bits.
    - Then $s = 0...0u$ where $u$ has length $n-(n/k)$.
    - Then $E(s) = 0E(u)$.
    - $D(E(s)) = D(0E(u))$
      - $= 0...0D(E(u))$
      - $= 0...0u$ by the induction hypothesis
      - $= s$

- Therefore, in both cases, $D(E(s)) = s$. 
Output Size

- How many bits is the output?

- To simplify our analysis, assume \( n/k \) is a power of two, so that
  - \( n/k \) is an integer, and
  - \( \log(n/k) \) is an integer.
• **Best Case:** ones are towards the beginning
  - Window size $= n/k$
  - 1 marker bit for each one in the string
  - $\log(n/k)$ bits to specify position of each one within a window
  - $k$ such ones
  - Total number of bits $= k*(\log(n/k) + 1)$
  \[= k*\log(n/k) + k\]
• **Worst Case:** ones are towards the end
  - Window size  = \( n/k \)
  - 1 marker bit for each one in the string
  - \( \log(n/k) \) bits to specify position of each one within a window
  - \( k \) such ones
  - plus some bits (at most \( k \)) to indicate that there are no ones in the first several windows
  - Total number of bits = \( k*(\log(n/k) + 1) + k \)
    
    \[ = k*\log(n/k) + 2k \]
If $t$ is the output of applying $\text{WindowEncode}$ to string $s$ of length $n$ with $k$ ones, then

$$k \cdot \log\left(\frac{n}{k}\right) + k \leq |t| \leq k \cdot \log\left(\frac{n}{k}\right) + 2k$$

Since we have seen how to encode any fixed density binary string in at most

$$k \cdot \log\left(\frac{n}{k}\right) + 2k$$

bits, it means the number of fixed density binary strings is at most
Output Size

- If $t$ is the output of applying `WindowEncode` to string $s$ of length $n$ with $k$ ones, then
  
  $k \times \log\left(\frac{n}{k}\right) + k \leq |t| \leq k \times \log\left(\frac{n}{k}\right) + 2k$

  Since we have seen how to encode any fixed density binary string in at most
  
  $k \times \log\left(\frac{n}{k}\right) + 2k$ bits,

  it means the number of fixed density binary strings is at most
  
  $k \times \log\left(\frac{n}{k}\right) + 2k$

  $\neq 2$
An Upper Bound

Since we know the exact number of fixed density binary strings, this gives

\[
\binom{n}{k} \leq 2^k \log \left( \frac{n}{k} \right) + 2k
\]

\[
\leq 2^k \log \left( \frac{n}{k} \right) \cdot 2^{2k}
\]

\[
\leq 2^k \log \left( \frac{n}{k} \right)^k + 4^k
\]

\[
\leq \left( \frac{n}{k} \right)^k \cdot 4^k
\]

\[
\leq \left( \frac{4n}{k} \right)^k
\]
A Lower Bound

- We will count *some* of the fixed density binary strings.
- Divide the $n$ positions into $k$ chunks, each of size $n/k$.
- Some of the fixed density binary strings will have a single one in each chunk.
- How many such strings?
A Lower Bound

- We will count *some* of the fixed density binary strings.
- Divide the $n$ positions into $k$ chunks, each of size $n/k$.
- Some of the fixed density binary strings will have a single one in each chunk.
- How many such strings?

$$\binom{n}{k}^k$$

In each chunk, choose a position for the one.
A Lower Bound

Since this only counts some of the fixed density binary strings (the strings where the ones are evenly distributed among the chunks), the total number of fixed density binary strings must be more.

\[
\left( \frac{n}{k} \right)^k \leq \binom{n}{k}
\]
Bounding the Binomial Coefficients

- What we have shown is that our compression algorithm gives us a way to bound the binomial coefficients:

\[
\left( \frac{n}{k} \right)^k \leq \binom{n}{k} \leq \left( \frac{4 \cdot n}{k} \right)^k
\]

- Counting helps us understand how good our compression algorithm is.
- Compression algorithms help us count.
Theoretically optimal encoding

- A theoretically optimal encoding would take $\lceil \log \binom{n}{k} \rceil$ bits to encode a fixed density binary string.

- This is the best we can hope for. With less bits, we could not uniquely represent all $\binom{n}{k}$ fixed density binary strings.

- Is it possible to achieve this theoretical best with an actual algorithm?
Theoretically optimal encoding

• One way to encode all $\binom{n}{k}$ fixed density strings is to list them in some order.
• Since there will be $\binom{n}{k}$ elements in the list, we can store the string's position in the list instead of the list itself.
• But how do you decode? Given a position $p$ in the list, you need a way to determine the $p^{th}$ string in the list.
• Lookup table?
What do we need?

• We need
  – A way to take a string and determine its position in the list (encode).
    \[ E(s, n, k) \]
  – A way to take a position, \( p \), and determine the string in position \( p \) in the list (decode).
    \[ D(p, n, k) \]
Lex Order

- How we encode and decode depends on the order in which we want to list the strings.
- We will use lexicographic (aka lex, aka dictionary) order.
Lex Order

- String $a$ comes before string $b$ in lex order if the first time they differ, $a$ is smaller.

$$a_1 a_2 \ldots a_n < b_1 b_2 \ldots b_n$$

means there exists a $j$ for which

$$a_i = b_i \text{ for all } i < j \text{ and } a_j < b_j$$

- Which comes first in lex order?

01011 or 01110
Example: $n = 5, \ k = 3$

<table>
<thead>
<tr>
<th>Original String $s$</th>
<th>Position $p$ (Encoded String)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00111</td>
<td>0 = 0000</td>
</tr>
<tr>
<td>01011</td>
<td>1 = 0001</td>
</tr>
<tr>
<td>01101</td>
<td>2 = 0010</td>
</tr>
<tr>
<td>01110</td>
<td>3 = 0011</td>
</tr>
<tr>
<td>10011</td>
<td>4 = 0100</td>
</tr>
<tr>
<td>10101</td>
<td>5 = 0101</td>
</tr>
<tr>
<td>10110</td>
<td>6 = 0110</td>
</tr>
<tr>
<td>11001</td>
<td>7 = 0110</td>
</tr>
<tr>
<td>11010</td>
<td>8 = 1000</td>
</tr>
<tr>
<td>11100</td>
<td>9 = 1001</td>
</tr>
</tbody>
</table>
Example: $n = 5, k = 3$

<table>
<thead>
<tr>
<th>Original String $s$</th>
<th>Position $p$ (Encoded String)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00111</td>
<td>0 = 0000</td>
</tr>
<tr>
<td>01011</td>
<td>1 = 0001</td>
</tr>
<tr>
<td>01101</td>
<td>2 = 0010</td>
</tr>
<tr>
<td>01110</td>
<td>3 = 0011</td>
</tr>
<tr>
<td>10011</td>
<td>4 = 0100</td>
</tr>
<tr>
<td>10101</td>
<td>5 = 0101</td>
</tr>
<tr>
<td>10110</td>
<td>6 = 0110</td>
</tr>
<tr>
<td>11001</td>
<td>7 = 0110</td>
</tr>
<tr>
<td>11010</td>
<td>8 = 1000</td>
</tr>
<tr>
<td>11100</td>
<td>9 = 1001</td>
</tr>
</tbody>
</table>
Pascal at Work

- We always know how many strings in the list start with 0, and how many start with 1, thanks to Pascal's Identity:

\[
\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}
\]
E(s, n, k)

- If \( s \) starts with 0, it will be among the first \( \binom{n-1}{k} \) items in the list.
- If \( s \) starts with 1, its position will be at least \( \binom{n-1}{k} \).

\[
E(s,n,k)
\]

If \( n=1 \), Return 0.

If \( s_1 = 0 \),

Return \( E(s_2 \ldots s_n, n-1, k) \).

Else if \( s_1 = 1 \),

Return \( E(s_2 \ldots s_n, n-1, k-1) + (n-1)\text{choose}(k) \).
D(p, n, k)

- If $p < \binom{n-1}{k}$, $s$ must start with 0.
- If $p \geq \binom{n-1}{k}$, $s$ must start with 1.

\begin{align*}
D(p, n, k) &= \\
&\text{If } n=k, \text{ Return the string of } n \text{ 1's.} \\
&\text{If } k=0, \text{ Return the string of } n \text{ 0's.} \\
&\text{If } p < (n-1)\text{choose}(k), \\
&\quad \text{Return } 0D(p, n-1, k). \\
&\text{Else if } p \geq (n-1)\text{choose}(k), \\
&\quad \text{Return } 1D(p - (n-1)\text{choose}(k), n-1, k-1).
\end{align*}
The encoded string for any input string $s$ is a number in the range 1 through $\binom{n}{k}$.

So it takes $\lceil \log(n/k) \rceil$ bits to store fixed density binary strings using this lex order encoding.

This is the theoretical lower bound, so the lex order encoding is **OPTIMAL**.