# Data Compression and IoT

Ulises Tirado Zatarain [1]
(ulises.tirado@cimat.mx)

[1] Algorists Group

July, 2016

# Outline

1. Introduction
    - Definitions
    - Example/Problem
    - Some ideas and approaches
2. Basic algorithms (Loseless)
    - Run-Length Encoding
    - Improving RLE
    - Decompression from RLE
    - More improves to RLE
3. Advanced algorithms (Loseless)
    - Probability theory review
    - Data structure review
    - Huffman Encoding
4. Advanced algorithms (Lossy)

# Outline

1. Introduction
   - Definitions
   - Example/Problem
   - Some ideas and approaches
2. Basic algorithms (Loseless)
   - Run-Length Encoding
   - Improving RLE
   - Decompression from RLE
   - More improves to RLE
3. Advanced algorithms (Loseless)
   - Probability theory review
   - Data structure review
   - Huffman Encoding
4. Advanced algorithms (Lossy)

# What is data compression?

**Definition (Data compression)**

Representation of information using less space than original data. The action to compress data is called **compression** and the opposite actions is called **decompression**. It's a particular case of encoding/decoding information.

- Kinds of compression:
  - Loseless
  - Lossy

# Loseless compression

- Information can be retrieved exactly as original data.
- Usually used for text compression
- Some known formats:
  - Zip
  - GZip
  - RAR
  - ACE
  - 7Zip
  - B2Zip
  - ...

# Lossy compression

- Information loses some data, that cannot be retrieved exactly as before it is compressed.
- Usually used for media compression: images, audio, video.
- Some known formats:
  - JPEG, GIF, PNG, ...
  - MP3, OGG, AAC, ...
  - H264, MPEG-4, VP8, ...

# Kinds of information

- **Redundant**
  - Repetitive data
  - Predictable data
- **Irrelevant**
  - Invisible data
  - Removing this data don't affect message content
- **Basic**
  - Essential data
  - It's needed to retrieve original data
  - It should be transmitted

## Kinds of information

- **Redundant**
  - Repetitive data
  - Predictable data
- **Irrelevant**
  - Invisible data
  - Removing this data don't affect message content
- **Basic**
  - Essential data
  - It's needed to retrieve original data
  - It should be transmitted

# Kinds of information

- **Redundant**
    - Repetitive data
    - Predictable data
- Irrelevant
    - Invisible data
    - Removing this data don't affect message content
- Basic
    - Essential data
    - It's needed to retrieve original data
    - It should be transmitted

# Kinds of information

- **Redundant**
    - Repetitive data
    - Predictable data
- **Irrelevant**
    - Invisible data
    - Removing this data don't affect message content
- Basic
    - Essential data
    - It's needed to retrieve original data
    - It should be transmitted

# Kinds of information

- **Redundant**
  - Repetitive data
  - Predictable data
- **Irrelevant**
  - Invisible data
  - Removing this data don't affect message content
- **Basic**
  - Essential data
  - It's needed to retrieve original data
  - It should be transmitted

# Kinds of information

- **Redundant**
  - Repetitive data
  - Predictable data
- **Irrelevant**
  - Invisible data
  - Removing this data don't affect message content
- Basic
  - Essential data
  - It's needed to retrieve original data
  - It should be transmitted

# Kinds of information

- **Redundant**
  - Repetitive data
  - Predictable data
- **Irrelevant**
  - Invisible data
  - Removing this data don't affect message content
- **Basic**
  - Essential data
  - It's needed to retrieve original data
  - It should be transmitted

# Kinds of information

- **Redundant**
  - Repetitive data
  - Predictable data
- **Irrelevant**
  - Invisible data
  - Removing this data don't affect message content
- **Basic**
  - Essential data
  - It's needed to retrieve original data
  - It should be transmitted

# Kinds of information

- **Redundant**
  - Repetitive data
  - Predictable data
- **Irrelevant**
  - Invisible data
  - Removing this data don't affect message content
- **Basic**
  - Essential data
  - It's needed to retrieve original data
  - It should be transmitted

# Kinds of information

- **Redundant**
  - Repetitive data
  - Predictable data
- **Irrelevant**
  - Invisible data
  - Removing this data don't affect message content
- **Basic**
  - Essential data
  - It's needed to retrieve original data
  - It should be transmitted

# Outline

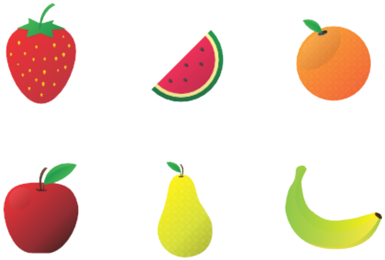
1. Introduction
   - Definitions
   - Example/Problem
   - Some ideas and approaches
2. Basic algorithms (Loseless)
   - Run-Length Encoding
   - Improving RLE
   - Decompression from RLE
   - More improves to RLE
3. Advanced algorithms (Loseless)
   - Probability theory review
   - Data structure review
   - Huffman Encoding
4. Advanced algorithms (Lossy)

# Lets see an example: Fruit 100% random & $\mathfrak{I}(\text{country})$

There are six popular fruits in an imaginary random country with some states (about 32). People in the country implements an elections system to know: What's the favorite fruit ever in this random, imaginary and 100% hypothetical country?

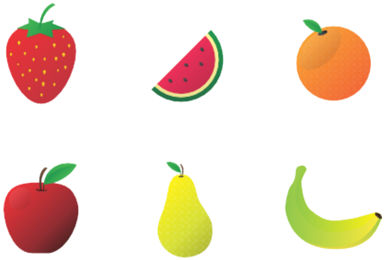
# Lets see an example: Fruit 100% random & $\Im$(country)

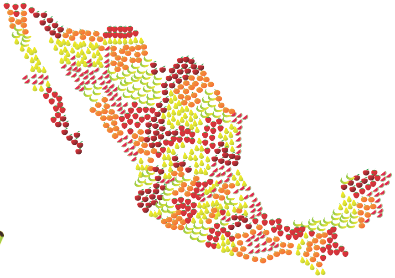
There are six popular fruits in an imaginary random country with some states (about 32). People in the country implements an elections system to know: What's the favorite fruit ever in this random, imaginary and 100% hypothetical country?



# Lets see an example: Fruit 100% random & $\mathfrak{I}(country)$

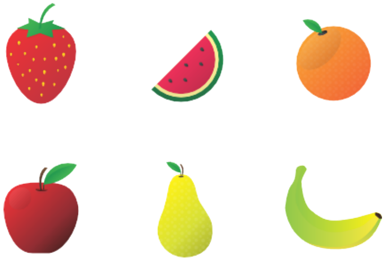
There are six popular fruits in an imaginary random country with some states (about 32). People in the country implements an elections system to know: What's the favorite fruit ever in this random, imaginary and 100% hypothetical country?

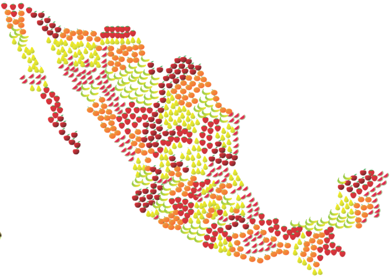




# Lets see an example: Fruit 100% random & $\mathfrak{I}$(country)

There are six popular fruits in an imaginary random country with some states (about 32). People in the country implements an elections system to know: What's the favorite fruit ever in this random, imaginary and 100% hypothetical country?





Can you see the different kinds of information?

# Fruit 100% random & $\mathfrak{I}$(country): Game rules

The election system has following rules:

- Each citizen has an unique ID scanned from his/her ID card.
- Each citizen can vote only once and only by one fruit.
- If somebody tries to vote twice or more, then all votes from this citizen will be invalidated.
- Any citizen can vote in any state.
- There is a central system publishing partial live results.
- Each state has a system to votes counting and this reports to the central system. This systems only can report (to central system) votes from citizens who are natives from that state.
- In anytime the systems in each states can communicate with the other state systems to report votes from non-native citizens.

# Fruit 100% random & $\mathfrak{I}$(country): Game rules

The election system has following rules:

- Each citizen has an unique ID scanned from his/her ID card.
- Each citizen can vote only once and only by one fruit.
- If somebody tries to vote twice or more, then all votes from this citizen will be invalidated.
- Any citizen can vote in any state.
- There is a central system publishing partial live results.
- Each state has a system to votes counting and this reports to the central system. This systems only can report (to central system) votes from citizens who are natives from that state.
- In anytime the systems in each states can communicate with the other state systems to report votes from non-native citizens.

# Fruit 100% random & $\mathfrak{I}$(country): Game rules

The election system has following rules:

- Each citizen has an unique ID scanned from his/her ID card.
- Each citizen can vote only once and only by one fruit.
- If somebody tries to vote twice or more, then all votes from this citizen will be invalidated.
- Any citizen can vote in any state.
- There is a central system publishing partial live results.
- Each state has a system to votes counting and this reports to the central system. This systems only can report (to central system) votes from citizens who are natives from that state.
- In anytime the systems in each states can communicate with the other state systems to report votes from non-native citizens.

# Fruit 100% random & $\mathfrak{I}$(country): Game rules

The election system has following rules:

- Each citizen has an unique ID scanned from his/her ID card.
- Each citizen can vote only once and only by one fruit.
- If somebody tries to vote twice or more, then all votes from this citizen will be invalidated.
- Any citizen can vote in any state.
- There is a central system publishing partial live results.
- Each state has a system to votes counting and this reports to the central system. This systems only can report (to central system) votes from citizens who are natives from that state.
- In anytime the systems in each states can communicate with the other state systems to report votes from non-native citizens.

# Fruit 100% random & $\mathfrak{I}$(country): Game rules

The election system has following rules:

- Each citizen has an unique ID scanned from his/her ID card.
- Each citizen can vote only once and only by one fruit.
- If somebody tries to vote twice or more, then all votes from this citizen will be invalidated.
- Any citizen can vote in any state.
- There is a central system publishing partial live results.
- Each state has a system to votes counting and this reports to the central system. This systems only can report (to central system) votes from citizens who are natives from that state.
- In anytime the systems in each states can communicate with the other state systems to report votes from non-native citizens.

# Fruit 100% random & $\mathfrak{I}$(country): Game rules

The election system has following rules:

- Each citizen has an unique ID scanned from his/her ID card.
- Each citizen can vote only once and only by one fruit.
- If somebody tries to vote twice or more, then all votes from this citizen will be invalidated.
- Any citizen can vote in any state.
- There is a central system publishing partial live results.
- Each state has a system to votes counting and this reports to the central system. This systems only can report (to central system) votes from citizens who are natives from that state.
- In anytime the systems in each states can communicate with the other state systems to report votes from non-native citizens.

# Fruit 100% random & $\mathfrak{I}$(country): Game rules

The election system has following rules:

- Each citizen has an unique ID scanned from his/her ID card.
- Each citizen can vote only once and only by one fruit.
- If somebody tries to vote twice or more, then all votes from this citizen will be invalidated.
- Any citizen can vote in any state.
- There is a central system publishing partial live results.
- Each state has a system to votes counting and this reports to the central system. This systems only can report (to central system) votes from citizens who are natives from that state.
- In anytime the systems in each states can communicate with the other state systems to report votes from non-native citizens.

# Outline

1. Introduction
   - Definitions
   - Example/Problem
   - Some ideas and approaches
2. Basic algorithms (Loseless)
   - Run-Length Encoding
   - Improving RLE
   - Decompression from RLE
   - More improves to RLE
3. Advanced algorithms (Loseless)
   - Probability theory review
   - Data structure review
   - Huffman Encoding
4. Advanced algorithms (Lossy)

# Logistics (brainstorming)

- How people can vote?
- Is an app needed?
- How people can vote outside of their state? (non-native people)
- Infrastructure?

# Logistics (brainstorming)

- How people can vote?
- Is an app needed?
- How people can vote outside of their state? (non-native people)
- Infrastructure?

# Logistics (brainstorming)

- How people can vote?
- Is an app needed?
- How people can vote outside of their state? (non-native people)
- Infrastructure?

# Logistics (brainstorming)

- How people can vote?
- Is an app needed?
- How people can vote outside of their state? (non-native people)
- Infrastructure?

# Architecture and design (brainstorming)

- First, think in the small case (i.e. one server by state)
- Solve for this case
- Improve to solve big case (i.e. dividing each states by districts)

## Architecture and design (brainstorming)

- First, think in the small case (i.e. one server by state)
- Solve for this case
- Improve to solve big case (i.e. dividing each states by districts)

# Architecture and design (brainstorming)

- First, think in the small case (i.e. one server by state)
- Solve for this case
- Improve to solve big case (i.e. dividing each states by districts)

# What about data transferring? (brainstorming)

Do you have some ideas for the system?

- One vote once?
- Several votes at once?
- What technology can we use?
  - XML
  - JSON
  - Our own coding method?

# What about data transferring? (brainstorming)

Do you have some ideas for the system?

- One vote once?
- Several votes at once?
- What technology can we use?
  - XML
  - JSON
  - Our own coding method?

# What about data transferring? (brainstorming)

Do you have some ideas for the system?

- One vote once?
- Several votes at once?
- What technology can we use?
  - XML
  - JSON
  - Our own coding method?

# What about data transferring? (brainstorming)

Do you have some ideas for the system?

- One vote once?
- Several votes at once?
- What technology can we use?
  - XML
  - JSON
  - Our own coding method?

# What about data transferring? (brainstorming)

Do you have some ideas for the system?

- One vote once?
- Several votes at once?
- What technology can we use?
  - XML
  - JSON
  - Our own coding method?

# What about data transferring? (brainstorming)

Do you have some ideas for the system?

- One vote once?
- Several votes at once?
- What technology can we use?
  - XML
  - JSON
  - Our own coding method?

# Data transferring: XML? (brainstorming)

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE FruitCountry SYSTEM "votes.dtd">
<state id="25">
    <vote>
        <citizen id="111999" />
        <by>Apple</by>
    </vote>
    ...
    <vote>
        <citizen id="333777" />
        <by>Strawberry</by>
    </vote>
</state>
```

# Data transferring: JSON? (brainstorming)

```
{
    state: 25,
    votes: [
        { citizen: 111999, by: 'Apple' },
        { citizen: 222888, by: 'Pear' },
        { citizen: 222888, by: 'Banana' },
        { citizen: 222888, by: 'Watermelon' },
        ...
        { citizen: 333777, by: 'Strawberry' }
        { citizen: 333777, by: 'Orange' }
    ]
}
```

# Data transferring: Our own coding method? (brainstorming)

- What if we use some abreviations?
  - A: Apple
  - B: Banana
  - O: Orange
  - P: Pear
  - S: Strawberry
  - W: Watermelon
- Do we really need to send the citizen ID?
- Do we really need to send the state ID?
- Fixed width messages?
- A possible message from state to central system:

$$25 \quad AAAAPPPPPBBBBBWWWSSOOOOOAAA$$

# Data transferring: Our own coding method? (brainstorming)

- What if we use some abreviations?
  - A: Apple
  - B: Banana
  - O: Orange
  - P: Pear
  - S: Strawberry
  - W: Watermelon
- Do we really need to send the citizen ID?
- Do we really need to send the state ID?
- Fixed width messages?
- A possible message from state to central system:

$$25 \quad AAAAPPPPPBBBBBWWWSSOOOOOAAA$$

## Data transferring: Our own coding method? (brainstorming)

- What if we use some abreviations?
    - A: Apple
    - B: Banana
    - O: Orange
    - P: Pear
    - S: Strawberry
    - W: Watermelon
- Do we really need to send the citizen ID?
- Do we really need to send the state ID?
- Fixed width messages?
- A possible message from state to central system:

$$25 \quad AAAAPPPPPBBBBBWWWSSOOOOOAAA$$

# Data transferring: Our own coding method? (brainstorming)

- What if we use some abreviations?
  - A: Apple
  - B: Banana
  - O: Orange
  - P: Pear
  - S: Strawberry
  - W: Watermelon
- Do we really need to send the citizen ID?
- Do we really need to send the state ID?
- Fixed width messages?
- A possible message from state to central system:

$$25 \quad AAAAPPPPPBBBBBWWWSSOOOOOAAA$$

# Data transferring: Our own coding method? (brainstorming)

- What if we use some abreviations?
    - A: Apple
    - B: Banana
    - O: Orange
    - P: Pear
    - S: Strawberry
    - W: Watermelon
- Do we really need to send the citizen ID?
- Do we really need to send the state ID?
- Fixed width messages?
- A possible message from state to central system:

$$25 \quad AAAAPPPPPBBBBBWWWSSOOOOOAAA$$

## Outline

1. Introduction
   - Definitions
   - Example/Problem
   - Some ideas and approaches
2. Basic algorithms (Loseless)
   - Run-Length Encoding
   - Improving RLE
   - Decompression from RLE
   - More improves to RLE
3. Advanced algorithms (Loseless)
   - Probability theory review
   - Data structure review
   - Huffman Encoding
4. Advanced algorithms (Lossy)

# Run-Length Encoding (basic idea)

## RLE Algorithm

The idea is counting the times that each character appears consecutively. For example, for a string:

$$S = aaaabbbbbbbbaaaaabbbbbbcccccbb$$

its compressed representation will be:

$$\tilde{S} = a4b8a5b6c5b2$$

## Run-Length Encoding (algorithm v1.0)

```
function char * compress(const char *input) begin
    char *str ← input;
    char *output ← new char;
    int length ← 0;
    while $*str \neq 0$ do
        char x ← *str;
        push-back(output,x);
        int k ← 1;
        while $x = *(++str)$ do k++;
        push-back(output,to-alpha(k));
        length ← length + k + 1;
    end
    return strlen(output) < length ? output : input;
end
```

# Run-Length Encoding (inconvenients)

- What about decompression?

# Run-Length Encoding (inconvenients)

- What about decompression? Very simple, duh!

# Run-Length Encoding (inconvenients)

- What about decompression? Very simple, duh!
- What happens if we got a compressed (and ambiguos) string like following?

$$\tilde{S} = a3\mathbf{15}b3$$

# Run-Length Encoding (inconvenients)

- What about decompression? Very simple, duh!
- What happens if we got a compressed (and ambiguos) string like following?

$$\tilde{S} = a3\mathbf{15}b3$$

- Maybe, original string was like:

$$S = aaa1111bbbb$$

# Run-Length Encoding (inconvenients)

- What about decompression? Very simple, duh!
- What happens if we got a compressed (and ambiguos) string like following?

$$\tilde{S} = a315b3$$

- Maybe, original string was like:

$$S = aaa1111bbbb$$

- Or maybe was:

$$S = aaa \cdots aaabbbb$$

# Run-Length Encoding (inconvenients)

- What about decompression? Very simple, duh!
- What happens if we got a compressed (and ambiguos) string like following?

$$\tilde{S} = a3\mathbf{15}b3$$

- Maybe, original string was like:

$$S = aaa1111bbbb$$

- Or maybe was:

$$S = aaa \cdots aaabbbb$$

- Is this algorithm effective with XML or JSON?

# Outline

1. Introduction
   - Definitions
   - Example/Problem
   - Some ideas and approaches
2. Basic algorithms (Loseless)
   - Run-Length Encoding
   - Improving RLE
   - Decompression from RLE
   - More improves to RLE
3. Advanced algorithms (Loseless)
   - Probability theory review
   - Data structure review
   - Huffman Encoding
4. Advanced algorithms (Lossy)

# Run-Length Encoding (improved v2.0)

- Maybe, we can use a separator/delimiter character?

# Run-Length Encoding (improved v2.0)

- Maybe, we can use a separator/delimiter character?
  - What character can we use to «,»,«|»,...?

# Run-Length Encoding (improved v2.0)

- Maybe, we can use a separator/delimiter character?
  - What character can we use to «,»,«|»,...?
  - What if this character is in the original message?

$$S = ||||,,, \Rightarrow \tilde{S} = |4,,3 \text{ or } \tilde{S} = |4|,3$$

# Run-Length Encoding (improved v2.0)

- Maybe, we can use a separator/delimiter character?
  - What character can we use to «,»,«|»,...?
  - What if this character is in the original message?

$$S = ||||,,, \Rightarrow \tilde{S} = |4,,3 \text{ or } \tilde{S} = |4|,3$$

- What if we only have two kinds of characters?

# Run-Length Encoding (improved v2.0)

- Maybe, we can use a separator/delimiter character?
  - What character can we use to «,»,«|»,...?
  - What if this character is in the original message?

$$S = ||||,,, \Rightarrow \tilde{S} = |4,,3 \text{ or } \tilde{S} = |4|,3$$

- What if we only have two kinds of characters?
  - Thinking in binary :)

# Run-Length Encoding (improved v2.0)

- Maybe, we can use a separator/delimiter character?
  - What character can we use to «,»,«|»,...?
  - What if this character is in the original message?

$$S = ||||,,, \Rightarrow \tilde{S} = |4,,3 \text{ or } \tilde{S} = |4|,3$$

- What if we only have two kinds of characters?
  - Thinking in binary :) The Beattles - Let it «bit»! XD

# Run-Length Encoding (improved v2.0)

- Maybe, we can use a separator/delimiter character?
  - What character can we use to «,»,«|»,...?
  - What if this character is in the original message?

$$S = ||||,,, \Rightarrow \tilde{S} = |4,,3 \text{ or } \tilde{S} = |4|,3$$

- What if we only have two kinds of characters?
  - Thinking in binary :) The Beattles - Let it «bit»! XD

$$S = "e >< v" = \quad 00100101 \quad 00111110 \quad 00111100 \quad 01110110$$

$$\tilde{S} = 010001010001001001010101011100011011001010001$$

# Run-Length Encoding (improved v2.0)

- Maybe, we can use a separator/delimiter character?
  - What character can we use to «,»,«|»,...?
  - What if this character is in the original message?

$$S = ||||,,, \Rightarrow \tilde{S} = |4,,3 \text{ or } \tilde{S} = |4|,3$$

- What if we only have two kinds of characters?
  - Thinking in binary :) The Beattles - Let it «bit»! XD

$$S = "e><v" = \quad 00100101 \quad 00111110 \quad 00111100 \quad 01110110$$
$$\tilde{S} = 010001010001001001010101011100011011001010001$$

# Run-Length Encoding (improved v2.0)

- Maybe, we can use a separator/delimiter character?
  - What character can we use to «,»,«|»,…?
  - What if this character is in the original message?

$$S = ||||,,, \Rightarrow \tilde{S} = |4,,3 \text{ or } \tilde{S} = |4|,3$$

- What if we only have two kinds of characters?
  - Thinking in binary :) The Beattles - Let it «bit»! XD

$$S = \text{"}e >< v\text{"} = \quad 00100101 \quad 00111110 \quad 00111100 \quad 01110110$$
$$\tilde{S} = 010001010001001001010101011100011011001010001$$

**FAIL!!**

# Run-Length Encoding (improved v2.0)

- Maybe, we can use a separator/delimiter character?
    - What character can we use to «,»,«|»,...?
    - What if this character is in the original message?

$$S = ||||,,, \Rightarrow \tilde{S} = |4,,3 \text{ or } \tilde{S} = |4|,3$$

- What if we only have two kinds of characters?
    - Thinking in binary :) The Beattles - Let it «bit»! XD

$$S = "e >< v" = \quad 00100101 \quad 00111110 \quad 00111100 \quad 01110110$$
$$\tilde{S} = 010001010001001001010101011100011011001010001$$

**FAIL!!**

- What if we limit the repetitions? (i.e. MAX 9 repetitions)

$$S = aaaaaaaaaaaabbb$$
$$\tilde{S} = a9a3b3$$

# Run-Length Encoding (improved v2.0)

- What if the number of repetitions always are represented by only one byte?

  - What is the maximum repetition for a single character in this representation?
  - How can detect corrupted data from compressed message?

# Run-Length Encoding (improved v2.0)

- What if the number of repetitions always are represented by only one byte?

  $$\tilde{S} = a!b\#$$

  This compressed string represents 33 $a$'s followed by 35 $b$'s. Why?

  - What is the maximum repetition for a single character in this representation?
  - How can detect corrupted data from compressed message?

# Run-Length Encoding (improved v2.0)

- What if the number of repetitions always are represented by only one byte?

  $$\tilde{S} = a!b\#$$

  This compressed string represents $33$ $a$'s followed by $35$ $b$'s. Why?

  - The ASCII code of ! is $33$ and the code of $\#$ is $35$. Then, in this representation the $2k$-th characters are basic information and $(2k+1)$-th characters are number of repetitions, where $k = 0, 1, 2, \ldots, \frac{n}{2}$.
  - What is the maximum repetition for a single character in this representation?
  - How can detect corrupted data from compressed message?

# Run-Length Encoding (improved v2.0)

- What if the number of repetitions always are represented by only one byte?

  $$\tilde{S} = a!b\#$$

  This compressed string represents $33$ $a$'s followed by $35$ $b$'s. Why?

  - The ASCII code of ! is $33$ and the code of $\#$ is $35$. Then, in this representation the $2k$-th characters are basic information and $(2k+1)$-th characters are number of repetitions, where $k = 0, 1, 2, \ldots, \frac{n}{2}$.
  - What is the maximum repetition for a single character in this representation?
  - How can detect corrupted data from compressed message?

# Run-Length Encoding (improved v2.0)

- What if the number of repetitions always are represented by only one byte?

  $$\tilde{S} = a!b\#$$

  This compressed string represents $33$ $a$'s followed by $35$ $b$'s. Why?

  - The ASCII code of ! is $33$ and the code of $\#$ is $35$. Then, in this representation the $2k$-th characters are basic information and $(2k+1)$-th characters are number of repetitions, where $k = 0, 1, 2, \ldots, \frac{n}{2}$.
  - What is the maximum repetition for a single character in this representation?
  - How can detect corrupted data from compressed message?

# Run-Length Encoding (improved v2.0)

```
function char $*$ compress(const char $*$input) begin
    const char $MAX \leftarrow \sim 0$;
    char $*str \leftarrow input$;
    char $*output \leftarrow$ new char;
    int $length \leftarrow 0$;
    while $*str \neq 0$ do
        char $x \leftarrow *str$;
        push-back(output,x);
        char $k \leftarrow 1$;
        while $x = *(++str)\ \&\&\ k < MAX$ do $k{+}{+}$;
        push-back(output,k);
        $length \leftarrow length + k + 1$;
    end
    return strlen(output) $< length$ ? output : input;
end
```

# Outline

1. Introduction
   - Definitions
   - Example/Problem
   - Some ideas and approaches
2. Basic algorithms (Loseless)
   - Run-Length Encoding
   - Improving RLE
   - Decompression from RLE
   - More improves to RLE
3. Advanced algorithms (Loseless)
   - Probability theory review
   - Data structure review
   - Huffman Encoding
4. Advanced algorithms (Lossy)

# Run-Length Encoding (decompression)

```
function char $*$ decompress(const char $*$input) begin
    char $*c \leftarrow$ input;
    char $*$output $\leftarrow$ new char;
    while $*c \neq 0$ do
        char $n \leftarrow *(c+1)$;
        for $i = 1$ to $n$ do push-back(output,$*$c) ;
        $c \leftarrow c + 2$;
    end
    return output;
end
```

# Outline

1. Introduction
   - Definitions
   - Example/Problem
   - Some ideas and approaches
2. Basic algorithms (Loseless)
   - Run-Length Encoding
   - Improving RLE
   - Decompression from RLE
   - More improves to RLE
3. Advanced algorithms (Loseless)
   - Probability theory review
   - Data structure review
   - Huffman Encoding
4. Advanced algorithms (Lossy)

# Run-Length Encoding (improved v3.0)

Maybe we can do better:

- We can analize input message to see what unique characters are contained in the mesage.
- Then, we can construct a reduced alphabet. For example, $\Sigma = \{A, B, O, P, S, W\}$.
- In this case only need 3 bits to represent characters.
- We can use 5 bits to store repetitions (so, we will have MAX 31 repetitions by character).

# Run-Length Encoding (improved v3.0)

Maybe we can do better:

- We can analize input message to see what unique characters are contained in the mesage.
- Then, we can construct a reduced alphabet. For example, $\Sigma = \{A, B, O, P, S, W\}$.
- In this case only need 3 bits to represent characters.
- We can use 5 bits to store repetitions (so, we will have MAX 31 repetitions by character).

# Run-Length Encoding (improved v3.0)

Maybe we can do better:

- We can analize input message to see what unique characters are contained in the mesage.
- Then, we can construct a reduced alphabet. For example, $\Sigma = \{A, \mathrm{B}, \mathrm{O}, \mathrm{P}, S, W\}$.
- In this case only need 3 bits to represent characters.
- We can use 5 bits to store repetitions (so, we will have MAX 31 repetitions by character).

# Run-Length Encoding (improved v3.0)

Maybe we can do better:

- We can analize input message to see what unique characters are contained in the mesage.
- Then, we can construct a reduced alphabet. For example, $\Sigma = \{A, B, O, P, S, W\}$.
- In this case only need $3$ bits to represent characters.
- We can use 5 bits to store repetitions (so, we will have MAX 31 repetitions by character).

# Run-Length Encoding (improved v3.0)

Maybe we can do better:

- We can analize input message to see what unique characters are contained in the mesage.
- Then, we can construct a reduced alphabet. For example, $\Sigma = \{A, B, O, P, S, W\}$.
- In this case only need $3$ bits to represent characters.
- We can use $5$ bits to store repetitions (so, we will have MAX $31$ repetitions by character).

# Run-Length Encoding (improved v4.0 PRO)

Moreover, what happens if we can detect whole words?

$$S = \mathrm{abcdabcdabcdbdbdbdbd}$$

$$\tilde{S} = [\mathrm{abcd}]\,3\,[\mathrm{bd}]\,4$$

# Outline

1. Introduction
   - Definitions
   - Example/Problem
   - Some ideas and approaches
2. Basic algorithms (Loseless)
   - Run-Length Encoding
   - Improving RLE
   - Decompression from RLE
   - More improves to RLE
3. Advanced algorithms (Loseless)
   - Probability theory review
   - Data structure review
   - Huffman Encoding
4. Advanced algorithms (Lossy)

# Probability basics

## Experiments and Events

An event is a set of posible results in an experiment execution. For example, taking a card from a deck or rolling a dice. We can denote all posible results with $\Omega$ and an event with uppercase letter such that $A \subseteq \Omega$ or $A \in 2^{\Omega}$, then $A$ is a subset of $\Omega$.

## Probability

Is an indicator that describes the frecuency of an event in one universal set of posibilities. Daily, we express that indicator as a percentage value or value between 0 and 1. Then we can define the probability as:

$$p : 2^{\Omega} \mapsto [0, 1]$$

## Probability basics

- So, when $\Omega$ is a discrete and finite set, then:

$$p(A) = \frac{\#A}{\#\Omega}$$

- We name this as uniform distribution or counting distribution.
- However, counting elements in $A$ and $\Omega$ isn't always trivial. Maybe we need to use operations like factorial, combinations, permutations, etcetera.

# Probability basics

- Key pressing random letter of the english keyboard such that the letter be a vowel.
  - Let $A = \{a, e, i, o, u\}$ and $\Omega = \{a, ..., z\}$,then $p(A) = \frac{5}{26}$.
- Rolling a dice such that the result be greater than 2.
  - Let $A = \{3, 4, 5, 6\}$ and $\Omega = \{1, 2, 3, 4, 5, 6\}$,then $p(A) = \frac{2}{3}$.,
- Taking a card from a deck such that getting a red card.
  - Let $A = \{\mathrm{A}\heartsuit, 2\heartsuit, ..., 10\heartsuit, \mathrm{J}\heartsuit, \mathrm{Q}\heartsuit, \mathrm{K}\heartsuit, \mathrm{A}\diamondsuit, 2\diamondsuit, ..., 10\diamondsuit, \mathrm{J}\diamondsuit, \mathrm{Q}\diamondsuit, \mathrm{K}\diamondsuit\}$ and $\Omega = \left\{ \begin{array}{cc} \mathrm{A}\heartsuit, 2\heartsuit, ..., 10\heartsuit, \mathrm{J}\heartsuit, \mathrm{Q}\heartsuit, \mathrm{K}\heartsuit, & \mathrm{A}\diamondsuit, 2\diamondsuit, ..., 10\diamondsuit, \mathrm{J}\diamondsuit, \mathrm{Q}\diamondsuit, \mathrm{K}\diamondsuit, \\ \mathrm{A}\spadesuit, 2\spadesuit, ..., 10\spadesuit, \mathrm{J}\spadesuit, \mathrm{Q}\spadesuit, \mathrm{K}\spadesuit, & \mathrm{A}\clubsuit, 2\clubsuit, ..., 10\clubsuit, \mathrm{J}\clubsuit, \mathrm{Q}\clubsuit, \mathrm{K}\clubsuit \end{array} \right\}$, then $p(A) = \frac{1}{2}$.

# Outline

1. Introduction
   - Definitions
   - Example/Problem
   - Some ideas and approaches
2. Basic algorithms (Loseless)
   - Run-Length Encoding
   - Improving RLE
   - Decompression from RLE
   - More improves to RLE
3. Advanced algorithms (Loseless)
   - Probability theory review
   - Data structure review
   - Huffman Encoding
4. Advanced algorithms (Lossy)

# Priority Queue

## Wikipedia

- In computer science, a priority queue is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority.
- While priority queues are often implemented with heaps, they are conceptually distinct from heaps. A priority queue is an abstract concept like "a list" or "a map";

# C++ STL Priority Queue

## std::priority_queue

template <class T, class Container = vector<T>,
class Compare = less<typename Container::value_type>> class
priority_queue;

**Priority queue**

Priority queues are a type of container adaptors, specifically designed such that its first element is always the greatest of the elements it contains.

The container shall be accessible through random access iterators and support the following operations:

- empty
- size
- front
- push_back
- pop_front

The standard container classes vector and deque fulfill these requirements. By default, if no container class is specified for a particular priority_queue class instantiation, the standard container vector is used.

# Tries

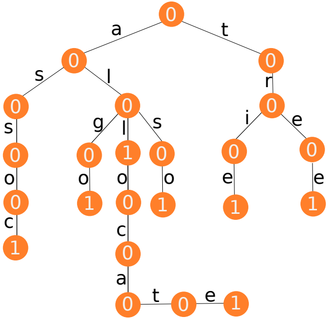
- Let a **word** be a single string and let **dictionary** be a large set of words.
- The **set<string>** and the **hash tables** can only find in a dictionary words that match exactly with the single word that we are finding.
- **Trie** is a tree type data structure that allows to represent a dictionary.
    - We can insert and find strings in $\mathcal{O}(\mathrm{L})$.
    - We can perform incremental search.

# Tries

- The word **trie** is an infix of the word "re**trie**val" because the trie can find a single word in a dictionary with only a prefix of the word.
- The trie is a tree where each vertex represents a single word or a prefix.
  - The root represents an empty string $\varepsilon$.
  - A vertex that are $k$ edges of distance of the root have an associated prefix of length $k$.
  - Let $v$ and $w$ be two vertexes of the trie, and assume that $v$ is a direct father of $w$, then $v$ must have an associated prefix of $w$.
  - Deterministic acyclic finite state automaton.

# Tries: Example

- The following trie stores the words: “algo”, “assoc”, “all”, ”allocate”, “also”, “tree” and “trie”.



- Note that every vertex of the tree does not store entire prefixes or entire words.

# How to represent tries?

- The most simple way to represent a trie is with an struct like following:

```
struct trie <typename Type = int> begin
    Type data;
    struct trie *edge[ALPHABET_SIZE];
end
```

- For the english alphabet, we can store the $'a'$-edge in $trie :: edge[0]$, $'b'$-edge in $trie :: edge[1]$, $'c'$-edge in $trie :: edge[2]$ and so on until $'z'$-edge in $trie :: edge[25]$.

# How to add a word to dictionary?

- We can add a word $w$ as following:

```
function add-word(struct trie *t , char *w) begin
    if is-empty(w) then:
        $t \rightarrow data \longleftarrow t \rightarrow data + 1$;
    else:
        if is-null($t \rightarrow edge[*w]$) then:
            $t \rightarrow edge[*w] \longleftarrow$ new struct trie;
        end
        add-word($t \rightarrow edge[*w], w + 1$);
    end
end
```

## How to find a word in dictionary?

- To find a word $w$, we can perform following algorithm:

```
function find-word(struct trie *t , char *$w$)begin
    if is-null(t) then:
        return 0;
    end
    if is-empty($w$) then:
        return t $\rightarrow$ data;
    end
    return find-word(t $\rightarrow$ edge[*$w$],$w+1$);
end
```

# Outline

1. Introduction
   - Definitions
   - Example/Problem
   - Some ideas and approaches
2. Basic algorithms (Loseless)
   - Run-Length Encoding
   - Improving RLE
   - Decompression from RLE
   - More improves to RLE
3. Advanced algorithms (Loseless)
   - Probability theory review
   - Data structure review
   - Huffman Encoding
4. Advanced algorithms (Lossy)

## Huffman Encoding

- The idea is...

# Statistical

- Statistical Pattern Recognition
  - Principal Component Analysis
- Digital Signal Processing
  - Filtering
- Image compression
  - Grayscale images
  - Color images

# References I

- Standford University
- HackerRank
- Code Forces
- Code Chef
- Wikipedia