A Crash Course on Data Compression

1. Introduction

Giulio Ermanno Pibiri

ISTI-CNR, giulio.ermanno.pibiri@isti.cnr.it

@giulio_pibiri

@jermp

Overview

- What is Data Compression and why do we need it?
- Fundamental questions and undecidability
- Some applications
- Technological limitations
- Warmup

What is Data Compression?

- The process for which data is transformed into another representation that takes *less storage space*:
  - *save space* when storing data,
  - *save time* when transmitting data.

- The process must be *reversible* (exactly or admitting some loss) to be useful.

What is Data Compression?

- The process for which data is transformed into another representation that takes *less storage space*:
  - *save space* when storing data,
  - *save time* when transmitting data.

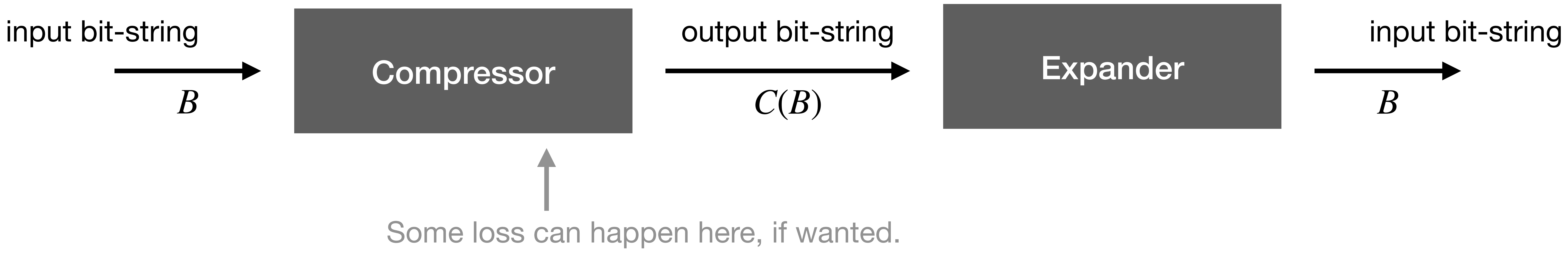
- The process must be *reversible* (exactly or admitting some loss) to be useful.

- **Q.** What is this “process”?
  A. A *computer program* that takes data as input and produces a *data structure* that takes less space than the input.

- We seek/need *efficient* programs that build such data structures.

Example: command line utility `gzip`.

Basic Model



- **Compression ratio.** The *compression ratio* is defined as $CR = |B|/|C(B)|$.
- If $CR = r$, then the size of the compressed output $|C(B)|$ is r times smaller than the input size $|B|$.

Space vs. Time Trade-Off

- The compression ratio depends on many factors:
  wanted compression/decompression speed, related to the amount of energy spent (CPU power); loss of precision; (…)

- The most common one: *the trade-off between the space of the compressed data structure and the efficiency of the operations* that we want to support on the data.

  Example: `gzip` has 9 compression "levels" (1 is fastest but "worst" compression; 9 is slower but "best").

Space vs. Time Trade-Off

- The compression ratio depends on many factors: wanted compression/decompression speed, related to the amount of energy spent (CPU power); loss of precision; (…)

- The most common one: *the trade-off between the space of the compressed data structure and the efficiency of the operations* that we want to support on the data.

  Example: `gzip` has 9 compression “levels” (1 is fastest but “worst” compression; 9 is slower but “best”).

- This trade-off is becoming more and more important: nowadays, we cannot afford the naive approach “decompress and compute”.

- Ultimate goal: allow *direct computation over compressed data*.

Limit

- **Proposition.** *No algorithm can compress every bit-string.*

  Proof.
  Proceed by contradiction. Assume that
  $|B| > |C(B)| > |C(C(B))| > |C(C(C(B)))| > \ldots$, like in the picture below.

$$\xrightarrow{B} \boxed{C} \xrightarrow{C(B)} \boxed{C} \xrightarrow{C(C(B))} \boxed{C} \longrightarrow \cdots \longrightarrow C(\ldots C(C(B))\ldots)$$

  Then all possible bit-strings could be compressed to **0 bits** — absurd. ■

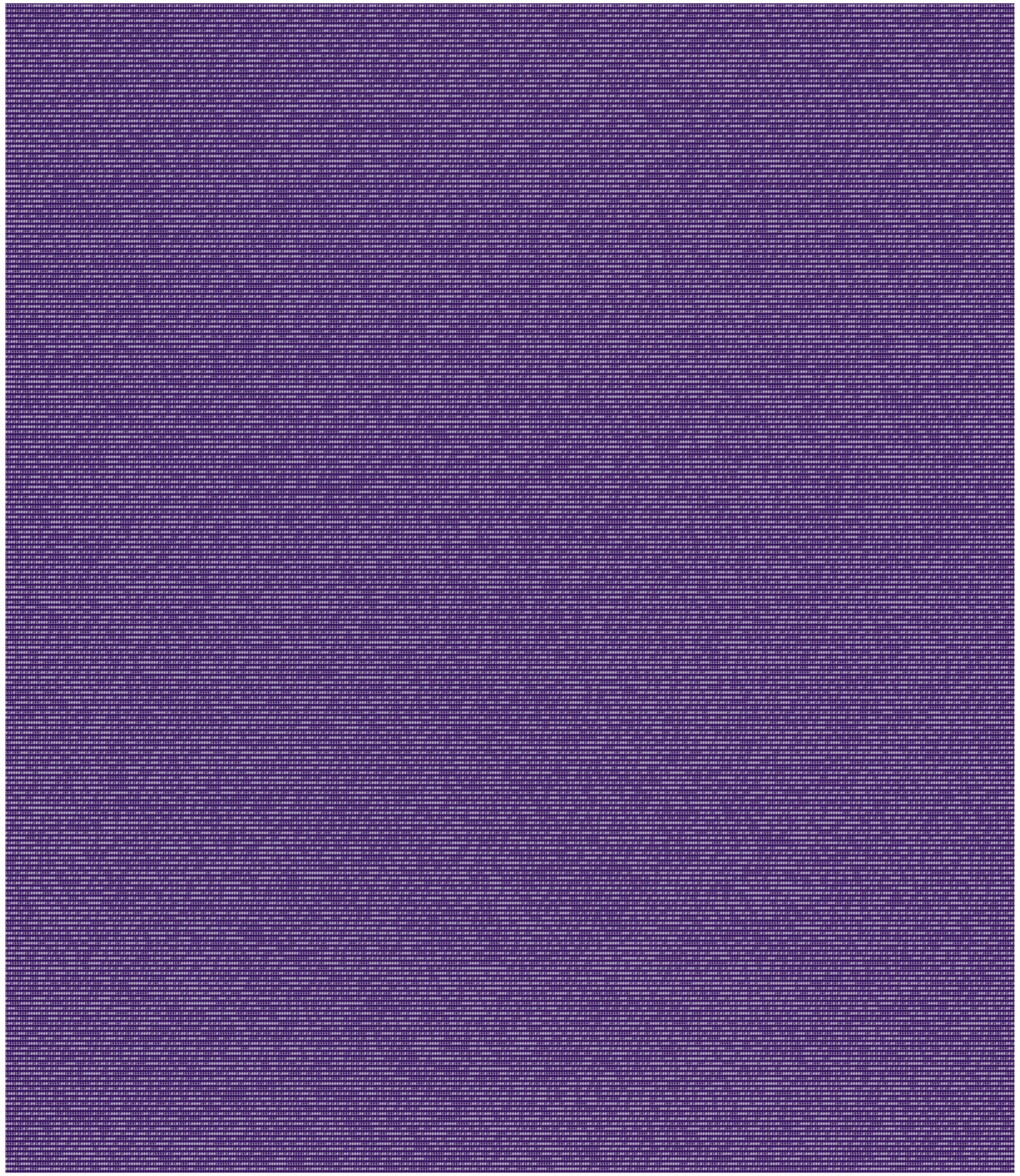
Fundamental Question(s)

- **Q.** What is the best way of compressing a file for my application?
  A. This is an *undecidable problem*.

- An extreme, but very common, example: to compress a file, you may replace it by *the program that originated the file* (related to the so-called *Kolmogorov* complexity).
  But how would you “find” (i.e., write) such a program?

- If you think: most of the data we deal with (Web pages, log files, sequencing data, ecc.) is created by programs, *not* by humans.

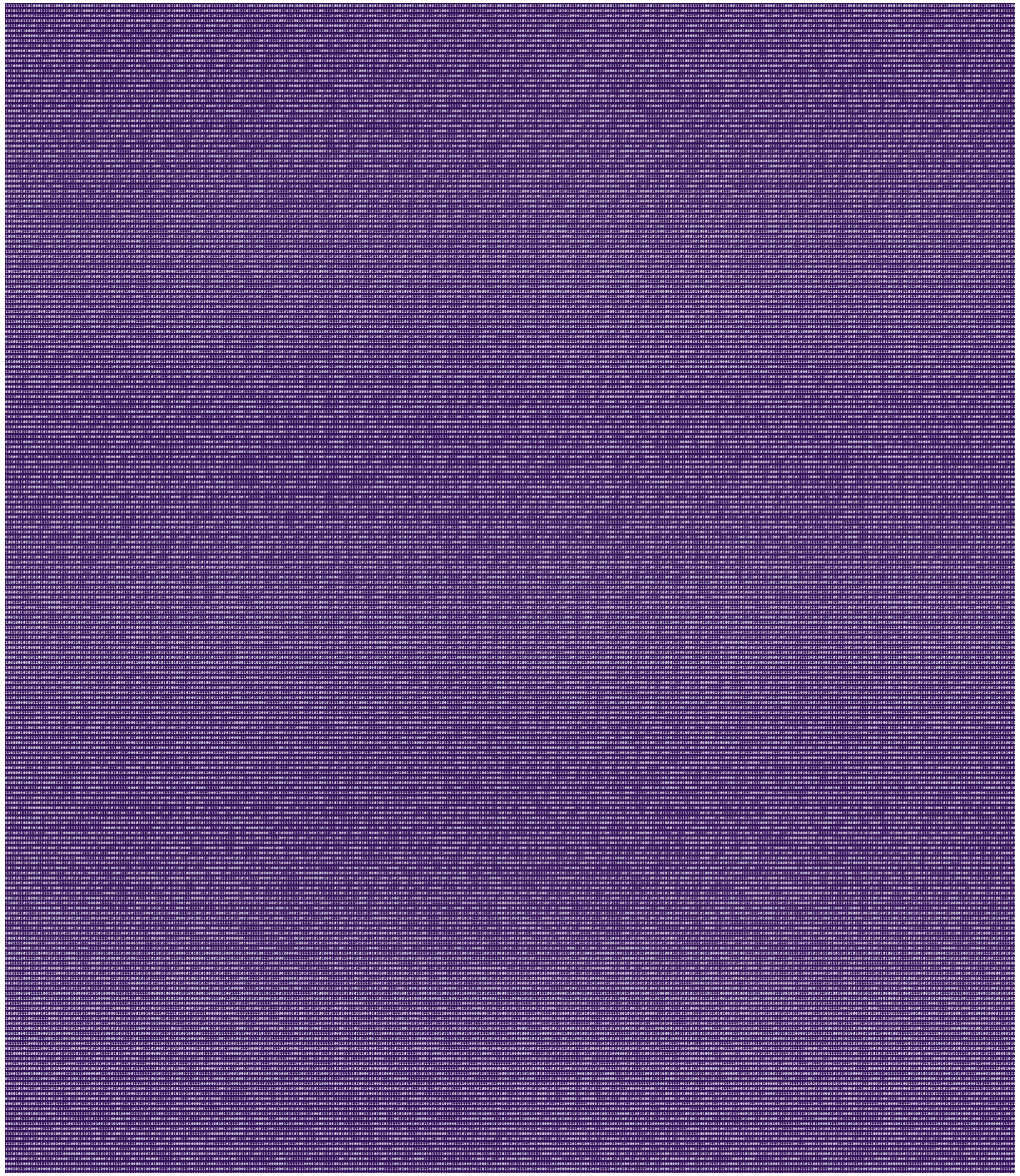
Undecidability



100,000 pseudo-random bits

Q. How would you compress these bits?

Undecidability



100,000 pseudo-random bits

Q. How would you compress these bits?

A. With the following piece of code.

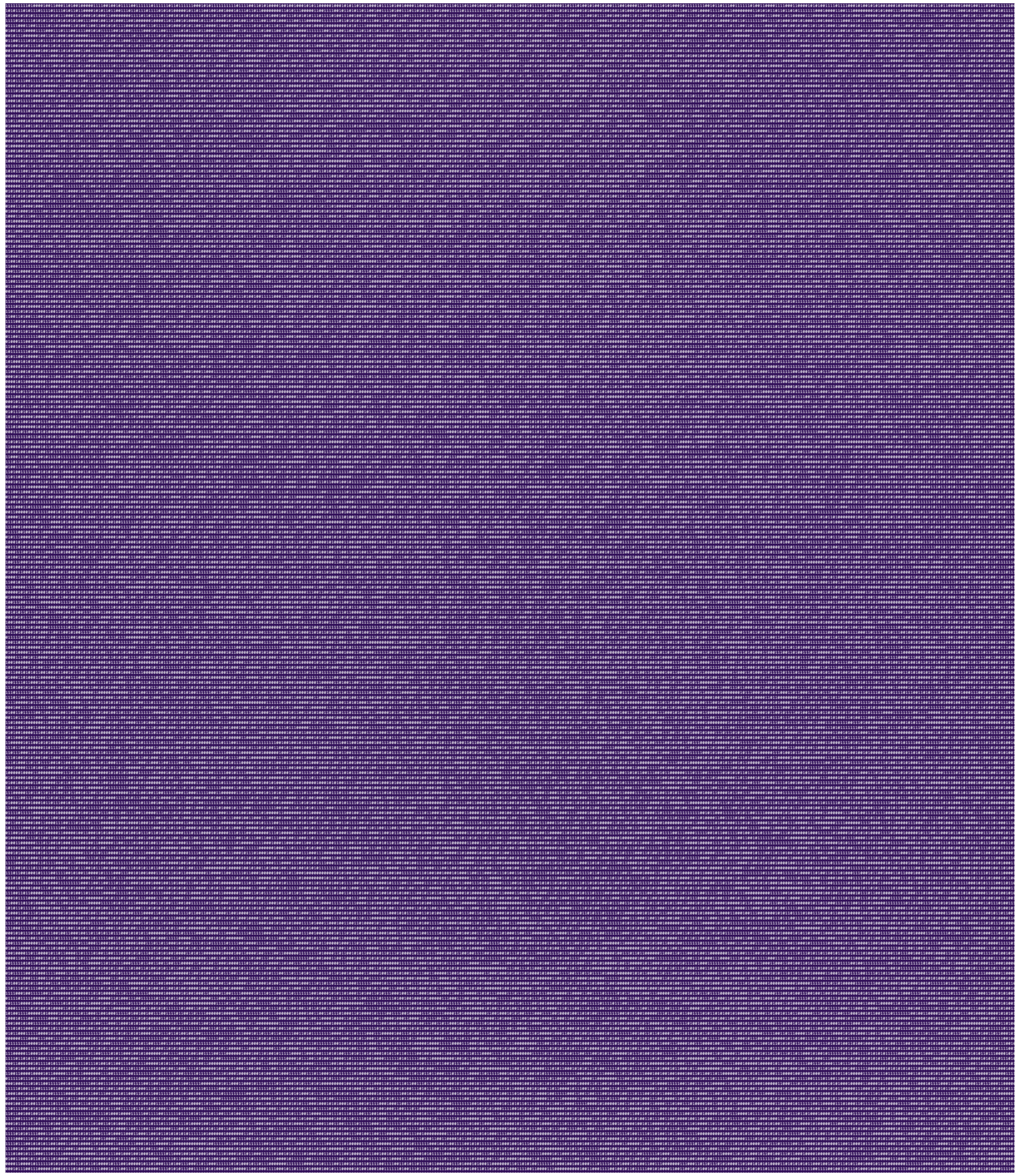
```cpp
#include <iostream>

int main() {
    int n = 100000;
    int x = 989511;
    for (int i = 0; i != n; ++i) {
        x = x * 312523 + 852596;
        std::cout << int(x > 0);
    }
    std::cout << '\n';
    return 0;
}
```

- 220 bytes (1 char = 1 byte)
- $CR = 100000/(8 \cdot 220) = 56.8$

Compile with:
`g++ random_bits.cpp -o random_bits`

Run with:
`./random_bits`

Undecidability



100,000 pseudo-random bits

Q. How would you compress these bits?

A. With the following piece of code.

```cpp
#include <iostream>

int main() {
    int n = 100000;
    int x = 989511;
    for (int i = 0; i != n; ++i) {
        x = x * 312523 + 852596;
        std::cout << int(x > 0);
    }
    std::cout << '\n';
    return 0;
}
```

What if n = `1000000`?
What happens now to the CR ?

- 220 bytes (1 char = 1 byte)
- $CR = 100000/(8 \cdot 220) = 56.8$

Compile with:
`g++ random_bits.cpp -o random_bits`

Run with:
`./random_bits`

Data and Information

- Data and information are *not* the same.
- *Information* is the knowledge coming from interpreting data according to a specific semantic scheme.
- In the future, it is foreseen that data will grow much faster than information: *data will become more and more redundant*.
- **So there are great possibilities for compression!**
  Huge amount of research being actively carried out.

Why Data Compression?

- We use compression *everywhere/anytime*; even without being aware of it.
- **Ever-increasing demand for storage and large-scale computing.**
  - Generic file compression: gzip, bzip, LZ4, Zstd, ecc.
  - Multimedia: images (jpeg, png, gif); sound (MP3); video (MPEG, DVD); Spotify, Netflix, ecc.
  - Search engines (Google, Yandex, Bing,…);
  - Distributed storage (Dropbox, Google Drive,…);
- **Communication cost.**
  - Skype, Zoom, FaceTime, WhatsApp, ecc.
  - Social networks (Facebook, Instagram, Twitter,…);

Why Data Compression?

- We use compression *everywhere/anytime*; even without being aware of it.

- **Ever-increasing demand for storage and large-scale computing.**
  - Generic file compression: gzip, bzip, LZ4, Zstd, ecc.
  - Multimedia: images (jpeg, png, gif); sound (MP3); video (MPEG, DVD); Spotify, Netflix, ecc.
  - Search engines (Google, Yandex, Bing,…);
  - Distributed storage (Dropbox, Google Drive,…);

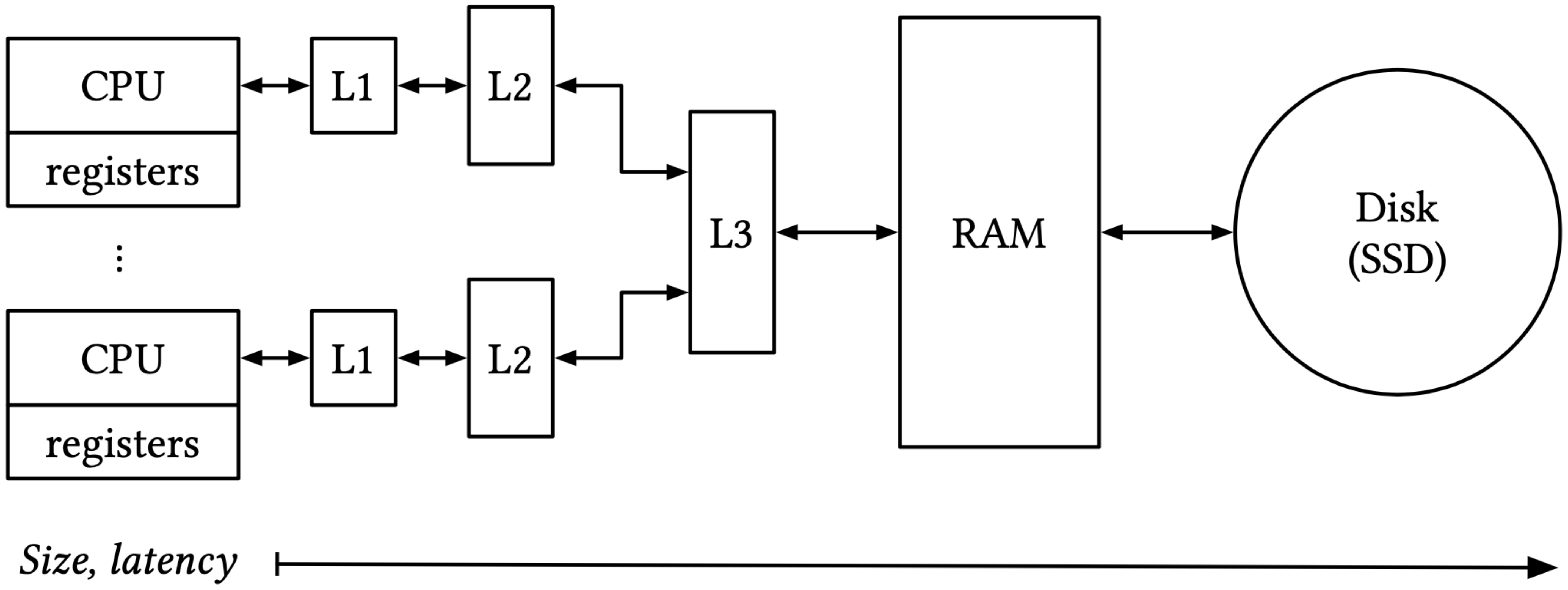
- **Communication cost.**
  - Skype, Zoom, FaceTime, WhatsApp, ecc.
  - Social networks (Facebook, Instagram, Twitter,…);

- **Increased software performance.**

Technological Limitations

- Whatever space we have available: we are going to fill it up, by virtue of our human, eager, nature.
- **Moore's Law.** *Number of transistors on a chip doubles every 1.5 – 2 years.*
- So we get faster processors…
- **But *not* faster memories!**

Memory Hierarchies



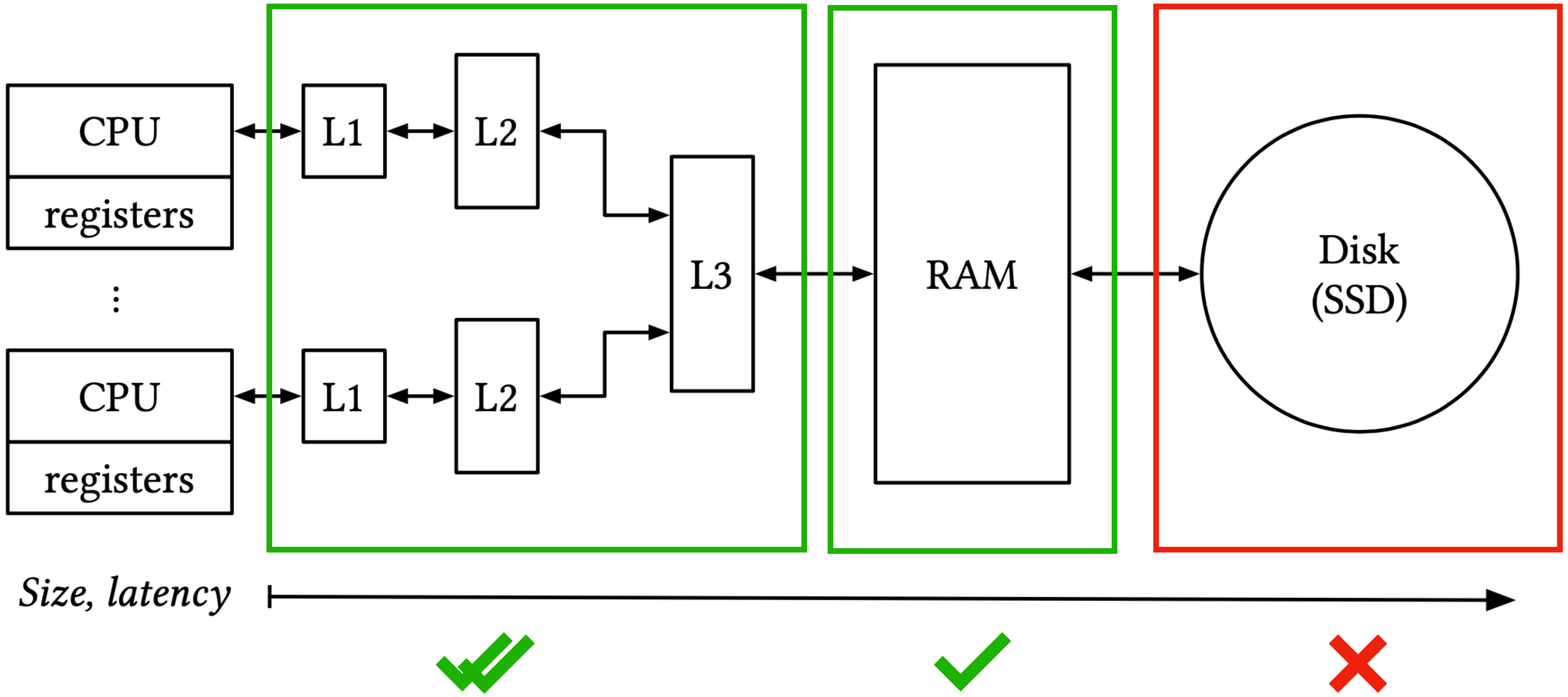
Memory type	Size	
registers	64	bits
L1	32	KB
L2	1	MB
L3	30	MB
RAM	64	GB
Disk	1	TB

Event	10^{-9} secs
L1 cache reference	1
L2 cache reference	4
RAM reference	100
SSD random read	16,000
Disk random read	3,000,000

- If a program stalls…it is likely that it is waiting for memory.
- Thus, it is more important than ever to trade-off processor time for RAM/disk access time.
- Action of compression: *transfer more data to the processor.*

https://colin-scott.github.io/personal_website/research/interactive_latency.html

Memory Hierarchies



Memory type	Size	
registers	64	bits
L1	32	KB
L2	1	MB
L3	30	MB
RAM	64	GB
Disk	1	TB

Event	10^{-9} secs
L1 cache reference	1
L2 cache reference	4
RAM reference	100
SSD random read	16,000
Disk random read	3,000,000

- If a program stalls…it is likely that it is waiting for memory.
- Thus, it is more important than ever to trade-off processor time for RAM/disk access time.
- Action of compression: *transfer more data to the processor.*

https://colin-scott.github.io/personal_website/research/interactive_latency.html

A Simple Experiment

`uint64_t` is a primitive data type for unsigned 64-bit ints;
`uint8_t` for unsigned 8-bit ints

```
struct large_record {
    uint64_t weight;
    uint64_t height;
    uint64_t day;
    uint64_t month;
    uint64_t year;
};

struct small_record {
    uint8_t weight;
    uint8_t height;
    uint8_t day;
    uint8_t month;
    uint16_t year;
};
```

With a b-bit unsigned integer, we can represent all values in $[0, 2^b)$.

`large_record` consumes **40 bytes** overall;
`small_record` consumes **6 bytes** overall (slight lie)

Experiment methodology.

1. Allocate two vectors of the same size, one holding `large_record` objects and the other holding `small_record` objects.
2. Fill the two vectors with the same data.
3. Sort the two vectors (say, on the `day` attribute).

A Simple Experiment

Experiment methodology.
1. Allocate two vectors of the same size, one holding `large_record` objects and the other holding `small_record` objects.
2. Fill the two vectors with the same data.
3. Sort the two vectors (say, on the `day` attribute).

A Simple Experiment

Experiment methodology.
1. Allocate two vectors of the same size, one holding `large_record` objects and the other holding `small_record` objects.
2. Fill the two vectors with the same data.
3. Sort the two vectors (say, on the `day` attribute).

```
constexpr unsigned seed = 13;
std::srand(seed);

std::vector<large_record> large_records;
std::vector<small_record> small_records;
large_records.reserve(vector_size);
small_records.reserve(vector_size);
for (uint64_t i = 0; i != vector_size; ++i) {
    uint64_t weight = std::rand() % 256;
    uint64_t height = std::rand() % 256;
    uint64_t day = std::rand() % 32;
    uint64_t month = std::rand() % 16;
    uint64_t year = std::rand() % 4096;
    large_records.emplace_back(weight, height, day, month, year);
    small_records.emplace_back(weight, height, day, month, year);
}
```

initialise the pseudo-random generator with a fixed seed to reproduce the results

create the vectors and reserve space

fill the vectors

1+2

A Simple Experiment

Experiment methodology.
1. Allocate two vectors of the same size, one holding `large_record` objects and the other holding `small_record` objects.
2. Fill the two vectors with the same data.
3. Sort the two vectors (say, on the `day` attribute).

```
constexpr unsigned seed = 13;
std::srand(seed);

std::vector<large_record> large_records;
std::vector<small_record> small_records;
large_records.reserve(vector_size);
small_records.reserve(vector_size);
for (uint64_t i = 0; i != vector_size; ++i) {
    uint64_t weight = std::rand() % 256;
    uint64_t height = std::rand() % 256;
    uint64_t day = std::rand() % 32;
    uint64_t month = std::rand() % 16;
    uint64_t year = std::rand() % 4096;
    large_records.emplace_back(weight, height, day, month, year);
    small_records.emplace_back(weight, height, day, month, year);
}
```

1+2

initialise the pseudo-random generator with a fixed seed to reproduce the results

create the vectors and reserve space

fill the vectors

`std::chrono` to measure time

3

```
typedef std::chrono::high_resolution_clock clock_t;
typedef std::chrono::milliseconds duration_t;
{
    auto start = clock_t::now();
    std::sort(large_records.begin(), large_records.end(),
              [](large_record const& x, large_record const& y) {
                  return x.day < y.day;
              });
    auto stop = clock_t::now();
    auto elapsed = std::chrono::duration_cast<duration_t>(stop - start);
    std::cout << "sorting vec took: " << elapsed.count() << " millisecs"
              << std::endl;
}
{
    auto start = clock_t::now();
    std::sort(small_records.begin(), small_records.end(),
              [](small_record const& x, small_record const& y) {
                  return x.day < y.day;
              });
    auto stop = clock_t::now();
    auto elapsed = std::chrono::duration_cast<duration_t>(stop - start);
    std::cout << "sorting vec took: " << elapsed.count() << " millisecs"
              << std::endl;
}
```

use the `std::sort` algorithm to sort the vectors, using a lambda function to implement the comparison

A Simple Experiment

- **Q. Which sort will take less time?**
- **Hint.** Remember! *The smaller the data, the more data can be transferred to the processor.*

Compile with:
```
g++ -std=c++11 -O3 sort_bench.cpp -o sort_bench
```

Run with:
```
./sort_bench 10000000
```

A Simple Experiment

- **Q. Which sort will take less time?**
- **Hint.** Remember! *The smaller the data, the more data can be transferred to the processor.*

Compile with:
```
g++ -std=c++11 -O3 sort_bench.cpp -o sort_bench
```

Run with:
```
./sort_bench 10000000
```

```
→  data_compression_course git:(master) ✗ cd 1_introduction/code
→  code git:(master) ✗ g++ -std=c++11 -O3 sort_bench.cpp -o sort_bench
→  code git:(master) ✗ ./sort_bench 10000000
sorting vec took: 348 millisecs
sorting vec took: 190 millisecs
→  code git:(master) ✗ ./sort_bench 100000000
sorting vec took: 3375 millisecs
sorting vec took: 1784 millisecs
→  code git:(master) ✗
```

The size of the data matters!

Further Readings

- Preface and Chapter 1 of:
  Alistair Moffat and Andrew Turpin. 2002. *Compression and coding algorithms*. Springer Science & Business Media, ISBN 978-1-4615-0935-6.

- Chapter 5.5 (pages 810-825) of:
  Robert Sedgewick and Kevin Wayne. 2011. *Algorithms.* 4-th Edition. Addison-Wesley Professional, ISBN 0-321-57351-X.