Introduction to Algorithms

IIS A. Pacinotti, Mestre (Venice)
06/06/2023

Giulio Ermanno Pibiri

Ca’ Foscari University of Venice
giulioermanno.pibiri@unive.it

About me

- As of June 2022:

  Assistant Prof. of Computer Science at Ca’ Foscari University of Venice.

- Before:
  - post-doctoral researcher at CNR, Pisa (March 2019 - June 2022)
  - Ph.D. in Computer Science from University of Pisa (Jan. 2016 - March 2019).

- Research interests:

  Algorithms and compressed data structures with applications to real-world problems, for example, in Information Retrieval and Computational Biology.

- Contact:
  - Web page: https://jermp.github.io
  - Email: giulioermanno.pibiri@unive.it

What is this lecture about?

- This is an introductory lecture to the field of *Algorithms and Data Structures*.
- **Our goal for today:** understand **why algorithms are fundamental** to solve large-scale problems.

What is this lecture about?

- This is an introductory lecture to the field of *Algorithms and Data Structures*.
- **Our goal for today:** understand **why algorithms are fundamental** to solve large-scale problems.

- **Algorithms**: **methods** (recipes) to solve a problem.
- **Data Structures**: ways to **organise the data** that it is accessed by an algorithm to solve a problem.

What is this lecture about?

- This is an introductory lecture to the field of *Algorithms and Data Structures*.
- **Our goal for today:** understand **why algorithms are fundamental** to solve large-scale problems.

- **Algorithms**: **methods** (recipes) to solve a problem.
- **Data Structures**: ways to **organise the data** that it is accessed by an algorithm to solve a problem.
- **Data Compression**: **better data representation** to enable more efficient algorithms (we will not talk about this today, though).

Overview

- 9:00 — 10:00

  Part 1 — Basic definitions, warm-up

- 10:10 — 11:00

  Part 2 — Motivations, analysis of algorithms, same applications

- 11:10 — 12:00

  Part 3 — Some example problems: integer search and sub-string search

Part 1 — Basic definitions, warm-up

Basic definitions — Algorithm

- Informally: a **recipe** to solve a problem.

Basic definitions — Algorithm

- Informally: a **recipe** to solve a problem.

Recipe to make bread (simplified):

1. Stir together water, yeast, and flour.
2. Add oil and salt.
3. Knead the dough.
4. Let the dough rest for 1 h.
5. Bake the dough for 20 min at 200° C.

Basic definitions — Algorithm

- Formally: a **finite** sequence of **well-defined** steps that consumes some **input** and produces some **output**.

Basic definitions — Algorithm

- Formally: a **finite** sequence of **well-defined** steps that consumes some **input** and produces some **output**.

Recipe to make bread (simplified):

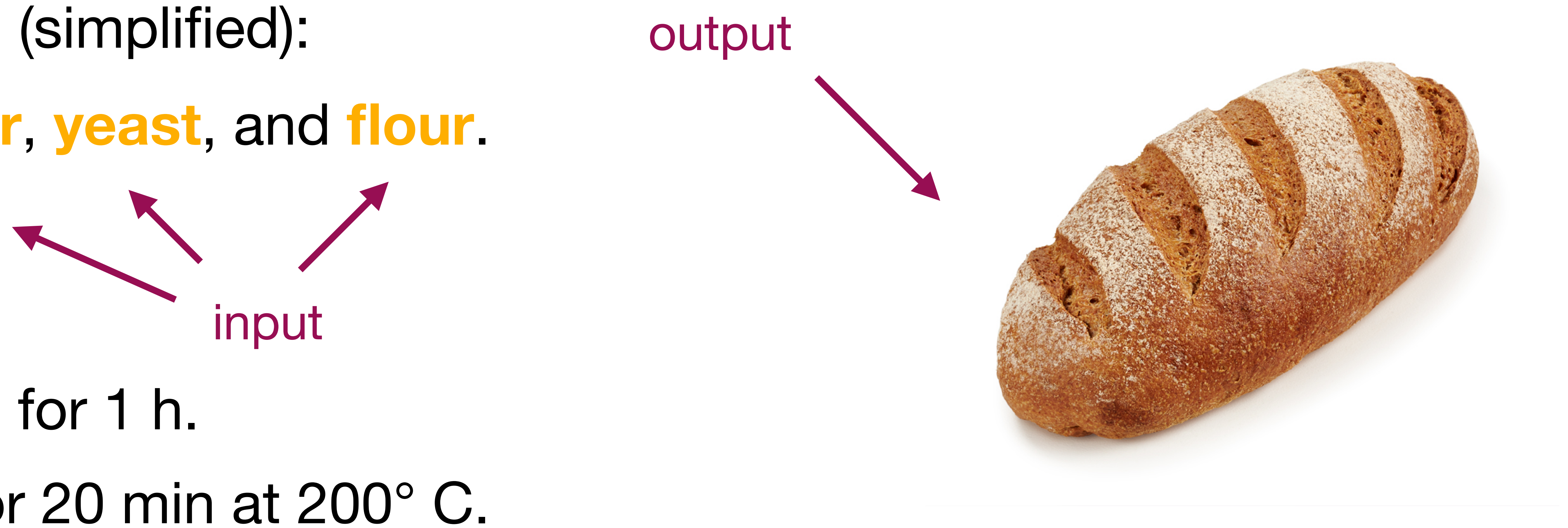
1. Stir together **water**, **yeast**, and **flour**.
2. Add **oil** and **salt**.
3. Knead the dough.
4. Let the dough rest for 1 h.
5. Bake the dough for 20 min at 200° C.

input

output

Basic definitions — Algorithm

- Formally: a **finite** sequence of **well-defined** steps that consumes some **input** and produces some **output**.



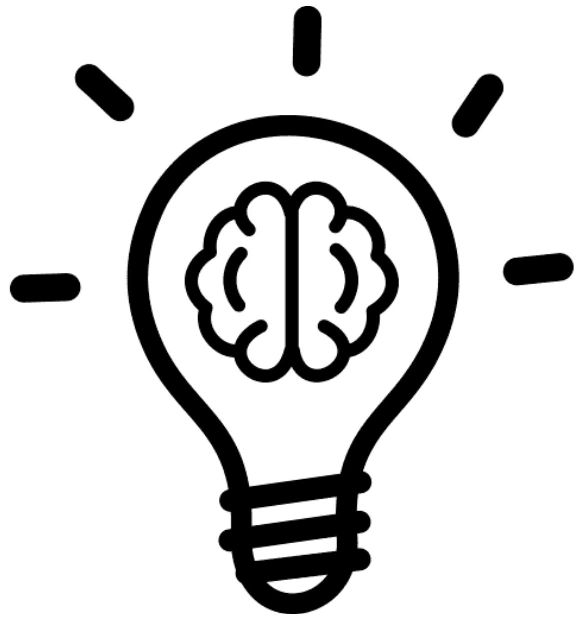
Recipe to make bread (simplified):

1. Stir together **water**, **yeast**, and **flour**.
2. Add **oil** and **salt**.
3. Knead the dough.
4. Let the dough rest for 1 h.
5. Bake the dough for 20 min at 200° C.

- In this lecture, we care about the algorithms that can be implemented on a **computer**.

Programming languages

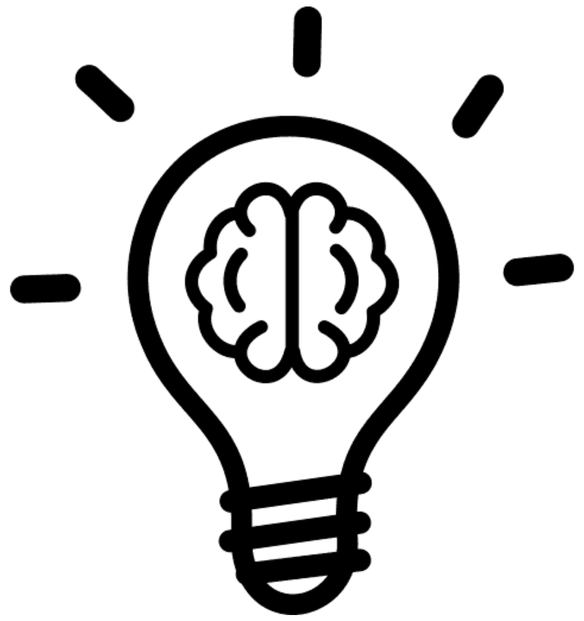
- **Implementation**: write the sequence of steps in a **programming language** (like C/C++, Java, Rust, Python, etc.) to let the algorithm be executed on a computer.



idea for a new algorithm

Programming languages

- **Implementation**: write the sequence of steps in a **programming language** (like C/C++, Java, Rust, Python, etc.) to let the algorithm be executed on a computer.



idea for a new algorithm

1

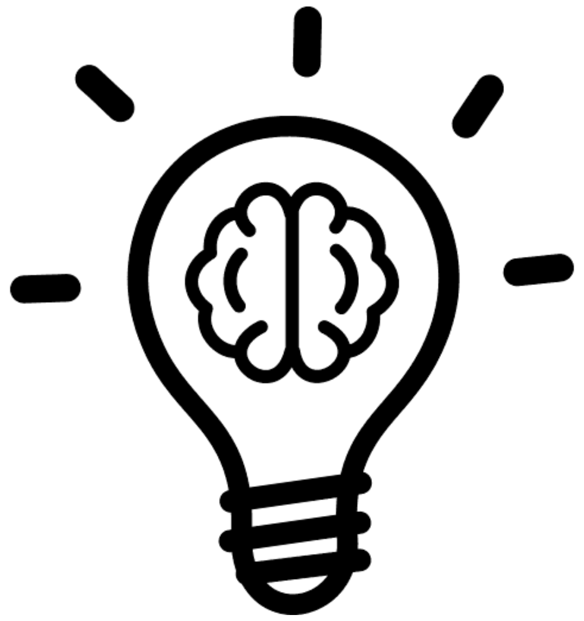
my_algorithm.cpp

```cpp
    for (; processed_buckets < num_non_empty_buckets; ++processed_buckets, ++buckets) {
        auto const& bucket = *buckets;
        assert(bucket.size() > 0);

        for (uint64_t pilot = 0; true; ++pilot) {
            uint64_t hashed_pilot = PTHASH_LIKELY(pilot < search_cache_size)
                                        ? hashed_pilots_cache[pilot]
                                        : default_hash64(pilot, seed);

            positions.clear();

            auto bucket_begin = bucket.begin(), bucket_end = bucket.end();
            for (; bucket_begin != bucket_end; ++bucket_begin) {
                uint64_t hash = *bucket_begin;
                uint64_t p = fastmod::fastmod_u64(hash ^ hashed_pilot, M, table_size);
                if (taken.get(p)) break;
                positions.push_back(p);
            }

            if (bucket_begin == bucket_end) {  // all keys do not have collisions with taken

                // check for in-bucket collisions
                std::sort(positions.begin(), positions.end());
                auto it = std::adjacent_find(positions.begin(), positions.end());
                if (it != positions.end())
                    continue;  // in-bucket collision detected, try next pilot

                pilots.emplace_back(bucket.id(), pilot);
                for (auto p : positions) {
                    assert(taken.get(p) == false);
                    taken.set(p, true);
                }
                if (config.verbose_output) log.update(processed_buckets, bucket.size(), pilot);
                break;
            }
        }
    }
```

code

Programming languages

- **Implementation**: write the sequence of steps in a **programming language** (like C/C++, Java, Rust, Python, etc.) to let the algorithm be executed on a computer.



idea for a new algorithm

1

written in the C++ programming language

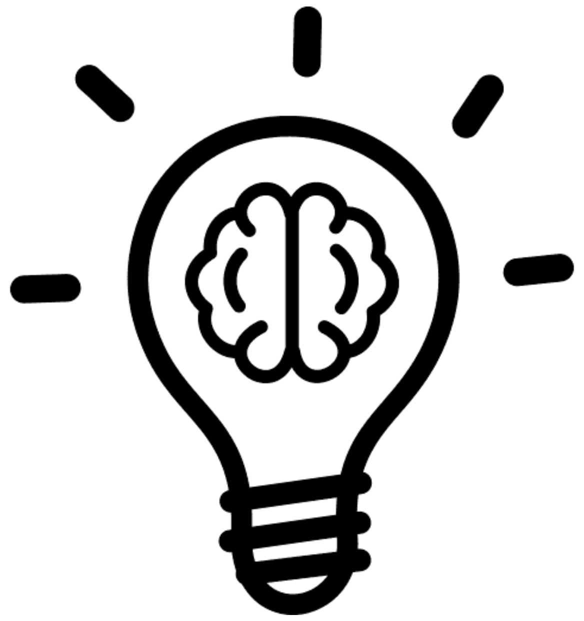
`my_algorithm.cpp`

```cpp
    for (; processed_buckets < num_non_empty_buckets; ++processed_buckets, ++buckets) {
        auto const& bucket = *buckets;
        assert(bucket.size() > 0);

        for (uint64_t pilot = 0; true; ++pilot) {
            uint64_t hashed_pilot = PTHASH_LIKELY(pilot < search_cache_size)
                                        ? hashed_pilots_cache[pilot]
                                        : default_hash64(pilot, seed);

            positions.clear();

            auto bucket_begin = bucket.begin(), bucket_end = bucket.end();
            for (; bucket_begin != bucket_end; ++bucket_begin) {
                uint64_t hash = *bucket_begin;
                uint64_t p = fastmod::fastmod_u64(hash ^ hashed_pilot, M, table_size);
                if (taken.get(p)) break;
                positions.push_back(p);
            }

            if (bucket_begin == bucket_end) {  // all keys do not have collisions with taken

                // check for in-bucket collisions
                std::sort(positions.begin(), positions.end());
                auto it = std::adjacent_find(positions.begin(), positions.end());
                if (it != positions.end())
                    continue;  // in-bucket collision detected, try next pilot

                pilots.emplace_back(bucket.id(), pilot);
                for (auto p : positions) {
                    assert(taken.get(p) == false);
                    taken.set(p, true);
                }
                if (config.verbose_output) log.update(processed_buckets, bucket.size(), pilot);
                break;
            }
        }
    }
```

code

Programming languages

- **Implementation**: write the sequence of steps in a **programming language** (like C/C++, Java, Rust, Python, etc.) to let the algorithm be executed on a computer.



idea for a new algorithm

1

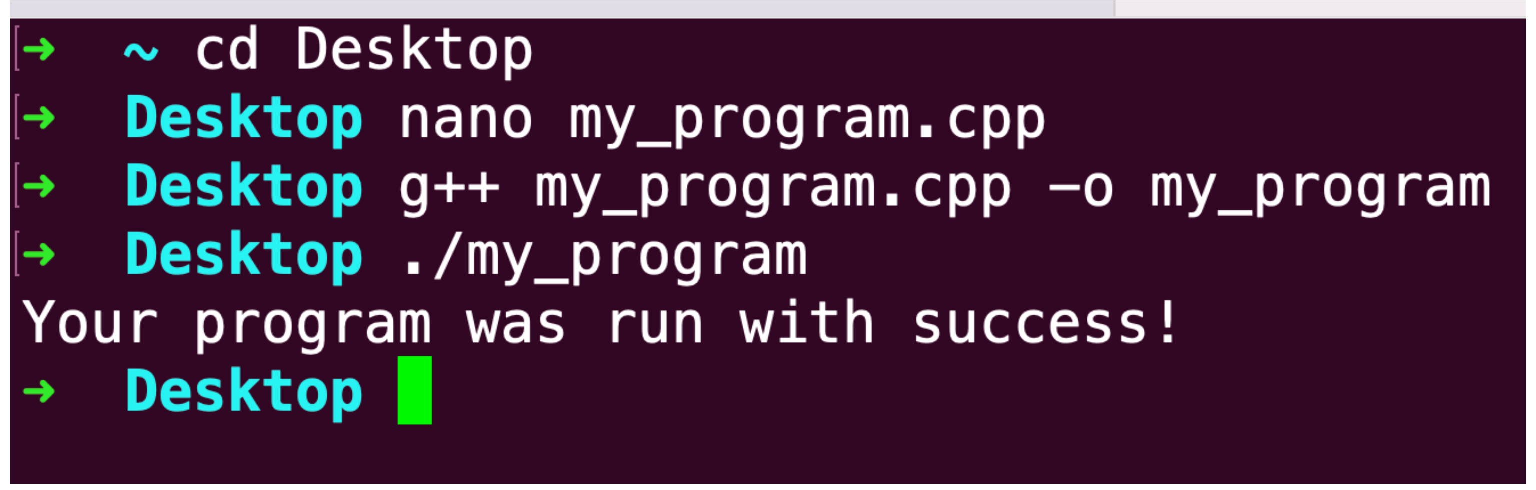
written in the C++ programming language

`my_algorithm.cpp`

```cpp
    for (; processed_buckets < num_non_empty_buckets; ++processed_buckets, ++buckets) {
        auto const& bucket = *buckets;
        assert(bucket.size() > 0);

        for (uint64_t pilot = 0; true; ++pilot) {
            uint64_t hashed_pilot = PTHASH_LIKELY(pilot < search_cache_size)
                                        ? hashed_pilots_cache[pilot]
                                        : default_hash64(pilot, seed);

            positions.clear();

            auto bucket_begin = bucket.begin(), bucket_end = bucket.end();
            for (; bucket_begin != bucket_end; ++bucket_begin) {
                uint64_t hash = *bucket_begin;
                uint64_t p = fastmod::fastmod_u64(hash ^ hashed_pilot, M, table_size);
                if (taken.get(p)) break;
                positions.push_back(p);
            }

            if (bucket_begin == bucket_end) {  // all keys do not have collisions with taken

                // check for in-bucket collisions
                std::sort(positions.begin(), positions.end());
                auto it = std::adjacent_find(positions.begin(), positions.end());
                if (it != positions.end())
                    continue;  // in-bucket collision detected, try next pilot

                pilots.emplace_back(bucket.id(), pilot);
                for (auto p : positions) {
                    assert(taken.get(p) == false);
                    taken.set(p, true);
                }
                if (config.verbose_output) log.update(processed_buckets, bucket.size(), pilot);
                break;
            }
        }
    }
```

code

2



```
➜  ~ cd Desktop
➜  Desktop nano my_program.cpp
➜  Desktop g++ my_program.cpp -o my_program
➜  Desktop ./my_program
Your program was run with success!
➜  Desktop 
```

result

Basic definitions — Data Structure

- **Data Structures** store the data that is accessed by an algorithm.
- Idea: the algorithm can read/write the data from/to a data structure to solve the problem **faster**.

Basic definitions — Data Structure

- **Data Structures** store the data that is accessed by an algorithm.
- Idea: the algorithm can read/write the data from/to a data structure to solve the problem **faster**.

- Let's introduce the most basic data structure in all Computer Science: the **array** — a sequence of items all of the same type.
- For example, a sequence of integer numbers, or a sequence of characters.

```
N = [1,4,5,13,23,0,-9,34]
     1 2 3  4  5 6  7  8

S = ['p','a','c','i','n','o','t','t','i']
      1   2   3   4   5   6   7   8   9
```

Basic definitions — Arrays

```
N = [1,4,5,13,23,0,-9,34]
     1 2 3  4  5 6  7  8

S = ['p','a','c','i','n','o','t','t','i']
      1   2   3   4   5   6   7   8   9
```

- **Notation.** With `|A|` we indicate the number of items in the array `A` (its length) and with `A[i]` the `i`-th item of the array, for all `i=1..|A|`.
- For example, `N[3]` is the integer number 5 and `S[7]` is the character `'t'`.

Basic definitions — Arrays

```
N = [1,4,5,13,23,0,-9,34]
     1 2 3  4  5 6  7  8

S = ['p','a','c','i','n','o','t','t','i']
      1   2   3   4   5   6   7   8   9
```

- **Notation.** With `|A|` we indicate the number of items in the array `A` (its length) and with `A[i]` the `i`-th item of the array, for all `i=1..|A|`.
- For example, `N[3]` is the integer number 5 and `S[7]` is the character `'t'`.
- If we do `S[1]='t'`, then we over-write the first character of `S`, so that now `S` is
  `S = ['t','a','c','i','n','o','t','t','i'].`
- If we do `N[4]+=3`, now `N[4]` is equal to `13+3=16.`

Basic definitions — Arrays

```
N = [1,4,5,13,23,0,-9,34]
     1 2 3  4  5 6  7  8

S = ['p','a','c','i','n','o','t','t','i']
      1   2   3   4   5   6   7   8   9
```

- **Notation.** With `|A|` we indicate the number of items in the array `A` (its length) and with `A[i]` the `i`-th item of the array, for all `i=1..|A|`.

- For example, `N[3]` is the integer number 5 and `S[7]` is the character `'t'`.

- If we do `S[1]='t'`, then we over-write the first character of `S`, so that now `S` is
  `S = ['t','a','c','i','n','o','t','t','i'].`

- If we do `N[4]+=3`, now `N[4]` is equal to `13+3=16.`

- **Important note:** `i` must be an integer. It does not make any sense to refer to the element in position `i=3.56`...

Basic definitions — Arrays and memory

- In practice, an array is stored in the memory of your computer as a contiguous sequence of *bytes*.

- The "byte" is the smallest unit of memory on a computer and corresponds to a group of 8 *bits* — 8 binary digits.

- For example, these are 3 bytes.

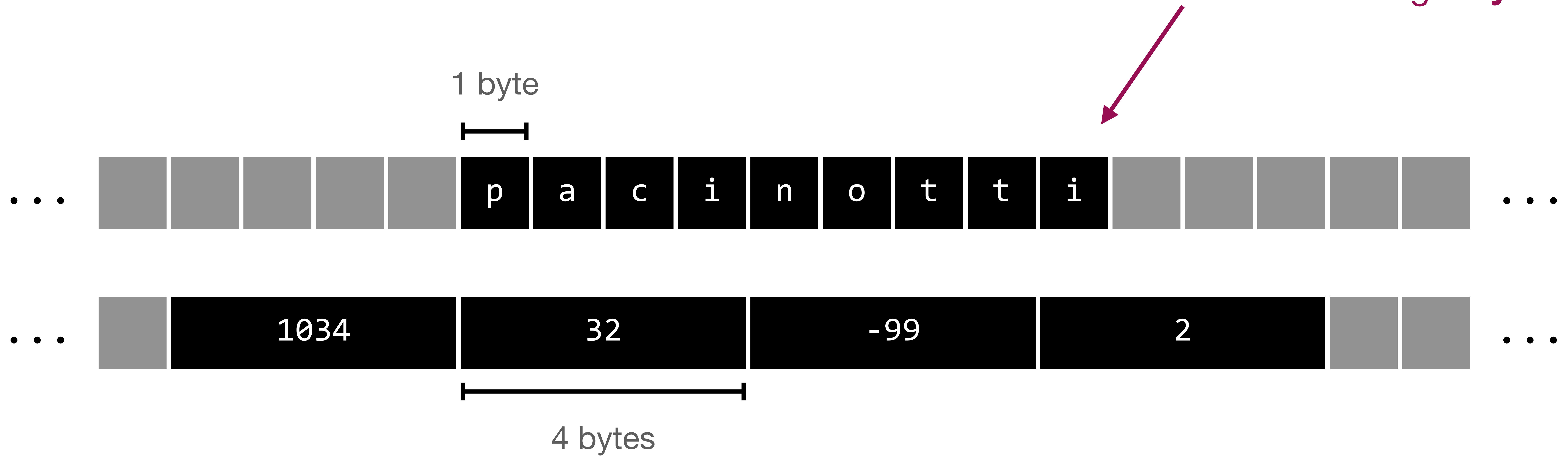
```
01001011   11100010   01010110
```

Basic definitions — Arrays and memory

- In practice, an array is stored in the memory of your computer as a contiguous sequence of *bytes*.
- The "byte" is the smallest unit of memory on a computer and corresponds to a group of 8 *bits* — 8 binary digits.
- For example, these are 3 bytes.

```
01001011   11100010   01010110
```

computer memory abstraction:
a sequence of memory cells,
each holding **1 byte**



Warm up — Counting occurrences

- **Problem 1.** Suppose we have a string `S = "abracadabraabracaba"` (an array of characters) and we want to count the **number of occurrences** of a given character `x` (which can be any character, like `a`, `b`, `c`, etc.).
- How would you do it by hand?

Warm up — Counting occurrences

- **Problem 1.** Suppose we have a string `S = "abracadabraabracaba"` (an array of characters) and we want to count the **number of occurrences** of a given character `x` (which can be any character, like `a`, `b`, `c`, etc.).
- How would you do it by hand?
- Easy with a small string. What if the string is 1 billion (i.e., 1,000,000,000) characters? We need an algorithm to do the task for us!

Warm up — Counting occurrences

- **Problem 1.** Suppose we have a string `S = "abracadabraabracaba"` (an array of characters) and we want to count the **number of occurrences** of a given character `x` (which can be any character, like `a`, `b`, `c`, etc.).
- How would you do it by hand?
- Easy with a small string. What if the string is 1 billion (i.e., 1,000,000,000) characters? We need an algorithm to do the task for us!
- Our method: "*For each character of* `S`*, check if it is equal to* `x`*: if so, we have found an occurrence of* `x`*.*"
- **Input**: the string `S`.
- **Output**: an integer number, indicating the number of occurrences of the character `x`. (For example, we expect the answer to be 4 for `x='b'`.)

Warm up — Counting occurrences

```
occ_count(S,x):
  count = 0
  for i = 1..|S|:
    if S[i] is equal to x:
      count += 1
  return count
```

↔

" *For each character of* `S`*, check if it is equal to* `x`*: if so, we have found an occurrence of* `x`*.* "

Warm up — Counting occurrences

```
occ_count(S,x):
  count = 0
  for i = 1..|S|:
    if S[i] is equal to x:
      count += 1
  return count
```

" For each character of `S`*, check if it is equal to* `x`*: if so, we have found an occurrence of* `x`*. "*

```
 x = 'b'

 S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
 i →   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
       ↑

count  0
```

Warm up — Counting occurrences

```
occ_count(S,x):
    count = 0
    for i = 1..|S|:
      if S[i] is equal to x:
        count += 1
    return count
```

↔

" *For each character of* `S`*, check if it is equal to* `x`*: if so, we have found an occurrence of* `x`*.* "

```
 x = 'b'

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑

count 0   1
```

Warm up — Counting occurrences

```
occ_count(S,x):
    count = 0
    for i = 1..|S|:
      if S[i] is equal to x:
        count += 1
    return count
```

↔

" *For each character of* `S`*, check if it is equal to* `x`*: if so, we have found an occurrence of* `x`*.* "

```
 x = 'b'

 S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
 i →  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19

      ↑   ↑   ↑

count 0   1   1
```

Warm up — Counting occurrences

```
occ_count(S,x):
count = 0
for i = 1..|S|:
  if S[i] is equal to x:
    count += 1
return count
```

↔

" For each character of `S`*, check if it is equal to* `x`*: if so, we have found an occurrence of* `x`*. "*

```
x = 'b'

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →  1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19

     ↑   ↑   ↑   ↑

count 0   1   1   1
```

Warm up — Counting occurrences

```
occ_count(S,x):
  count = 0
  for i = 1..|S|:
    if S[i] is equal to x:
      count += 1
  return count
```

" For each character of `S`, check if it is equal to `x`: if so, we have found an occurrence of `x`. "

```
 x = 'b'

 S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
 i →  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19

      ↑   ↑   ↑   ↑   ↑

count 0   1   1   1   1
```

Warm up — Counting occurrences

```
occ_count(S,x):
  count = 0
  for i = 1..|S|:
    if S[i] is equal to x:
      count += 1
  return count
```

↔

" For each character of `S`*, check if it is equal to* `x`*: if so, we have found an occurrence of* `x`*. "*

```
x = 'b'

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19

      ↑   ↑   ↑   ↑   ↑   ↑

count 0   1   1   1   1   1
```

Warm up — Counting occurrences

```
occ_count(S,x):
count = 0
for i = 1..|S|:
  if S[i] is equal to x:
    count += 1
return count
```

" For each character of `S`*, check if it is equal to* `x`*: if so, we have found an occurrence of* `x`*. "*

x = 'b'

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']

i →	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	↑	↑	↑	↑	↑	↑	↑												
count	0	1	1	1	1	1	1												

Warm up — Counting occurrences

```
occ_count(S,x):
count = 0
for i = 1..|S|:
  if S[i] is equal to x:
    count += 1
return count
```

" *For each character of* `S`*, check if it is equal to* `x`*: if so, we have found an occurrence of* `x`*.* "

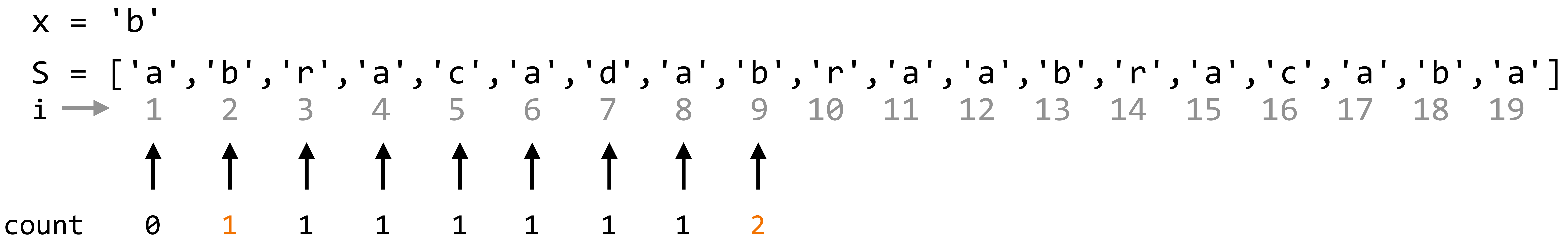
```
x = 'b'

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19

      ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑

count 0   1   1   1   1   1   1   1
```

Warm up — Counting occurrences

```
occ_count(S,x):
count = 0
for i = 1..|S|:
  if S[i] is equal to x:
    count += 1
return count
```

↔

" *For each character of* `S`*, check if it is equal to* `x`*: if so, we have found an occurrence of* `x`*.* "



```
x = 'b'

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →  1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19

     ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑

count 0  1   1   1   1   1   1   1   2
```

Warm up — Counting occurrences

```
occ_count(S,x):
count = 0
for i = 1..|S|:
  if S[i] is equal to x:
    count += 1
return count
```

↔

" For each character of `S`, check if it is equal to `x`: if so, we have found an occurrence of `x`. "

`x = 'b'`

`S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']`

i →	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑									
count	0	1	1	1	1	1	1	1	2	2									

Warm up — Counting occurrences

```
occ_count(S,x):
count = 0
for i = 1..|S|:
  if S[i] is equal to x:
    count += 1
return count
```

↔

" For each character of `S`*, check if it is equal to* `x`*: if so, we have found an occurrence of* `x`*. "*

`x = 'b'`

`S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']`

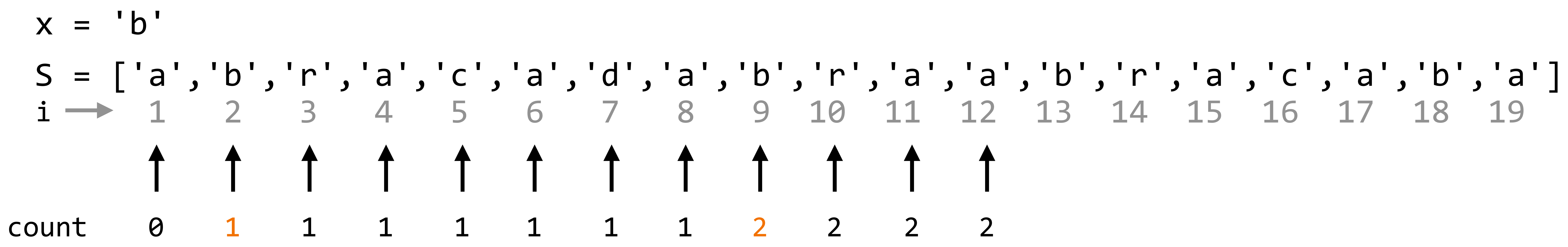
i →	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑								
count	0	1	1	1	1	1	1	1	2	2	2								

Warm up — Counting occurrences

```
occ_count(S,x):
    count = 0
    for i = 1..|S|:
      if S[i] is equal to x:
        count += 1
    return count
```

↔

" For each character of S*, check if it is equal to* x*: if so, we have found an occurrence of* x*. "*

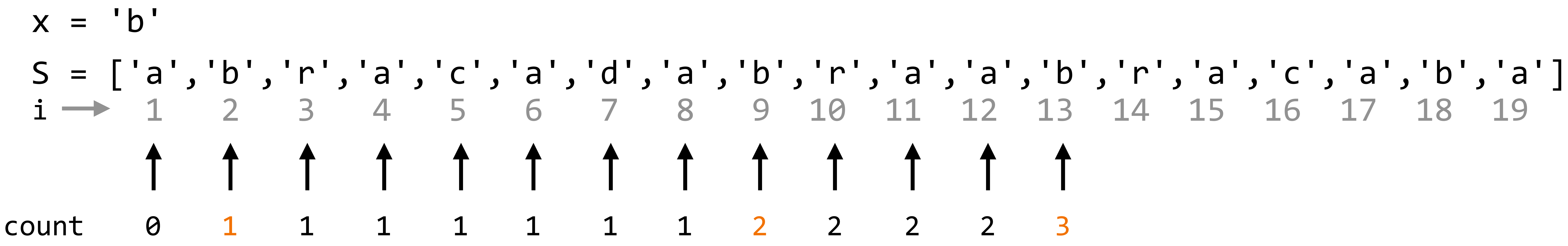


```
x = 'b'

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19

count 0   1   1   1   1   1   1   1   2   2   2   2
```

Warm up — Counting occurrences

```
occ_count(S,x):
  count = 0
  for i = 1..|S|:
    if S[i] is equal to x:
      count += 1
  return count
```

" For each character of `S`*, check if it is equal to* `x`*: if so, we have found an occurrence of* `x`*. "*



```
x = 'b'

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →  1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19

count  0   1   1   1   1   1   1   1   2   2   2   2   3
```

Warm up — Counting occurrences

```
occ_count(S,x):
  count = 0
  for i = 1..|S|:
    if S[i] is equal to x:
      count += 1
  return count
```

↔

" For each character of `S`*, check if it is equal to* `x`*: if so, we have found an occurrence of* `x`*. "*

x = 'b'

S =	'a'	'b'	'r'	'a'	'c'	'a'	'd'	'a'	'b'	'r'	'a'	'a'	'b'	'r'	'a'	'c'	'a'	'b'	'a'
i →	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑					
count	0	1	1	1	1	1	1	1	2	2	2	2	3	3					

Warm up — Counting occurrences

```
occ_count(S,x):
  count = 0
  for i = 1..|S|:
    if S[i] is equal to x:
      count += 1
  return count
```

" For each character of `S`*, check if it is equal to* `x`*: if so, we have found an occurrence of* `x`*. "*

`x = 'b'`

`S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']`

i →	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
count	0	1	1	1	1	1	1	1	2	2	2	2	3	3	3				

Warm up — Counting occurrences

```
occ_count(S,x):
count = 0
for i = 1..|S|:
  if S[i] is equal to x:
    count += 1
return count
```

" *For each character of* `S`*, check if it is equal to* `x`*: if so, we have found an occurrence of* `x`*.* "

```
x = 'b'

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
```

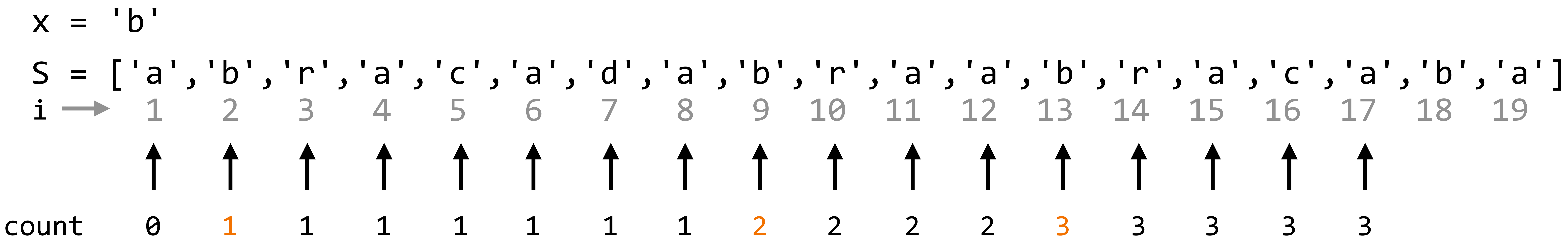
i →	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
count	0	1	1	1	1	1	1	1	2	2	2	2	3	3	3	3			

Warm up — Counting occurrences

```
occ_count(S,x):
count = 0
for i = 1..|S|:
  if S[i] is equal to x:
    count += 1
return count
```

↔

" *For each character of `S`, check if it is equal to `x`: if so, we have found an occurrence of `x`.* "



x = 'b'

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']

i →	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
count	0	1	1	1	1	1	1	1	2	2	2	2	3	3	3	3	3		

Warm up — Counting occurrences

```
occ_count(S,x):
count = 0
for i = 1..|S|:
  if S[i] is equal to x:
    count += 1
return count
```

↔

" *For each character of* `S`*, check if it is equal to* `x`*: if so, we have found an occurrence of* `x`*.* "

```
x = 'b'

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
```

i →	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
count	0	1	1	1	1	1	1	1	2	2	2	2	3	3	3	3	3	4	

Warm up — Counting occurrences

```
occ_count(S,x):
    count = 0
    for i = 1..|S|:
      if S[i] is equal to x:
        count += 1
    return count
```

↔

" *For each character of* `S`*, check if it is equal to* `x`*: if so, we have found an occurrence of* `x`*.* "

```
x = 'b'

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
```

i →	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
count	0	1	1	1	1	1	1	1	2	2	2	2	3	3	3	3	3	4	4

Warm up — Counting occurrences

- **Problem 2.** Suppose we have a string `S = "abracadabraabracaba"` (an array of characters) and we want to count the number of occurrences **of each character** appearing in the string.
- **Input**: the string S.
- **Output**: `('a',9) ('b',4) ('c',2) ('d',1) ('r',3)`.

Warm up — Counting occurrences

- **Problem 2.** Suppose we have a string `S = "abracadabraabracaba"` (an array of characters) and we want to count the number of occurrences **of each character** appearing in the string.

- **Input**: the string S.

- **Output**: `('a',9) ('b',4) ('c',2) ('d',1) ('r',3)`.

- **Idea 1**: use the previous `occ_count(S,x)` algorithm.

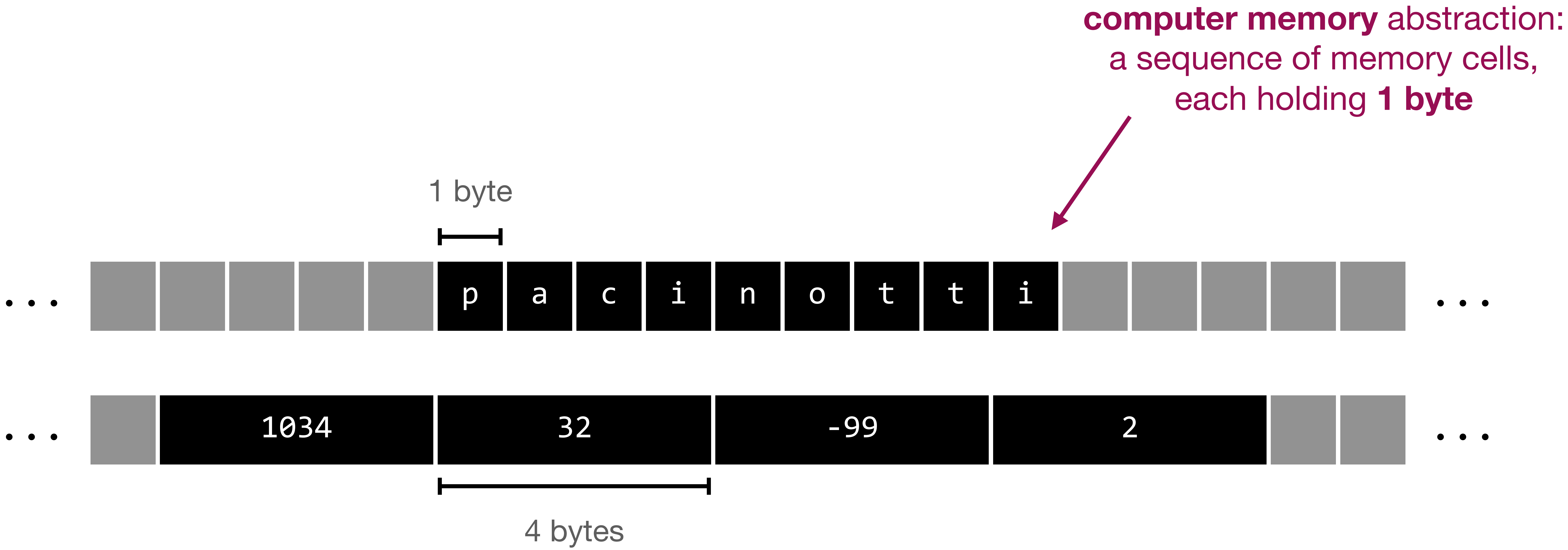
```
all_occ_count_v1(S):
  for each character x in ['a','b','c','d','e','f',...,'z']:
    occ = occ_count(S,x)
    print(x,occ)
```

Warm up — Counting occurrences

- **Idea 2**: exploit the fact that each character is actually a small integer (1 byte = 8 bits).

Warm up — Counting occurrences

- **Idea 2**: exploit the fact that each character is actually a small integer (1 byte = 8 bits).



computer memory abstraction:
a sequence of memory cells,
each holding **1 byte**

Warm up — Counting occurrences

- **Idea 2**: exploit the fact that each character is actually a small integer (1 byte = 8 bits).
- How many distinct integers can we represent with 8 bits?

Warm up — Counting occurrences

- **Idea 2**: exploit the fact that each character is actually a small integer (1 byte = 8 bits).
- How many distinct integers can we represent with 8 bits?
    - With 1 bit: either `0` or `1`. (2 integers)

Warm up — Counting occurrences

- **Idea 2**: exploit the fact that each character is actually a small integer (1 byte = 8 bits).
- How many distinct integers can we represent with 8 bits?
  - With 1 bit: either `0` or `1`. (2 integers)
  - With 2 bits: `00`, `01`, `10`, `11`. (4 integers)

Warm up — Counting occurrences

- **Idea 2**: exploit the fact that each character is actually a small integer (1 byte = 8 bits).
- How many distinct integers can we represent with 8 bits?
    - With 1 bit: either `0` or `1`. (2 integers)
    - With 2 bits: `00`, `01`, `10`, `11`. (4 integers)
    - With 3 bits: `000`, `001`, `010`, `011`, `100`, `101`, `110`, `111`. (8 integers)
    - …
    - With 8 bits: $2^8 = 256$ integers.

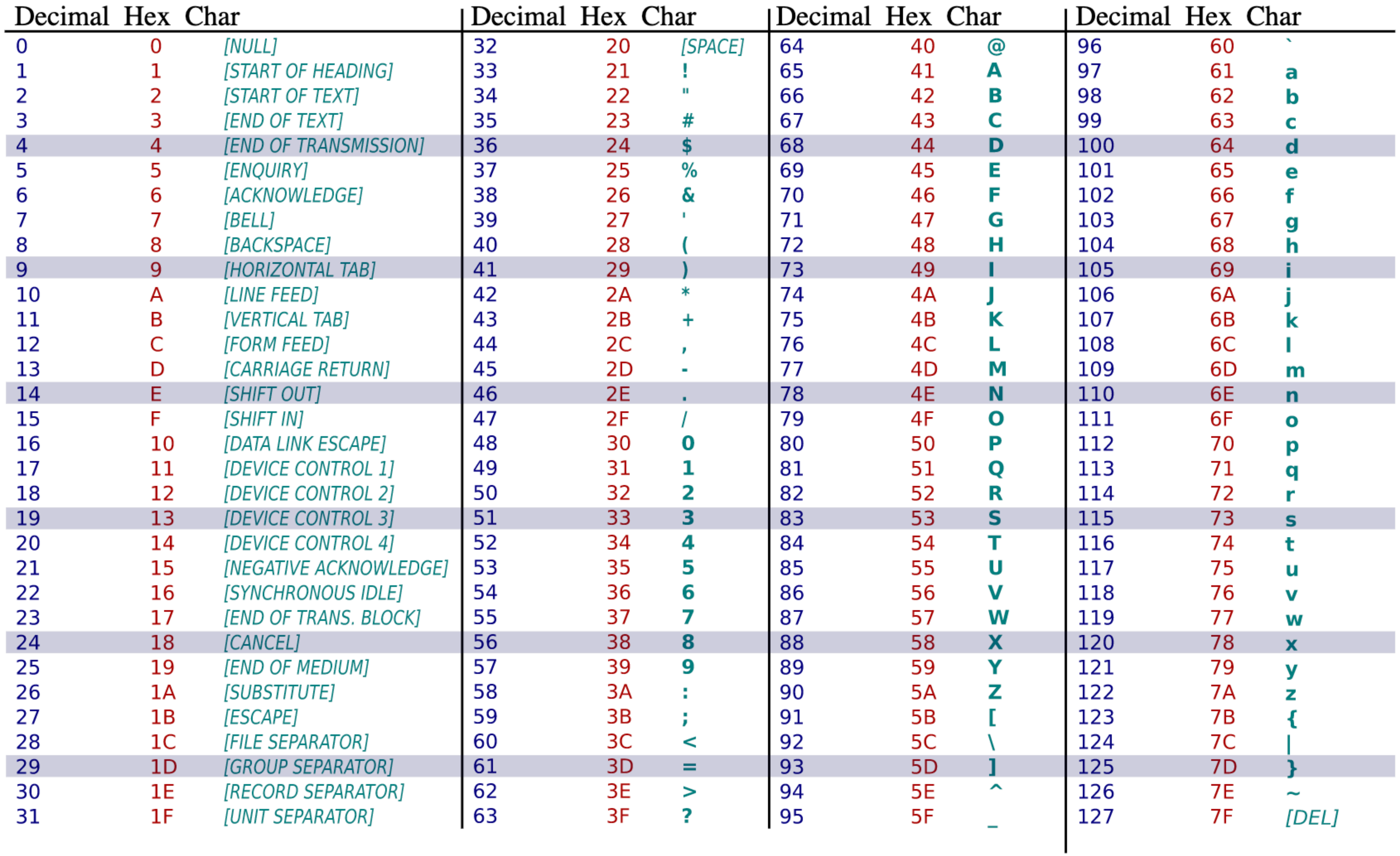
Warm up — Counting occurrences

- **Idea 2**: exploit the fact that each character is actually a small integer (1 byte = 8 bits).
- How many distinct integers can we represent with 8 bits?
    - With 1 bit: either `0` or `1`. (2 integers)
    - With 2 bits: `00`, `01`, `10`, `11`. (4 integers)
    - With 3 bits: `000`, `001`, `010`, `011`, `100`, `101`, `110`, `111`. (8 integers)
    - …
    - With 8 bits: $2^8 = 256$ integers.
- A character, when interpreted as an integer, can therefore be used **as an index** into an array of length 256. This is known as the **ASCII** representation.

Warm up — Counting occurrences

- **Idea 2**: exploit the fact that each character is actually a small integer (1 byte = 8 bits).

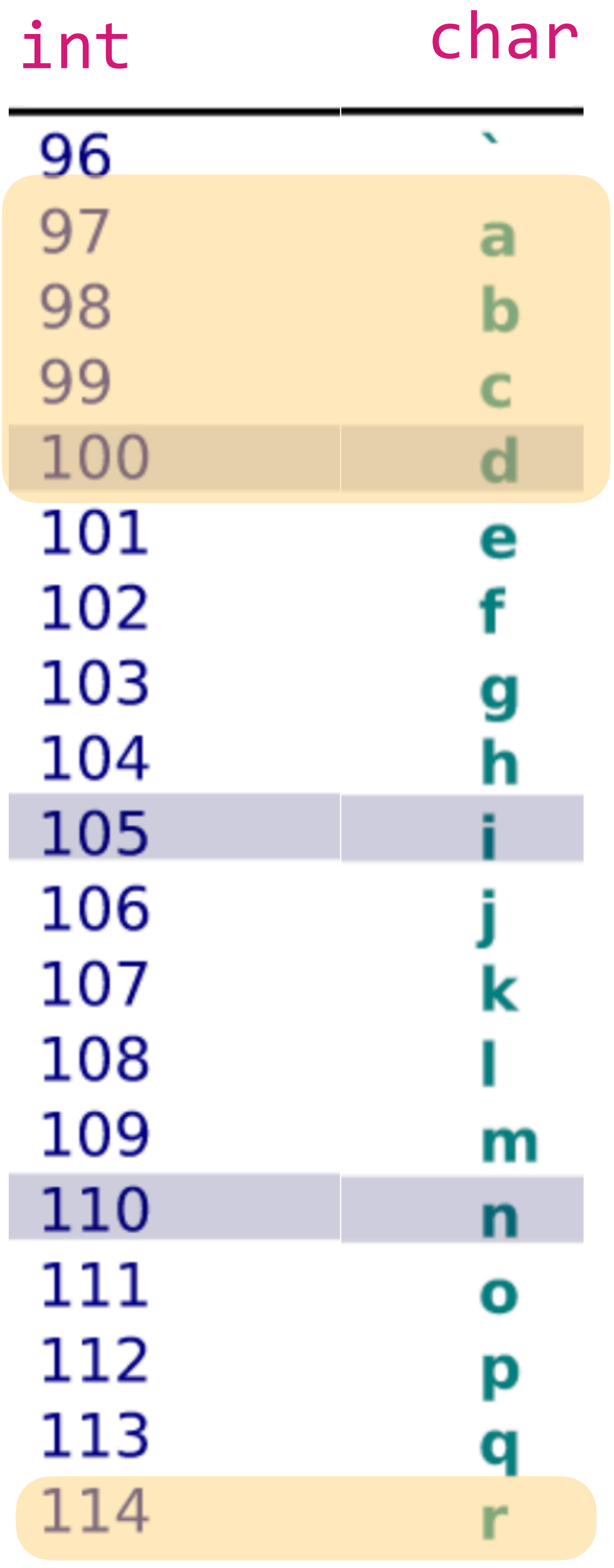
ASCII table



Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
      j = int(S[i])
      C[j] += 1
    for i = 1..|C|:
      print(char(i),C[i])
```



int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 0 ,  0 ,  0 ,   0 , ... ,   0 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...
```

```
S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 0 ,  0 ,  0 ,   0 , ... ,   0 , ... ]
     0  1  ...   97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

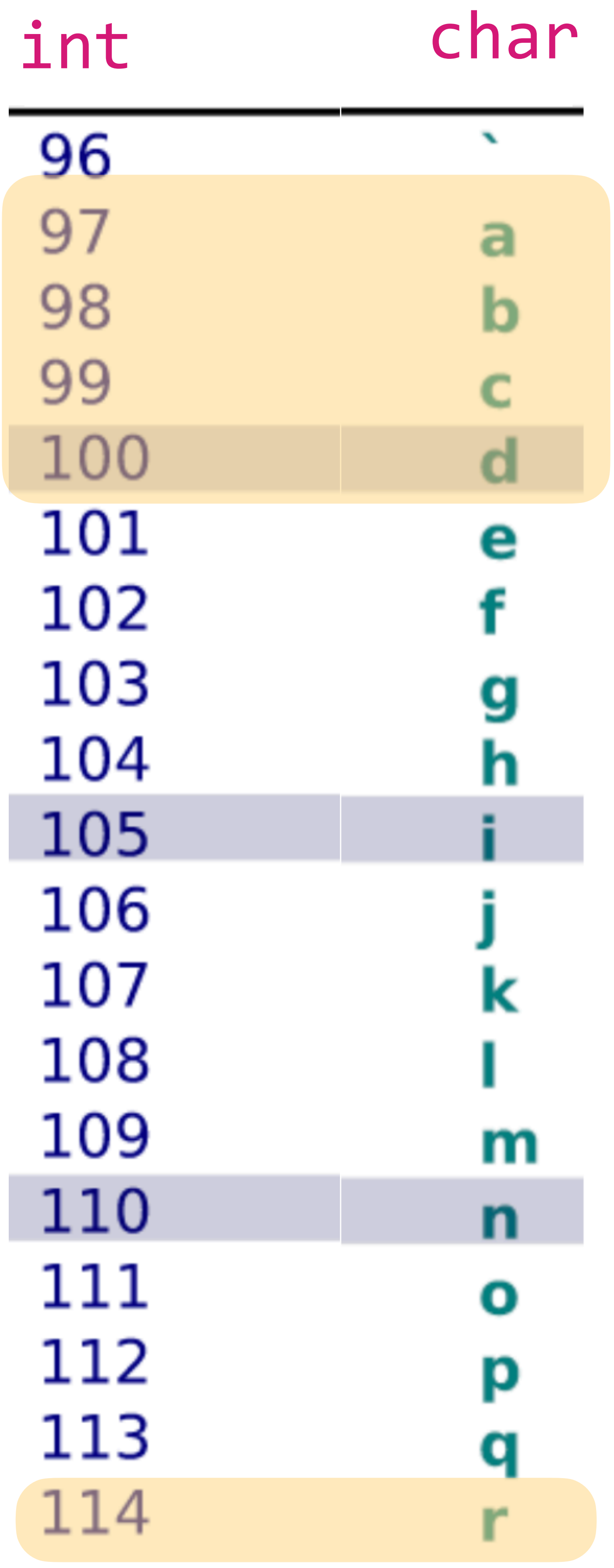
```
C = [0, 0, ... , 1 ,  0 ,  0 ,   0 , ... ,   0 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

```
C = [0, 0, ... , 1 ,  0 ,  0 ,   0 , ... ,   0 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...
```

```
S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑
```



int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 1 ,  1 ,  0 ,   0 , ... ,   0 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

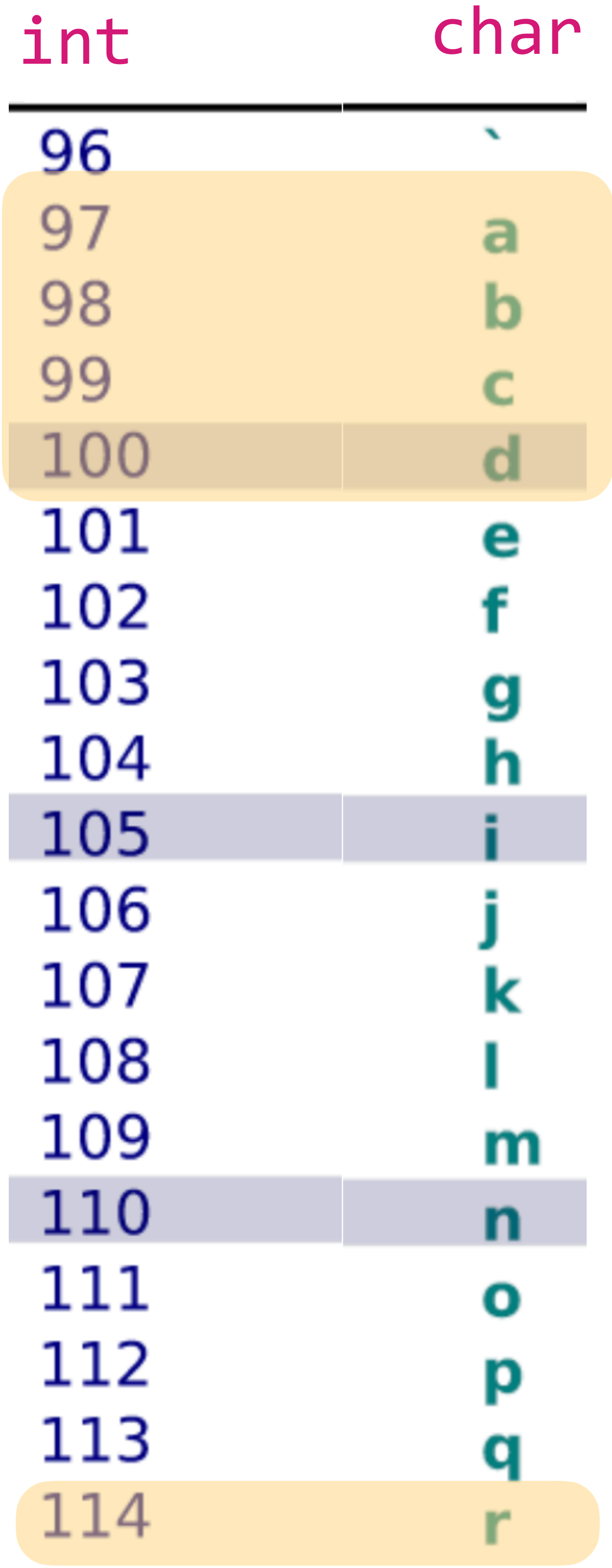
int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 1 ,  1 ,  0 ,   0 , ... ,   0 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...
```

```
S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```



int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 1 , 1 , 0 ,  0 , ... ,  1 , ... ]
     0  1  ...  97  98  99  100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 1 ,  1 ,  0 ,   0 , ... ,   1 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
  C[1..256] = [0,0,...,0]
  for i = 1..|S|:
    j = int(S[i])
    C[j] += 1
  for i = 1..|C|:
    print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 2 ,  1 ,  0 ,   0 , ... ,   1 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...
```

```
S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 2 ,  1 ,  0 ,   0 , ... ,   1 , ... ]
     0  1  ...   97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i ⟶   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
  C[1..256] = [0,0,...,0]
  for i = 1..|S|:
    j = int(S[i])
    C[j] += 1
  for i = 1..|C|:
    print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 2 ,  1 ,  1 ,   0 , ... ,   1 , ... ]
     0  1  ...   97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 2 , 1 , 1 ,  0 , ... ,  1 , ... ]
     0  1  ...  97  98  99  100   ...  114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

```
C = [0, 0, ... , 3 ,  1 ,  1 ,   0 , ... ,   1 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑   ↑   ↑   ↑
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

Warm up — Counting occurrences

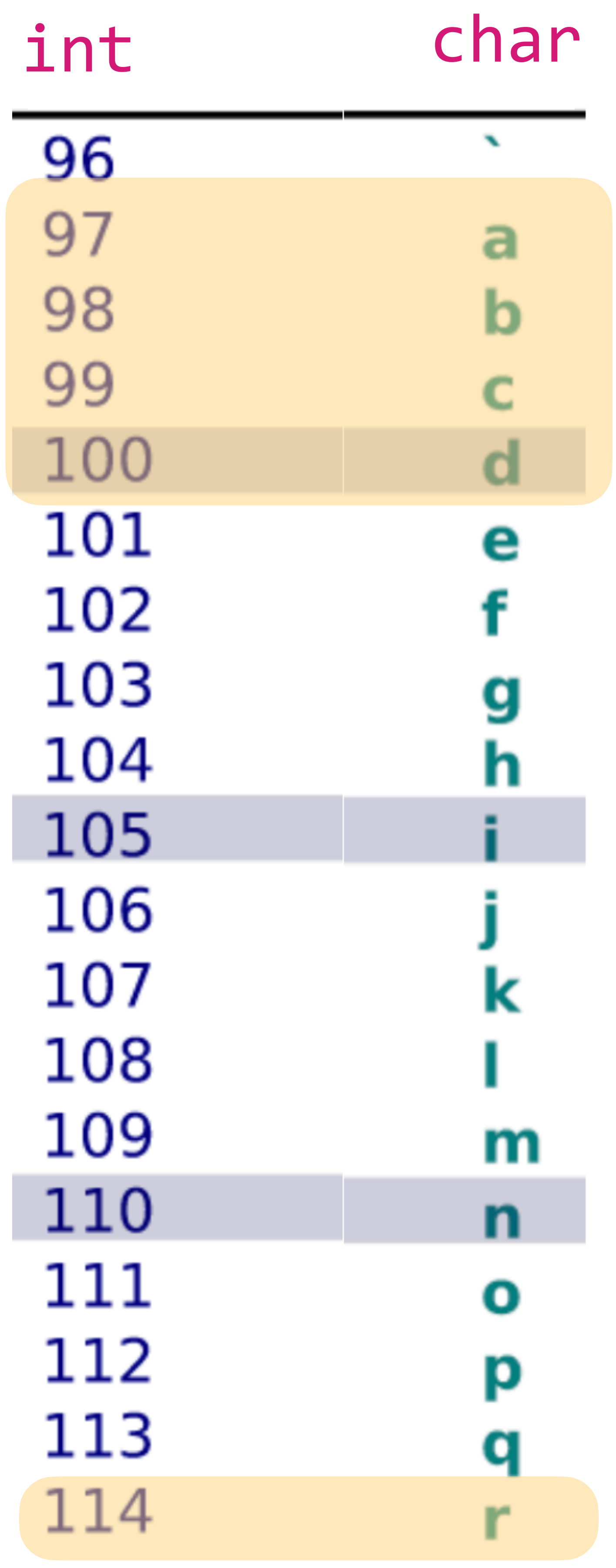
```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 3 ,  1 ,  1 ,   0 , ... ,   1 , ... ]
     0  1  ...   97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

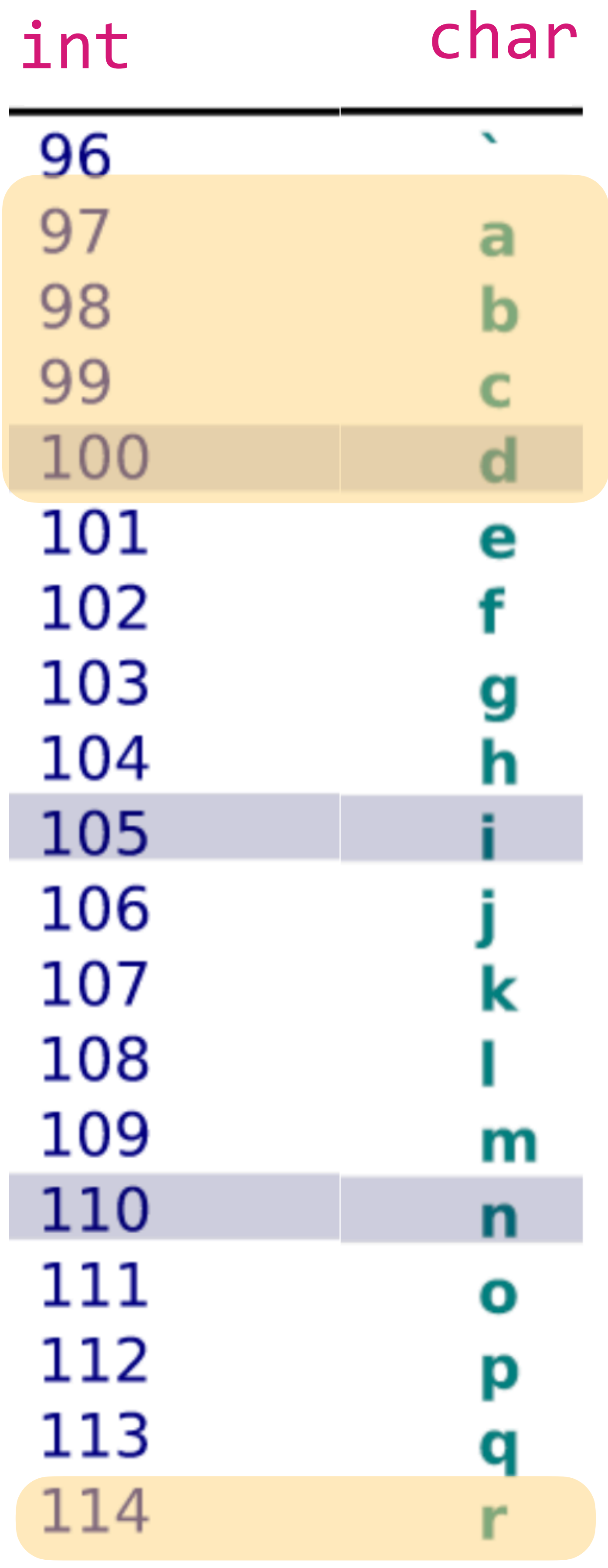


int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 3 ,  1 ,  1 ,   1 , ... ,   1 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19

      ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```



int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 3 ,  1 ,  1 ,   1 , ... ,   1 , ... ]
     0  1  ...   97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
  C[1..256] = [0,0,...,0]
  for i = 1..|S|:
    j = int(S[i])
    C[j] += 1
  for i = 1..|C|:
    print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 4 ,  1 ,  1 ,   1 , ... ,   1 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...
```

```
S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 4 ,  1 ,  1 ,   1 , ... ,   1 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
      j = int(S[i])
      C[j] += 1
    for i = 1..|C|:
      print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 4 ,  2 ,  1 ,   1 , ... ,   1 , ... ]
     0  1  ...   97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i ⟶   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 4 ,  2 ,  1 ,   1 , ... ,   1 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...
```

```
S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 4 ,  2 ,  1 ,   1 , ... ,   2 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 4 ,  2 ,  1 ,   1 , ... ,   2 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19

      ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

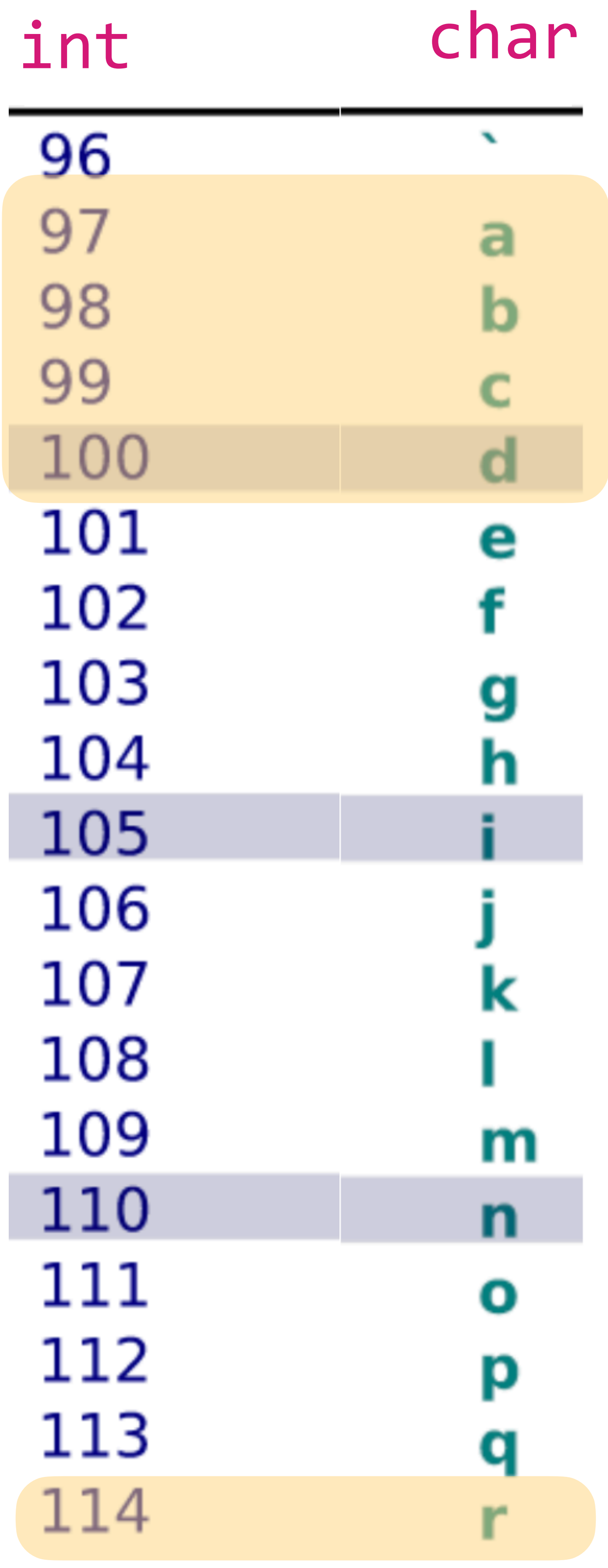
```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 5 ,  2 ,  1 ,   1 , ... ,   2 , ... ]
     0  1  ...   97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```



int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 5 ,  2 ,  1 ,   1 , ... ,   2 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 6 ,  2 ,  1 ,   1 , ... ,   2 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →  1   2   3   5   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19
     ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

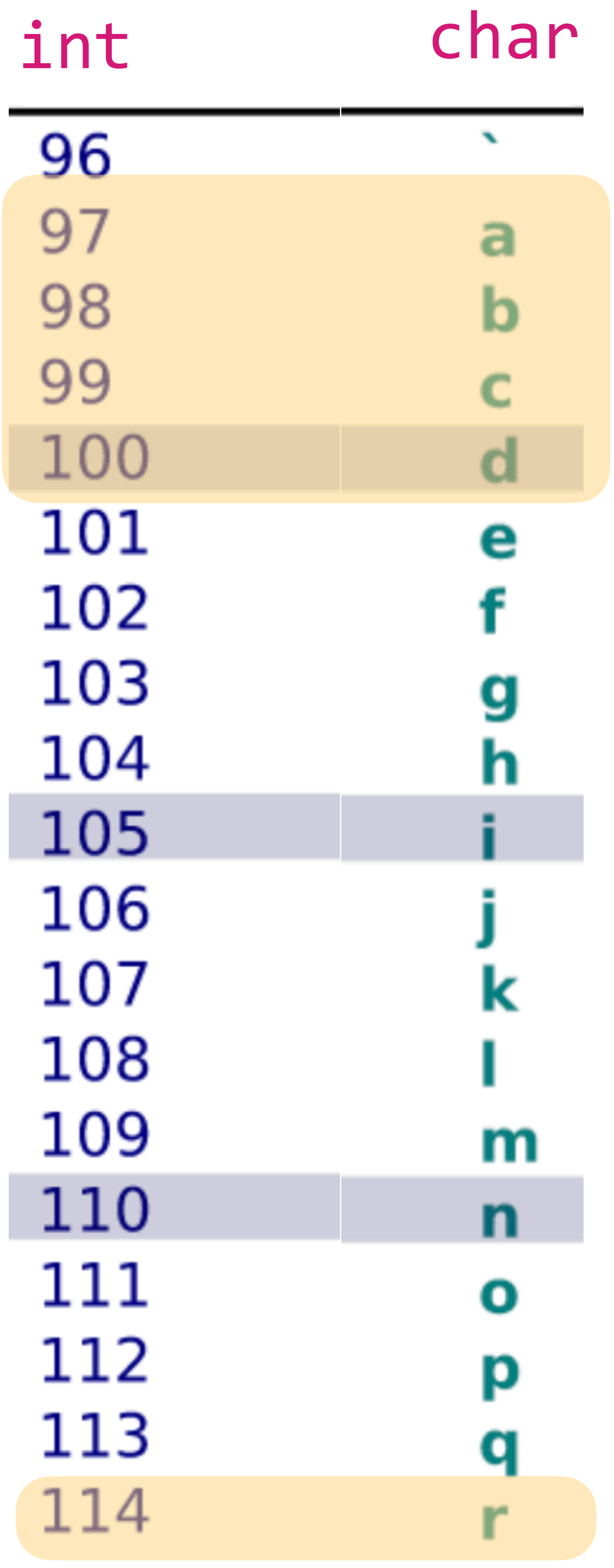
int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 6 ,  2 ,  1 ,   1 , ... ,   2 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...
```

```
S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```



int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 6 ,  3 ,  1 ,   1 , ... ,   2 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...
```

```
S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →  1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
     ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 6 ,  3 ,  1 ,   1 , ... ,   2 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 6 ,  3 ,  1 ,   1 , ... ,   3 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
  C[1..256] = [0,0,...,0]
  for i = 1..|S|:
    j = int(S[i])
    C[j] += 1
  for i = 1..|C|:
    print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 6 ,  3 ,  1 ,   1 , ... ,  3 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...
```

```
S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i ⟶  1   2   3   5   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19
     ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
      j = int(S[i])
      C[j] += 1
    for i = 1..|C|:
      print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 7 ,  3 ,  1 ,   1 , ... ,   3 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...
```

```
S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i ⟶  1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
     ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

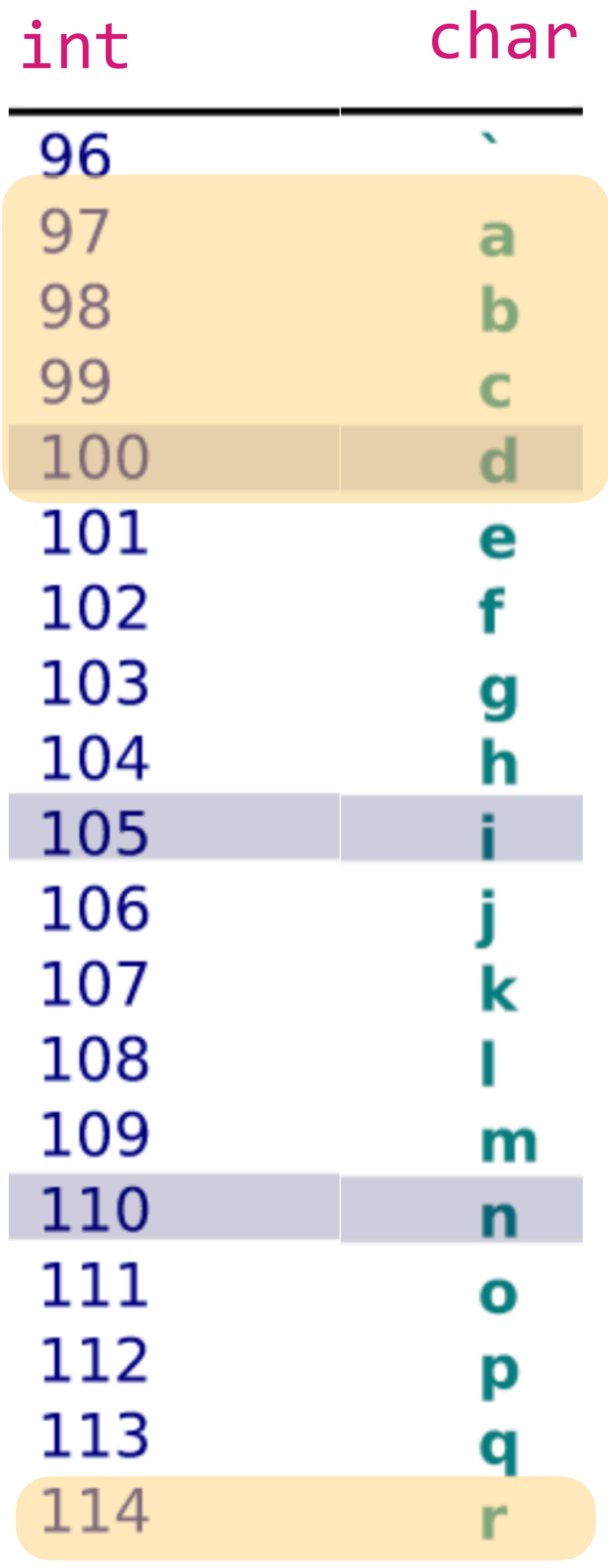
```
all_occ_count_v2(S):
  C[1..256] = [0,0,...,0]
  for i = 1..|S|:
    j = int(S[i])
    C[j] += 1
  for i = 1..|C|:
    print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 7 ,  3 ,  1 ,   1 , ... ,   3 , ... ]
     0  1  ...   97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
      j = int(S[i])
      C[j] += 1
    for i = 1..|C|:
      print(char(i),C[i])
```



int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 7 , 3 , 2 ,  1 , ... ,  3 , ... ]
     0  1  ...  97  98  99  100   ...  114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →  1   2   3   5   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19
     ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

```
C = [0, 0, ... , 7 ,  3 ,  2 ,   1 , ... ,   3 , ... ]
     0  1  ...   97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i ⟶  1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19

     ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 8 ,  3 ,  2 ,   1 , ... ,   3 , ... ]
     0  1  ...   97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 8 ,  3 ,  2 ,   1 , ... ,  3 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...
```

```
S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

```
all_occ_count_v2(S):
  C[1..256] = [0,0,...,0]
  for i = 1..|S|:
    j = int(S[i])
    C[j] += 1
  for i = 1..|C|:
    print(char(i),C[i])
```

```
C = [0, 0, ... , 8 ,  4 ,  2 ,   1 , ... ,   3 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...
```

```
S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →  1   2   3   5   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19
     ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

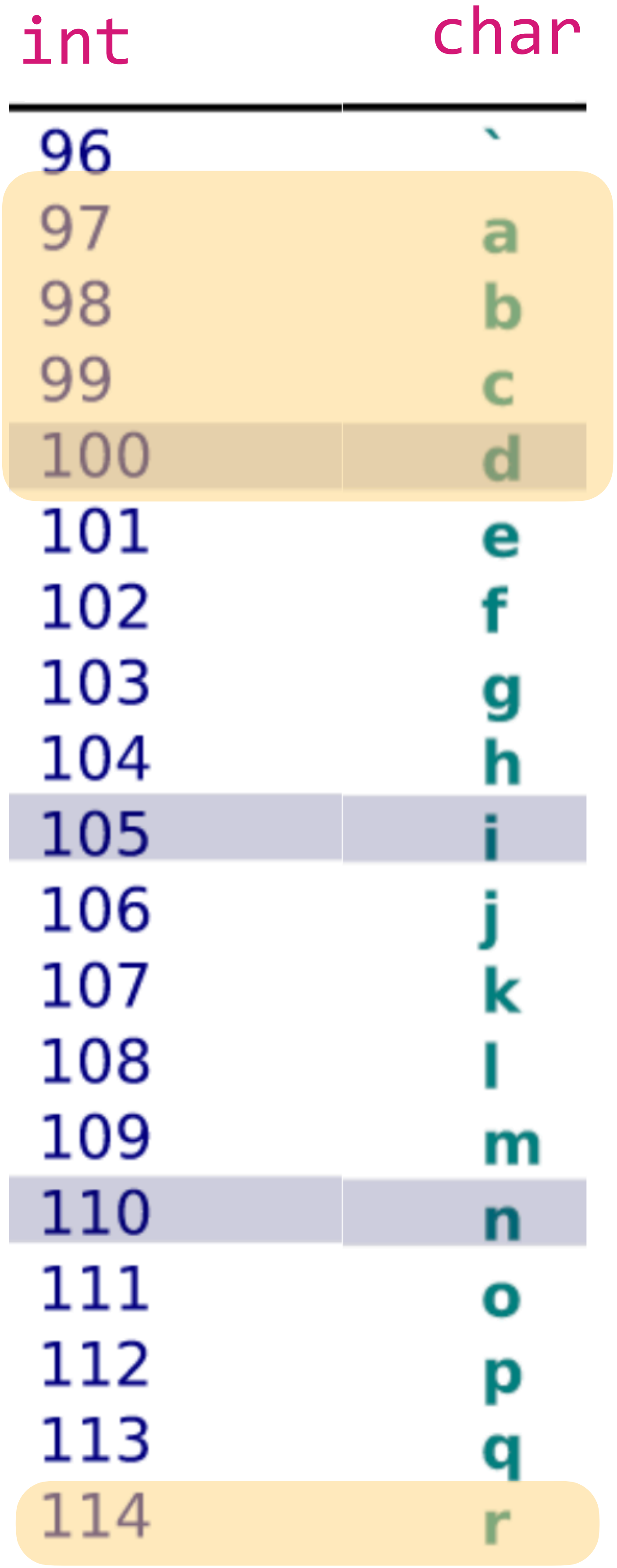
Warm up — Counting occurrences

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
C = [0, 0, ... , 8 ,  4 ,  2 ,   1 , ... ,   3 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i →   1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
      ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

Warm up — Counting occurrences

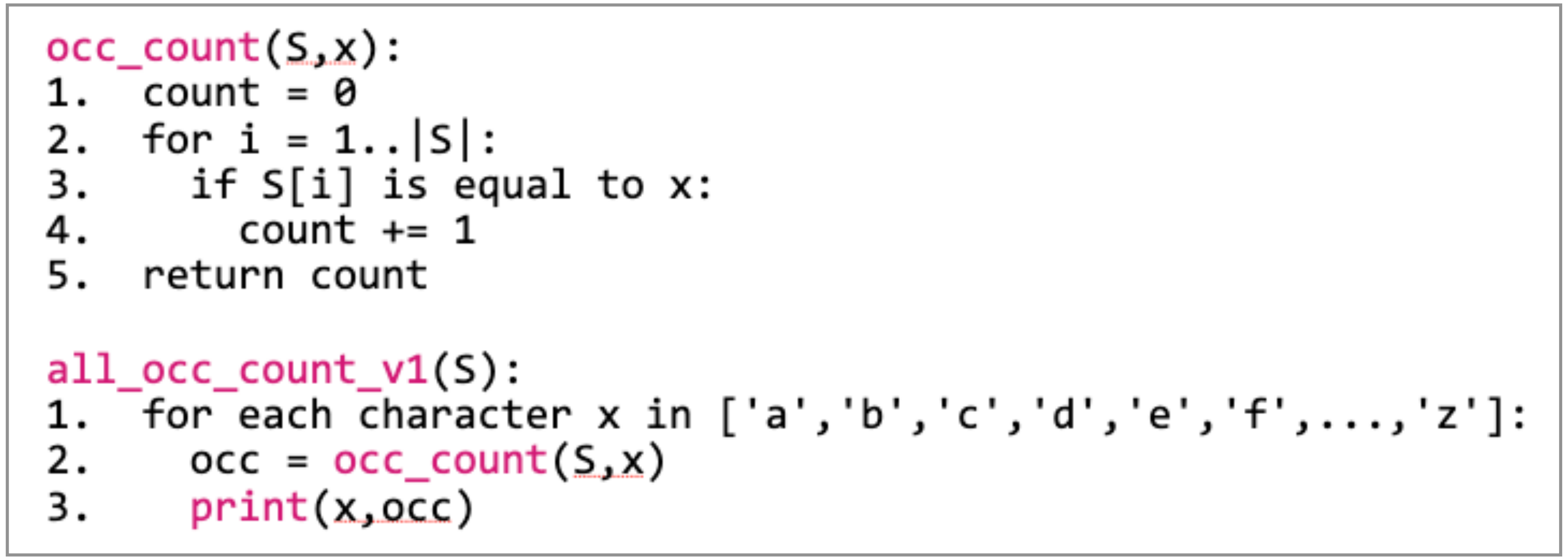


int	char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r

```
all_occ_count_v2(S):
C[1..256] = [0,0,...,0]
for i = 1..|S|:
  j = int(S[i])
  C[j] += 1
for i = 1..|C|:
  print(char(i),C[i])
```

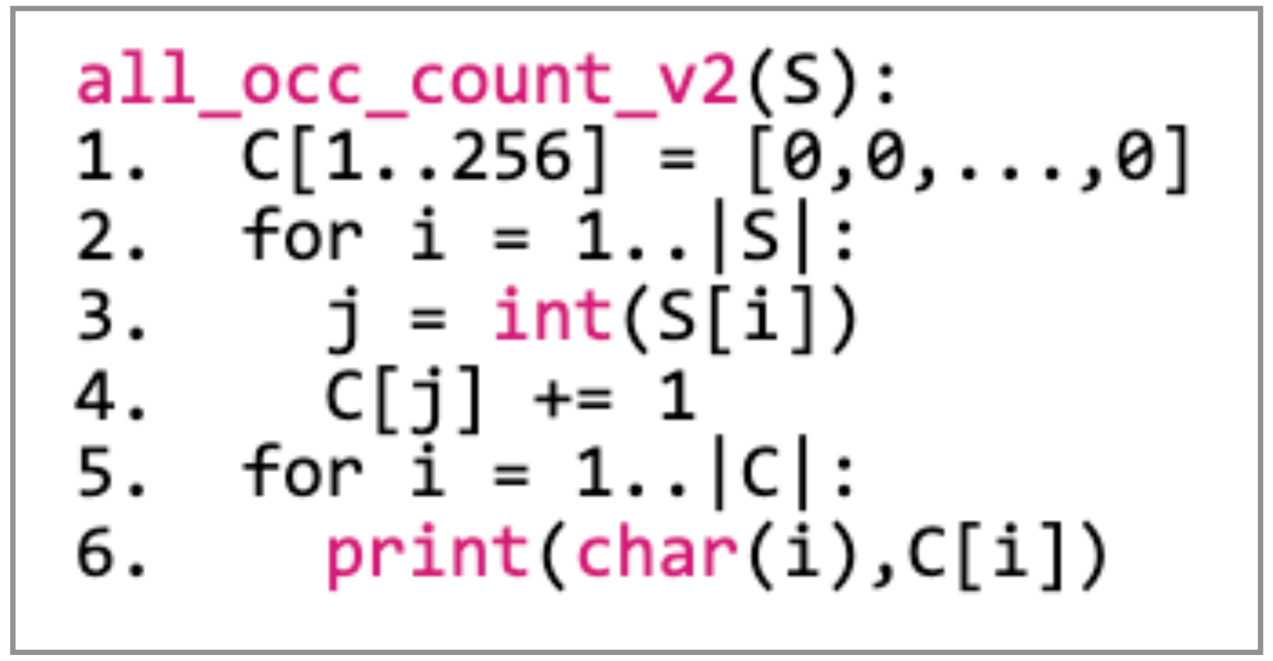
```
C = [0, 0, ... , 9 ,  4 ,  2 ,   1 , ... ,   3 , ... ]
     0  1  ...  97   98   99   100   ...   114   ...

S = ['a','b','r','a','c','a','d','a','b','r','a','a','b','r','a','c','a','b','a']
i ⟶  1   2   3   5   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
     ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
```

We have two different algorithms for the same problem



```
occ_count(S,x):
    count = 0
    for i = 1..|S|:
      if S[i] is equal to x:
        count += 1
    return count

all_occ_count_v1(S):
    for each character x in ['a','b','c','d','e','f',...,'z']:
      occ = occ_count(S,x)
      print(x,occ)
```

v1



```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
      j = int(S[i])
      C[j] += 1
    for i = 1..|C|:
      print(char(i),C[i])
```

v2

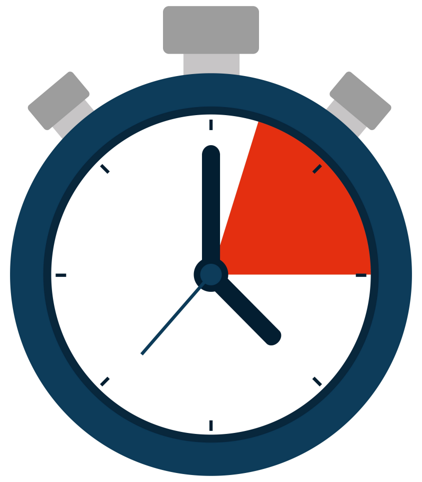
- Algorithm v2 uses a data structure (an array), whereas algorithm v1 does not.
- **Q.** Which one should we use?
- To answer this question we need to **analyse** an algorithm.

Basic definitions — Analysis of algorithms

- When developing a solution to a problem with an algorithm, we are concerned about two things:

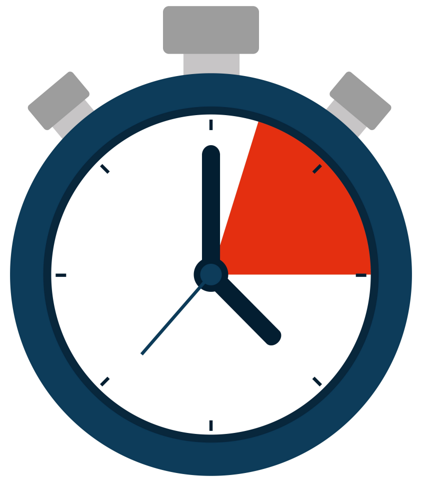
Basic definitions — Analysis of algorithms

- When developing a solution to a problem with an algorithm, we are concerned about two things:
  - the **running time** of the algorithm;
    Q. After how many seconds will my algorithm terminate?



Basic definitions — Analysis of algorithms

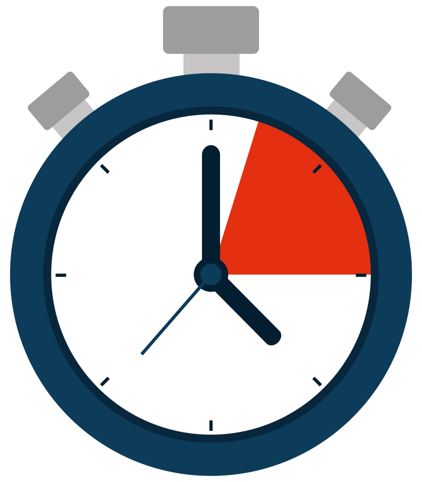
- When developing a solution to a problem with an algorithm, we are concerned about two things:
  - the **running time** of the algorithm;
Q. After how many seconds will my algorithm terminate?



  - the **space** taken by the data structure(s) it uses.
Q. Can I run my algorithm on my computer with 4GB of RAM?

Basic definitions — Analysis of algorithms

- When developing a solution to a problem with an algorithm, we are concerned about two things:
  - the **running time** of the algorithm;
    Q. After how many seconds will my algorithm terminate?



  - the **space** taken by the data structure(s) it uses.
    Q. Can I run my algorithm on my computer with 4GB of RAM?

- **The less, the better. We strive for efficient algorithms.**
- Analysing an algorithm means understanding its running time and memory usage.
  We will talk more about this soon.

Part 1 — Summary

- Definition of Algorithm and Data Structure
- Arrays and memory
- Warm up: two algorithms for counting the occurrences of characters in strings

Part 2 — Motivations, analysis of algorithms, same applications

Why algorithms?

- The romantic/philosophical view: **algorithms describe our life**.
- Fundamental questions:
  - **Q.** What problems can I solve?
  - **Q.** And how, i.e., what resources do I need?
  - **Q.** Can I do better (use less resources)?

Why algorithms?

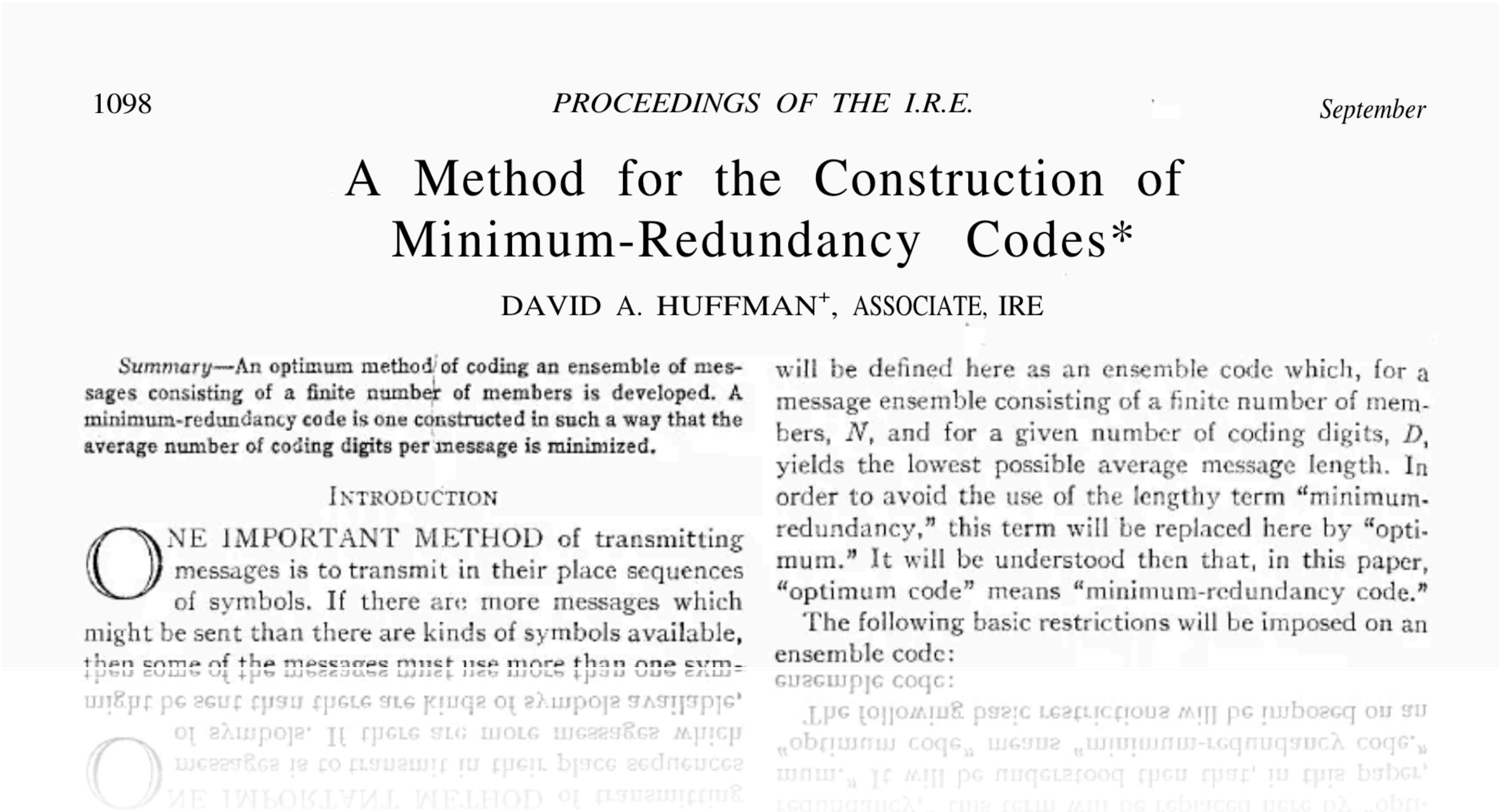
- The romantic/philosophical view: **algorithms describe our life**.
- Fundamental questions:
  - **Q.** What problems can I solve?
  - **Q.** And how, i.e., what resources do I need?
  - **Q.** Can I do better (use less resources)?
- Understanding if we can do something better has always been a primary question in the history of human evolution.
- There are many known algorithms. Yet, probably more need to be invented!
- **Democracy: can be invented by anyone, anywhere. You could be next!**

Huffman's data compression algorithm

Robert Fano
(1917 - 2016)

David Huffman
(1925 - 1999)

- D. Huffman was a graduate student at MIT in 1951.
- He solved an open problem left by his teacher R. Fano, during a class on Information Theory.



1098 *PROCEEDINGS OF THE I.R.E.* *September*

A Method for the Construction of
Minimum-Redundancy Codes*

DAVID A. HUFFMAN$^+$, ASSOCIATE, IRE

Summary—An optimum method of coding an ensemble of messages consisting of a finite number of members is developed. A minimum-redundancy code is one constructed in such a way that the average number of coding digits per message is minimized.

INTRODUCTION

ONE IMPORTANT METHOD of transmitting messages is to transmit in their place sequences of symbols. If there are more messages which might be sent than there are kinds of symbols available, then some of the messages must use more than one sym-

will be defined here as an ensemble code which, for a message ensemble consisting of a finite number of members, *N*, and for a given number of coding digits, *D*, yields the lowest possible average message length. In order to avoid the use of the lengthy term "minimum-redundancy," this term will be replaced here by "optimum." It will be understood then that, in this paper, "optimum code" means "minimum-redundancy code."

The following basic restrictions will be imposed on an ensemble code:

Why algorithms?

- The practical view: to **solve problems** that are otherwise “impossible” to solve in a **reasonable amount of time**.
- **Example.** Sub-string search.
  Q. Does the following string contain `"CGTGGTTAAACGAGC"` and, if so, at what position?

```
TTGCTCATGCGCCGGTGCCTGCAGTGAACTGAGGAAAATAAGTTGTTTAACCGGCGTCTGGCGCAGCGCGTCGCGCACGTTGAGCGCCGCCTGACGCTCATGGGCGATAAAATCGCCGCCTTCGCCCATGCCGTGTACCAGATAGTAAAC
GGTATCAATGTCGCGAAGCAGCGCGGGTAAATTTTCCGGCCAGTGCAGATCGACCTTATGACAACTGACGTTGGCGAGGCGCTGTTTTTCCAGACGTTCGATGCGTCGCGCCGCCGCTCGCACCTGATGCCCTTGCTGACTTAGCGCAAA
GACCAGGTGCTGGCCGATATAGCCGCTGGCGCCGAGAACCAGAATGCGTTGCGCCACGTTGCTCTCCTTAGCGGGCTAAAAAGGCGCGCCAGTGGGCGACGACGTCGGTAAGCTGTTCGCGAGAGACATCCAGATGCGTCACCAGACGCA
CAATCGGCGCGGCGTTAATCAGGATATTCCGTTCCCGCAAATAGTCGCCAAGCGCGGCGGCCTGTGCTTCGCCAACGCGAACAAACAGCATATTCGTTTCGTGGCGTATGACCTCCGCGCCCGCTTCCCGAAGCTGCTGCGCCAGCCAGG
CGGCGTTATCATGATCCTCTTGCAGACGCGCCACGTTATGCTTCAGCGCATACAGTCCGGCCGCTGCCAGAATCCCGGCCTGACGCATTCCGCCGCCGACCATTTTACGCCAGCGCGTCGCGCGTTTAATGTAATCGCGGTTGCCGACCA
GCAGTGAACCGACCGGCGTTCCGAGACCTTTTGACAGGCAGATGGTAAAAGAGTCGCAATACTGCGTAATCTCTTTTAACTCACAGCCGTAGGCAACCACCGCGTTAAAAATTCGGGCGCCGTCAACGTGCAGCGCCAGTCCACGTTCGC
GGGTAAATGTCCAGGCGTCTTTCAGATACGCGCGCGGCAGCACTTTCCCGTTATGCGTATTTTCCAGACTGAGCAAGCGCGTGCGCGCGAAGTGGATGTCATCCGCTTTAATCTTCGCCGCCACGTTCTCCAGCGGCAGCGTACCGTCCG
CGGCGGCGTCGATGGGCTGCGGCTGAATGCTGCCGAGCACCGCCGCGCCGCCAGCTTCATAGAGATAATTATGCGCGCCCTGACCGACGATATACTCTTCGCCGCGTTCACAATGGCTAAGCAGCGCGACCAGATTGGCCTGGGTGCCGG
TGGGTAAAAAAAGCGCCGCTTCTTTACCGGAAAGGTCGGCGGCGTAGCGCTGAAGGGCGTTAACAGTAGGGTCATCCCCGACCGGGGCGGTCATCATCGCCTCGAGCATGGCGCGGCCCGGTCGGGTAACGGTATCACTGCGTAAATCAA
TCATGGCACATCCCTGGATTTTAAAAGGTGATGTGCACTGTTTTACCTTAGCCAGTTCGTTTTCGCCAGTTCGATCACTTCATCGCCGCGGCCGCTAATAATGGCGCGTAGCATGTACAGGCTAAAGCCTTTCGCTTGTTCCAGTTTGAT
CTGCGGCGGGATCGCCAGCTCTTCTTTTGCCACGACCACGTCAACCAGTACCGGGCCGTCAATGGAAAACGCGCGCTGTAGCGCACCGTCCACGTCTGCGGCTTTTTCCACGCGAATACCGGTAATGCCGCAGGCTTCGGCGATACGCGC
GAAATTGGTGTCGTGCAGTTCGGTACCGTCGGTAAGGTAGCCGCCGGCTTTCATTTCCATCGCCACAAAGCCCAGCACGCTGTTATTAAAGACGACGATTTTTATCGGCAGCTTCATCTGTACCACCGAGAGAAAATCGCCCATCAGCAT
ACTGAAGCCGCCATCACCGCACATCGCGATAACCTGACGACCCGGCGCGGTAGCCTGAGCGCCGAGCGCCTGCGGCATAGCGTTGGCCATTGACCCGTGGTTAAACGAGCCTAGCAGGCGGCGCTTGCCGTTCATTTTTAGATAGCGGGC
CGCCCAGACGGTCGGCGTGCCGACATCGCAGGTAAAAATAGCGTCGTCAGCGGCGAAATGACTAATTTGTTGCGCCAGATATTGTGGGTGGATGGCTTTATCGCTGAGTTTGGCTAAGTCATCAAGTCCCTTACGGGCGTCCCGATAGTG
CTCCAGAGCTTTATCGAGGAATTTACGATTGCTTTTTTCTTCCACCAGCGGCAGCAGGGCGCGAAGCGTGGCTTTAATATCGCCCACTAGCGCCATGTCGACTTTGCTGTGCGCGCCAATACTGCCCGGGTTGATGTCAATCTGAATGAT
TTTGGCATCGCTCGGATAAAAGGCGCGATAGGGGAACTGGGTGCCGAGCAGGATCAGCGTATCGGCGTTCATCATGGTGTGGAAGCCAGAAGAGAAGCCAATCAGGCCGGTCATTCCCACATCATAAGGGTTATCGTACTCAACGTGCTC
TTTGCCGCGCAGGGCATGAGCGATTGGCGCTTTTAGTTTTGCCGCCAACGCGACCAACTCCTCATGCGCGCCCGCGCAGCCGCTACCGCACATCAATGCGATATTGCTGGAGTAGCGCAGCAGTTGCGCCAGTTTTTTCAG
```

Why algorithms?

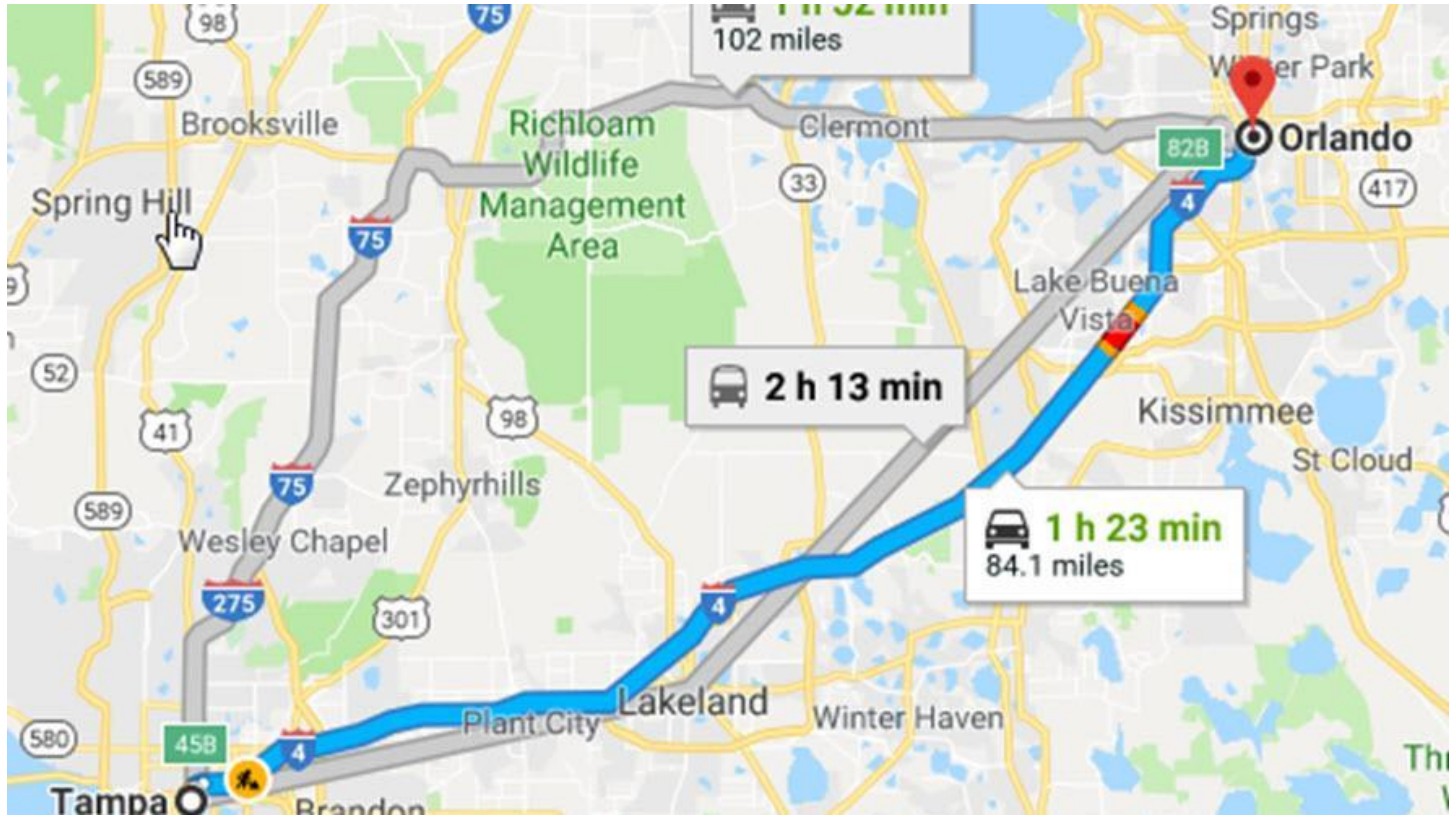
- The practical view: to **solve problems** that are otherwise “impossible” to solve in a **reasonable amount of time**.
- **Example.** Sub-string search.
  Q. Does the following string contain "`CGTGGTTAAACGAGC`" and, if so, at what position?

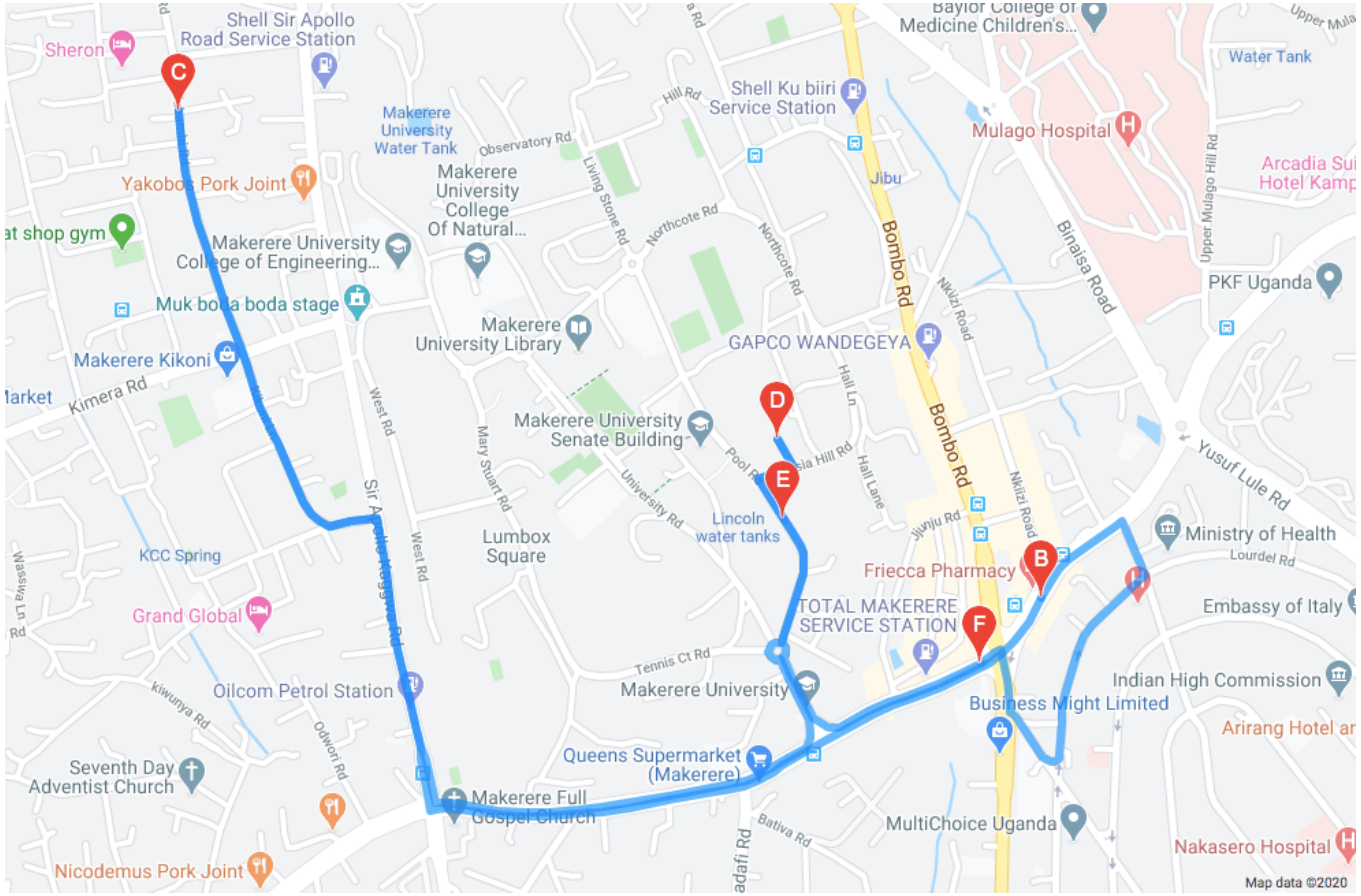
```
TTGCTCATGCGCCGGTGCCTGCAGTGAACTGAGGAAAATAAGTTGTTTAACCGGCGTCTGGCGCAGCGCGTCGCGCACGTTGAGCGCCGCCTGACGCTCATGGGCGATAAAATCGCCGCCTTCGCCCATGCCGTGTACCAGATAGTAAAC
GGTATCAATGTCGCGAAGCAGCGCGGGTAAATTTTCCGGCCAGTGCAGATCGACCTTATGACAACTGACGTTGGCGAGGCGCTGTTTTTCCAGACGTTCGATGCGTCGCGCCGCCGCTCGCACCTGATGCCCTTGCTGACTTAGCGCAAA
GACCAGGTGCTGGCCGATATAGCCGCTGGCGCCGAGAACCAGAATGCGTTGCGCCACGTTGCTCTCCTTAGCGGGCTAAAAAGGCGCGCCAGTGGGCGACGACGTCGGTAAGCTGTTCGCGAGAGACATCCAGATGCGTCACCAGACGCA
CAATCGGCGCGGCGTTAATCAGGATATTCCGTTCCCGCAAATAGTCGCCAAGCGCGGCGGCCTGTGCTTCGCCAACGCGAACAAACAGCATATTCGTTTCGTGGCGTATGACCTCCGCGCCCGCTTCCCGAAGCTGCTGCGCCAGCCAGG
CGGCGTTATCATGATCCTCTTGCAGACGCGCCACGTTATGCTTCAGCGCATACAGTCCGGCCGCTGCCAGAATCCCGGCCTGACGCATTCCGCCGCCGACCATTTTACGCCAGCGCGTCGCGCGTTTAATGTAATCGCGGTTGCCGACCA
GCAGTGAACCGACCGGCGTTCCGAGACCTTTTGACAGGCAGATGGTAAAAGAGTCGCAATACTGCGTAATCTCTTTTAACTCACAGCCGTAGGCAACCACCGCGTTAAAAATTCGGGCGCCGTCAACGTGCAGCGCCAGTCCACGTTCGC
GGGTAAATGTCCAGGCGTCTTTCAGATACGCGCGCGGCAGCACTTTCCCGTTATGCGTATTTTCCAGACTGAGCAAGCGCGTGCGCGCGAAGTGGATGTCATCCGCTTTAATCTTCGCCGCCACGTTCTCCAGCGGCAGCGTACCGTCCG
CGGCGGCGTCGATGGGCTGCGGCTGAATGCTGCCGAGCACCGCCGCGCCGCCAGCTTCATAGAGATAATTATGCGCGCCCTGACCGACGATATACTCTTCGCCGCGTTCACAATGGCTAAGCAGCGCGACCAGATTGGCCTGGGTGCCGG
TGGGTAAAAAAAGCGCCGCTTCTTTACCGGAAAGGTCGGCGGCGTAGCGCTGAAGGGCGTTAACAGTAGGGTCATCCCCGACCGGGGCGGTCATCATCGCCTCGAGCATGGCGCGGCCCGGTCGGGTAACGGTATCACTGCGTAAATCAA
TCATGGCACATCCCTGGATTTTAAAAGGTGATGTGCACTGTTTTACCTTAGCCAGTTCGTTTTCGCCAGTTCGATCACTTCATCGCCGCGGCCGCTAATAATGGCGCGTAGCATGTACAGGCTAAAGCCTTTCGCTTGTTCCAGTTTGAT
CTGCGGCGGGATCGCCAGCTCTTCTTTTGCCACGACCACGTCAACCAGTACCGGGCCGTCAATGGAAAACGCGCGCTGTAGCGCACCGTCCACGTCTGCGGCTTTTTCCACGCGAATACCGGTAATGCCGCAGGCTTCGGCGATACGCGC
GAAATTGGTGTCGTGCAGTTCGGTACCGTCGGTAAGGTAGCCGCCGGCTTTCATTTCCATCGCCACAAAGCCCAGCACGCTGTTATTAAAGACGACGATTTTTATCGGCAGCTTCATCTGTACCACCGAGAGAAAATCGCCCATCAGCAT
ACTGAAGCCGCCATCACCGCACATCGCGATAACCTGACGACCCGGCGCGGTAGCCTGAGCGCCGAGCGCCTGCGGCATAGCGTTGGCCATTGACCCGTGGTTAAACGAGCCTAGCAGGCGGCGCTTGCCGTTCATTTTTAGATAGCGGGC
CGCCCAGACGGTCGGCGTGCCGACATCGCAGGTAAAAATAGCGTCGTCAGCGGCGAAATGACTAATTTGTTGCGCCAGATATTGTGGGTGGATGGCTTTATCGCTGAGTTTGGCTAAGTCATCAAGTCCCTTACGGGCGTCCCGATAGTG
CTCCAGAGCTTTATCGAGGAATTTACGATTGCTTTTTTCTTCCACCAGCGGCAGCAGGGCGCGAAGCGTGGCTTTAATATCGCCCACTAGCGCCATGTCGACTTTGCTGTGCGCGCCAATACTGCCCGGGTTGATGTCAATCTGAATGAT
TTTGGCATCGCTCGGATAAAAGGCGCGATAGGGGAACTGGGTGCCGAGCAGGATCAGCGTATCGGCGTTCATCATGGTGTGGAAGCCAGAAGAGAAGCCAATCAGGCCGGTCATTCCCACATCATAAGGGTTATCGTACTCAACGTGCTC
TTTGCCGCGCAGGGCATGAGCGATTGGCGCTTTTAGTTTTGCCGCCAACGCGACCAACTCCTCATGCGCGCCCGCGCAGCCGCTACCGCACATCAATGCGATATTGCTGGAGTAGCGCAGCAGTTGCGCCAGTTTTTTCAG
```

In this case, the answer is "**yes**: at position **1896**".

Why algorithms?

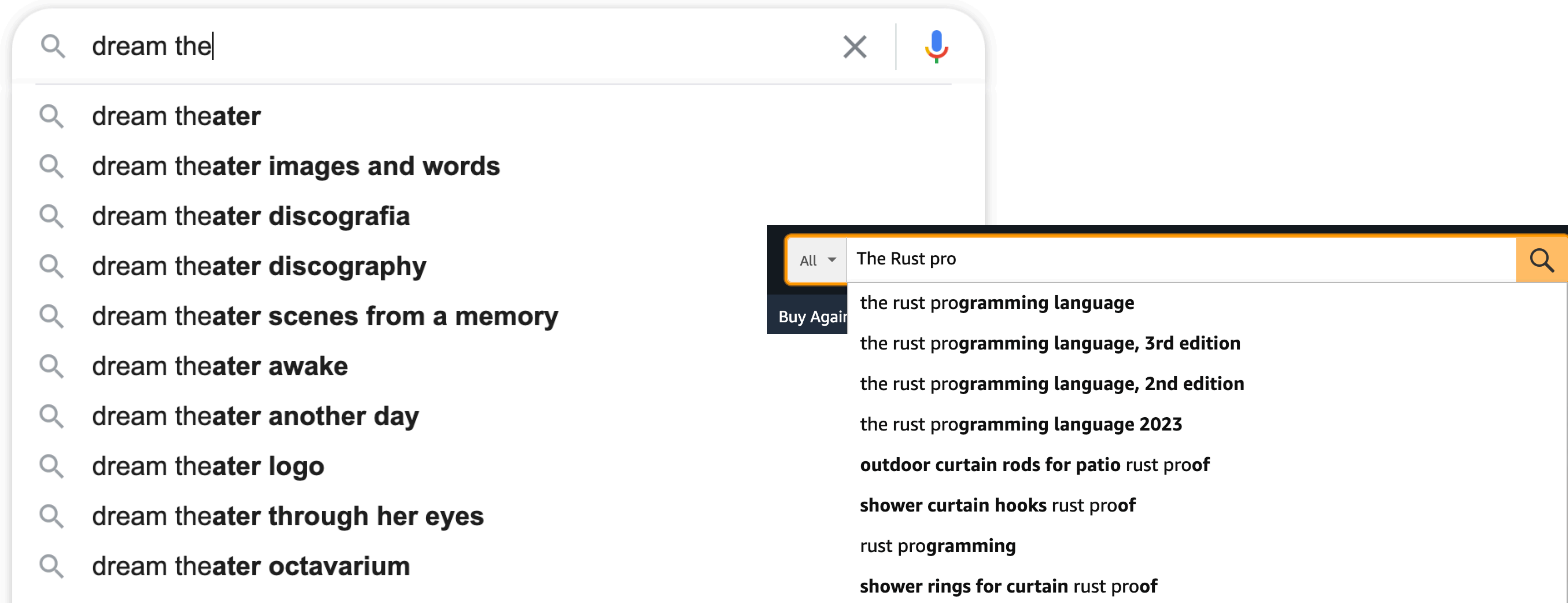
- **Example.** Shortest path between two points in a map.





Why algorithms?

- **Example.** Query suggestion.



dream the

- dream the**ater**
- dream the**ater images and words**
- dream the**ater discografia**
- dream the**ater discography**
- dream the**ater scenes from a memory**
- dream the**ater awake**
- dream the**ater another day**
- dream the**ater logo**
- dream the**ater through her eyes**
- dream the**ater octavarium**

The Rust pro

- the rust pro**gramming language**
- the rust pro**gramming language, 3rd edition**
- the rust pro**gramming language, 2nd edition**
- the rust pro**gramming language 2023**
- **outdoor curtain rods for patio** rust proof
- **shower curtain hooks** rust proof
- rust pro**gramming**
- **shower rings for curtain** rust proof

Why algorithms?

- The practical view: for **profit**.
- **Build better systems/applications** in terms of reduced latency to use the service.

  → Make your users happy so that they will keep using your service (and you will keep earning)!

Why algorithms?

- The practical view: for **profit**.
- **Build better systems/applications** in terms of reduced latency to use the service.

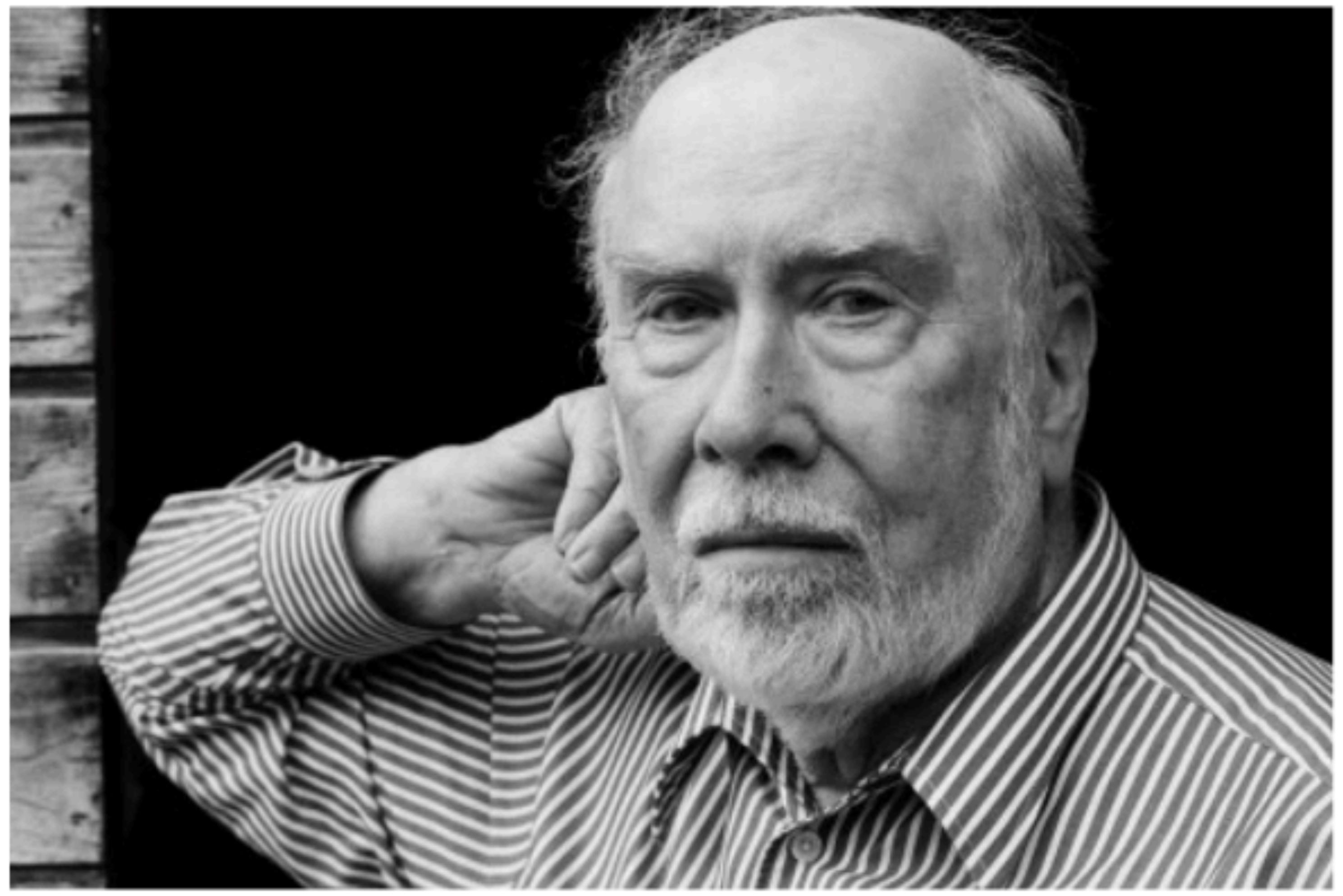
  → Make your users happy so that they will keep using your service (and you will keep earning)!

- **Save computer resources** (power and storage machines).

The increase of data does not scale with technology

- These considerations are **even more relevant today** than in the past.
- Today we are facing a **data explosion** phenomenon.

"Software is getting slower more rapidly than hardware becomes faster."

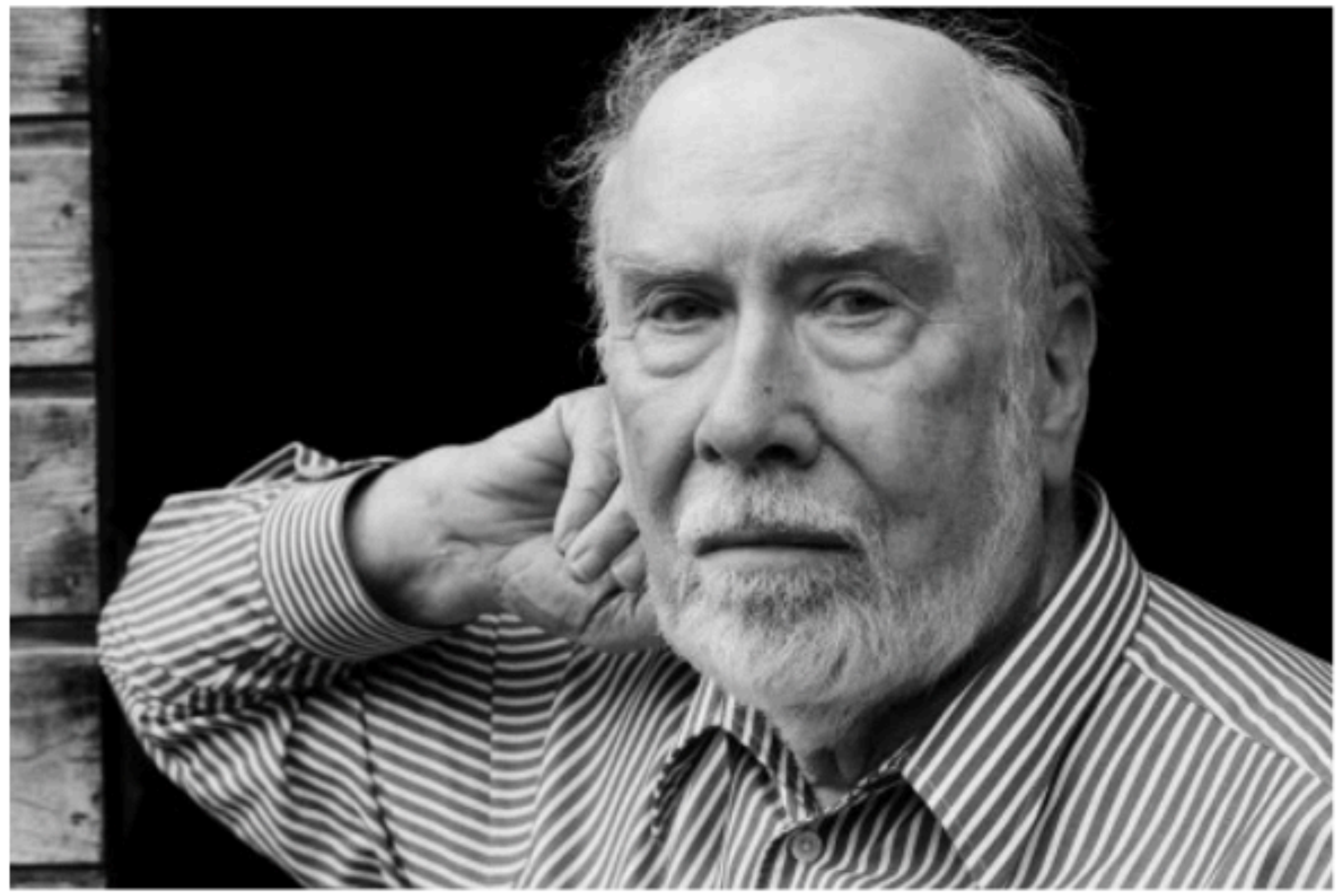


Niklaus Wirth

The increase of data does not scale with technology

- These considerations are **even more relevant today** than in the past.
- Today we are facing a **data explosion** phenomenon.

“Software is getting slower more rapidly than hardware becomes faster.”



Niklaus Wirth

→ Lesson learnt: **a better algorithm is always better than a better computer!**

Data explosion

- More data…

Data centers

- More computers…

Applications are more data intensive than ever

- More electricity spent → more money spent!
- **The more efficient an algorithm is, the less electricity it requires to run.**

Why algorithms?

- We need **good programmers to implement efficient algorithms**.

> *"Bad programmers worry about the code.*
> *Good programmers worry about data*
> *structures and their relationships."*

Linus Torvalds
(creator of Linux)

Why algorithms?

- We need **good programmers to implement efficient algorithms**.

> *"Bad programmers worry about the code. Good programmers worry about data structures and their relationships."*

Linus Torvalds
(creator of Linux)

Why algorithms? — Recap

- To **better understand** what we can do with computers.
- To **solve** real-world problems that could be otherwise impossible to solve.
- To get a **well-paid job**.

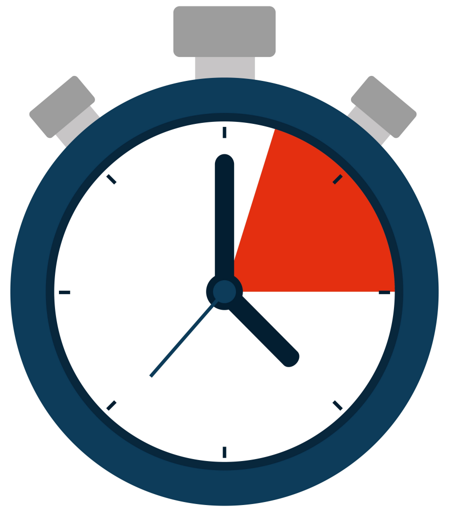
Why algorithms? — Recap

- To **better understand** what we can do with computers.
- To **solve** real-world problems that could be otherwise impossible to solve.
- To get a **well-paid job**.

→ **No reason not to study Computer Science and algorithms!**

Analysis of algorithms

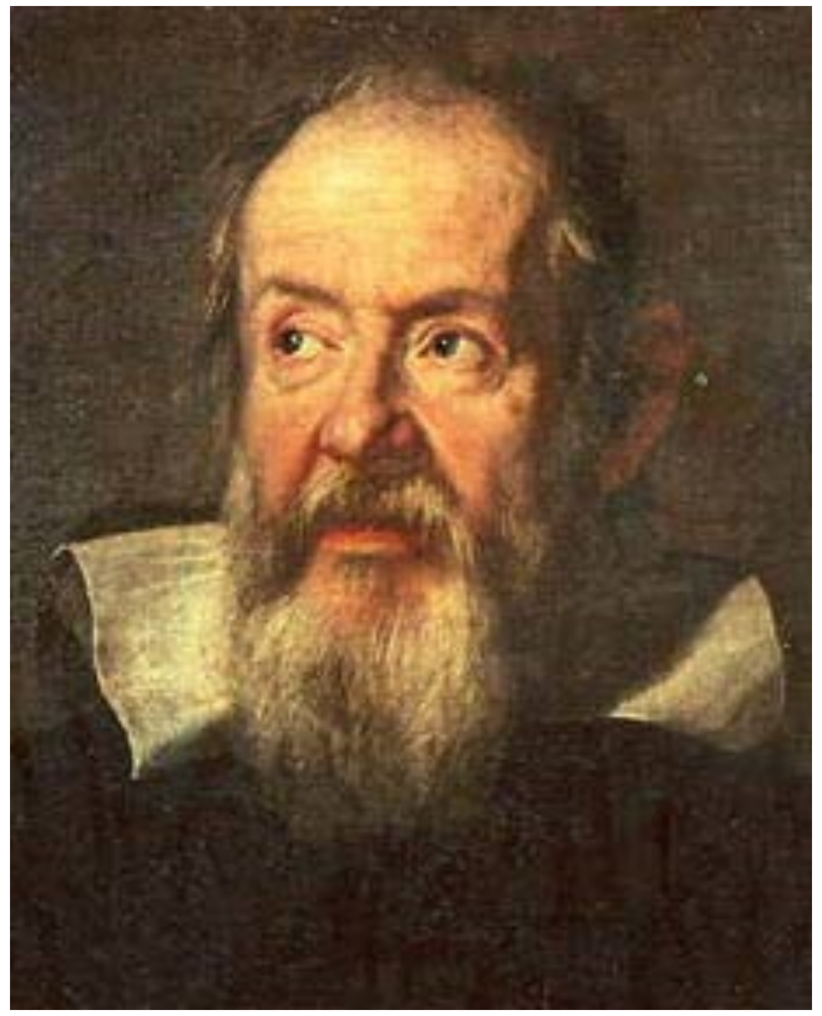
- When developing a solution to a problem with an algorithm, we are concerned about two things:
  - the **running time** of the algorithm;
  - the **space** taken by the data structure(s) it uses.



- **The less, the better.**
- **Trade-off between time and space** of a solution.

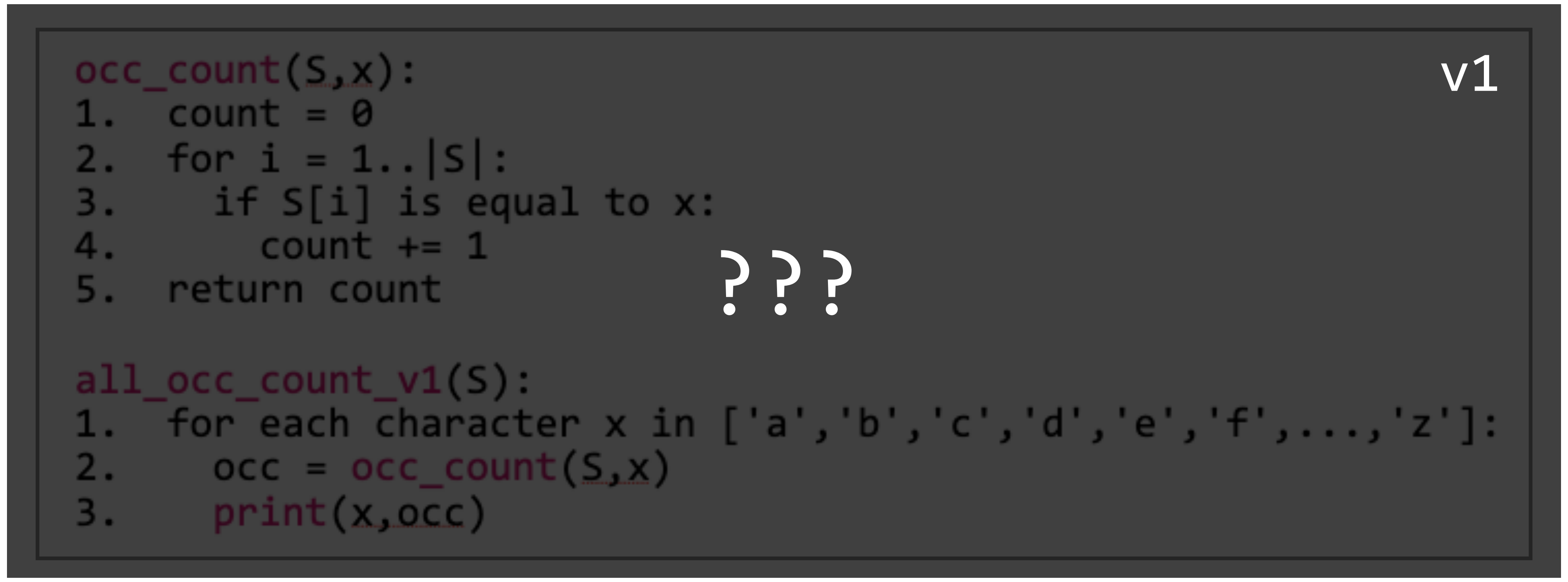
The running time — The scientific method

- Scientific method:
  1. **Observe**.
  2. Formulate an **hypothesis**.
  3. Make a **prediction**.
  4. **Validate**: if prediction is valid, then stop; repeat otherwise.

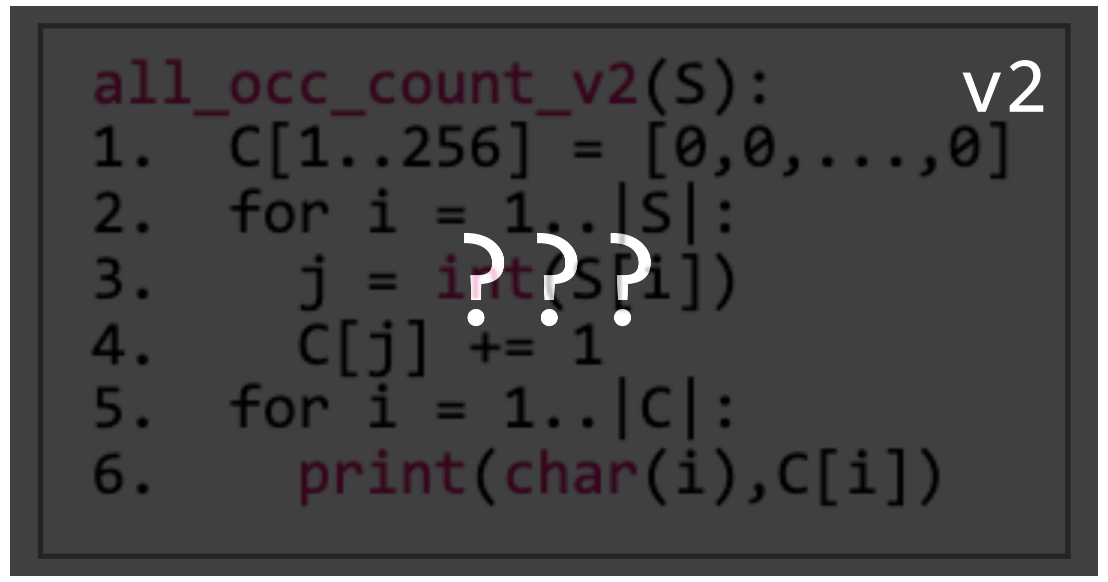


Galileo Galilei

The running time — The scientific method



```
occ_count(S,x):
    count = 0
    for i = 1..|S|:
      if S[i] is equal to x:
        count += 1
    return count

all_occ_count_v1(S):
    for each character x in ['a','b','c','d','e','f',...,'z']:
      occ = occ_count(S,x)
      print(x,occ)
```

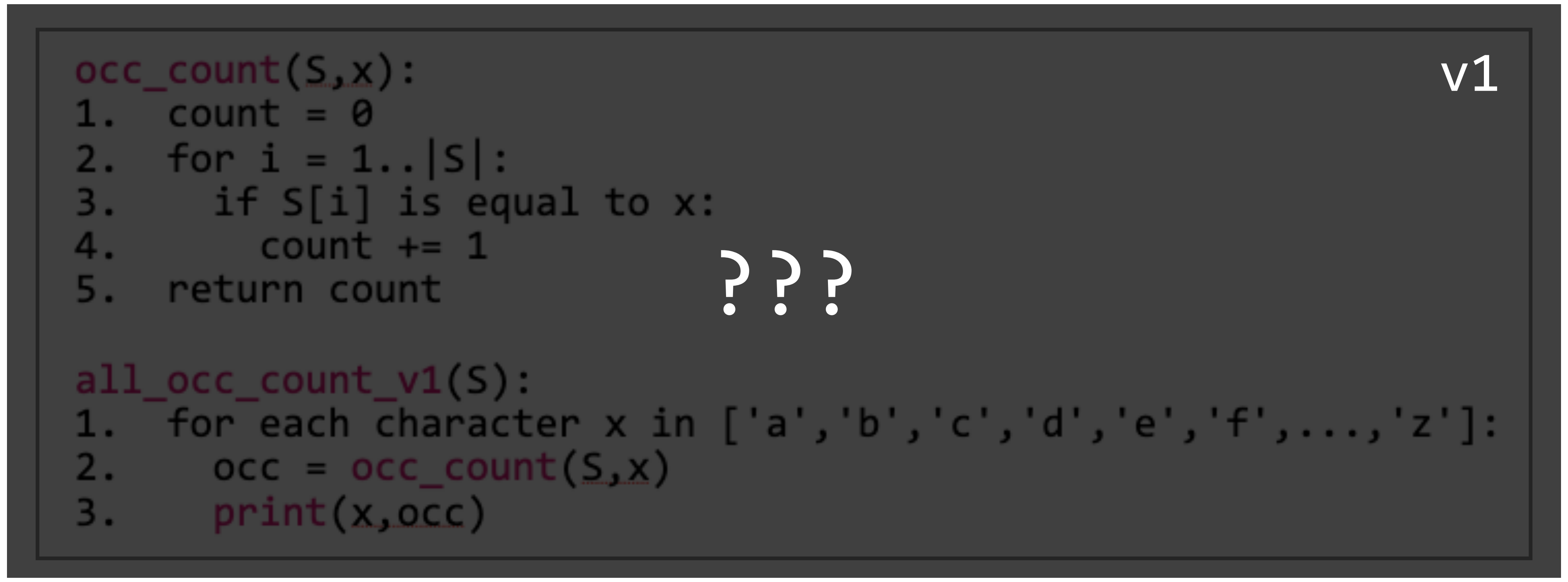


```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
      j = int(S[i])
      C[j] += 1
    for i = 1..|C|:
      print(char(i),C[i])
```

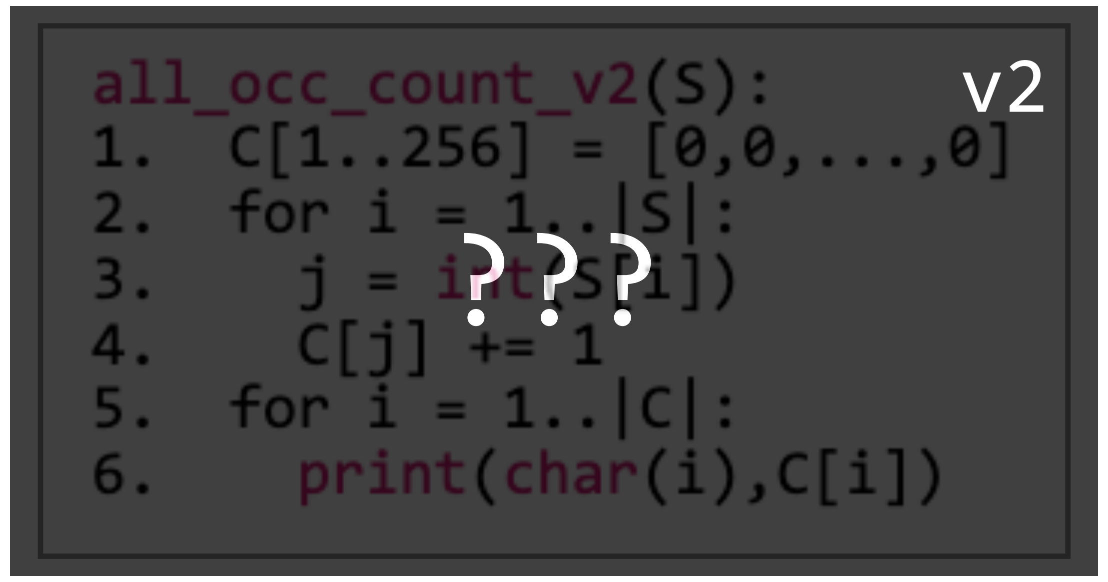
|S|	v1	v2
0.5M	118 ms	3 ms
1M	201 ms	6 ms
2M	372 ms	13 ms
4M	721 ms	26 ms

M = 1 million; 1 ms = 1/1000 sec

The running time — The scientific method



```
occ_count(S,x):
   count = 0
   for i = 1..|S|:
     if S[i] is equal to x:
       count += 1
   return count

all_occ_count_v1(S):
   for each character x in ['a','b','c','d','e','f',...,'z']:
     occ = occ_count(S,x)
     print(x,occ)
```



```
all_occ_count_v2(S):
   C[1..256] = [0,0,...,0]
   for i = 1..|S|:
     j = int(S[i])
     C[j] += 1
   for i = 1..|C|:
     print(char(i),C[i])
```

|S|	v1	v2
0.5M	118 ms	3 ms
	↓ ≈1.70	↓ ≈2.00
1M	201 ms	6 ms
	↓ ≈1.85	↓ ≈2.17
2M	372 ms	13 ms
	↓ ≈1.94	↓ ≈2.00
4M	721 ms	26 ms

M = 1 million; 1 ms = 1/1000 sec

The running time — The scientific method

v1

```
occ_count(S,x):
    count = 0
    for i = 1..|S|:
        if S[i] is equal to x:
            count += 1
    return count

all_occ_count_v1(S):
    for each character x in ['a','b','c','d','e','f',...,'z']:
        occ = occ_count(S,x)
        print(x,occ)
```

? ? ?

v2

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

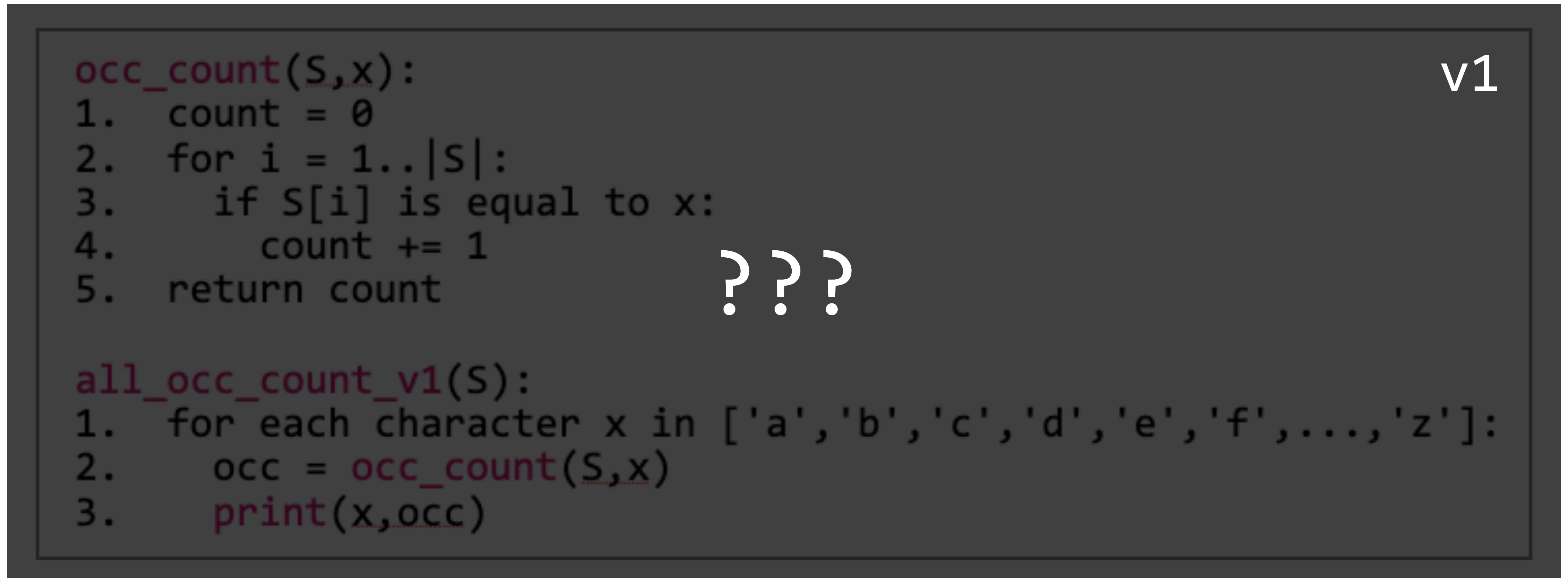
? ? ?

|S|	v1	v2
0.5M	118 ms	3 ms
	↓ ≈1.70	↓ ≈2.00
1M	201 ms	6 ms
	↓ ≈1.85	↓ ≈2.17
2M	372 ms	13 ms
	↓ ≈1.94	↓ ≈2.00
4M	721 ms	26 ms

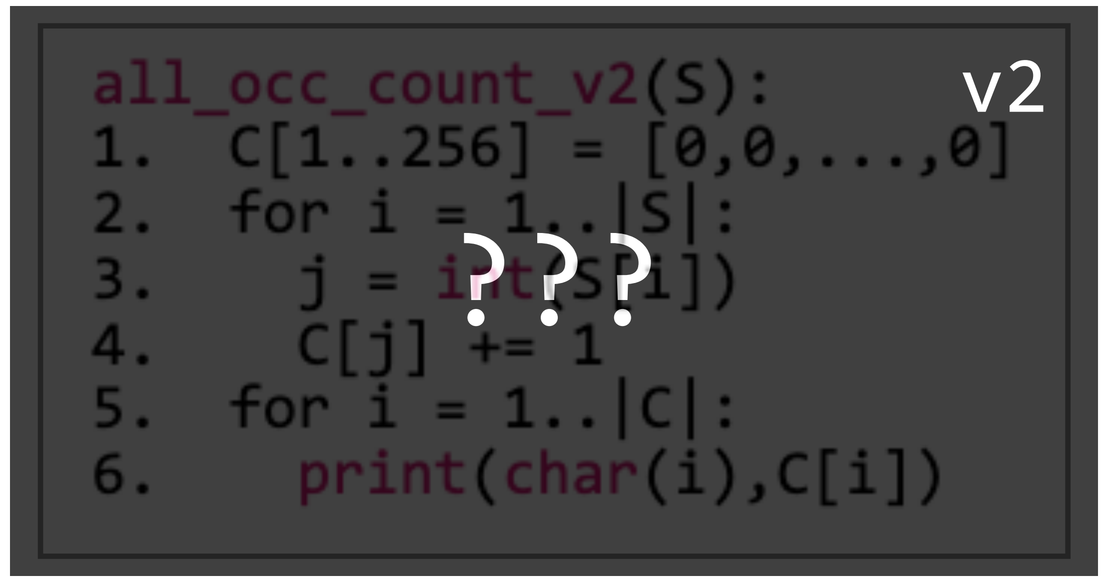
M = 1 million; 1 ms = 1/1000 sec

- First observation: as the **input doubles in size**, **also the running time** of both v1 and v2 **doubles**.
- First hypothesis: the running time has a **linear** dependency from the input size.

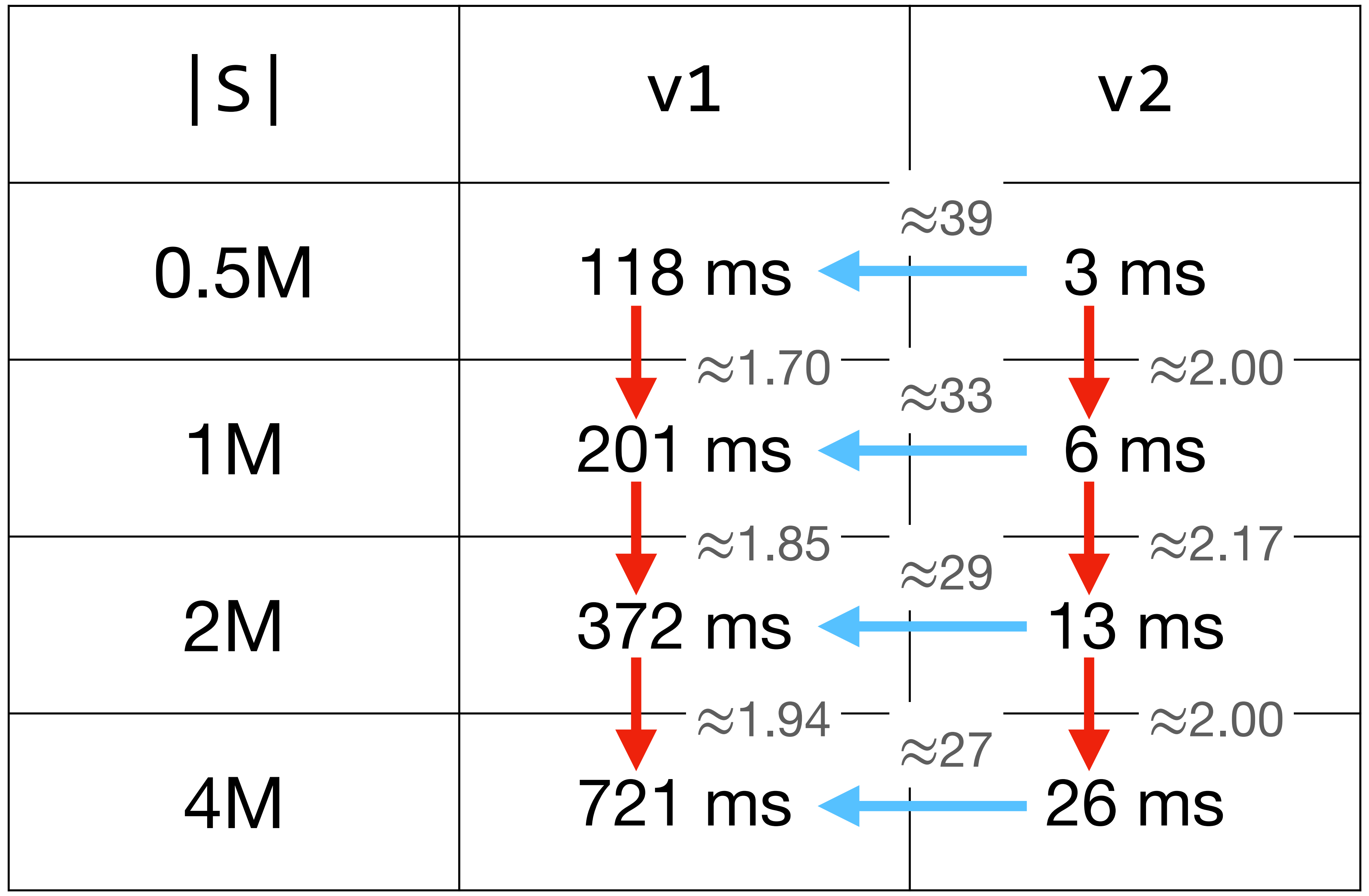
The running time — The scientific method



```
occ_count(S,x):                                                  v1
    count = 0
    for i = 1..|S|:
      if S[i] is equal to x:
        count += 1
    return count                     ???

all_occ_count_v1(S):
    for each character x in ['a','b','c','d','e','f',...,'z']:
      occ = occ_count(S,x)
      print(x,occ)
```



```
all_occ_count_v2(S):       v2
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
      j = int(S[i])         ???
      C[j] += 1
    for i = 1..|C|:
      print(char(i),C[i])
```

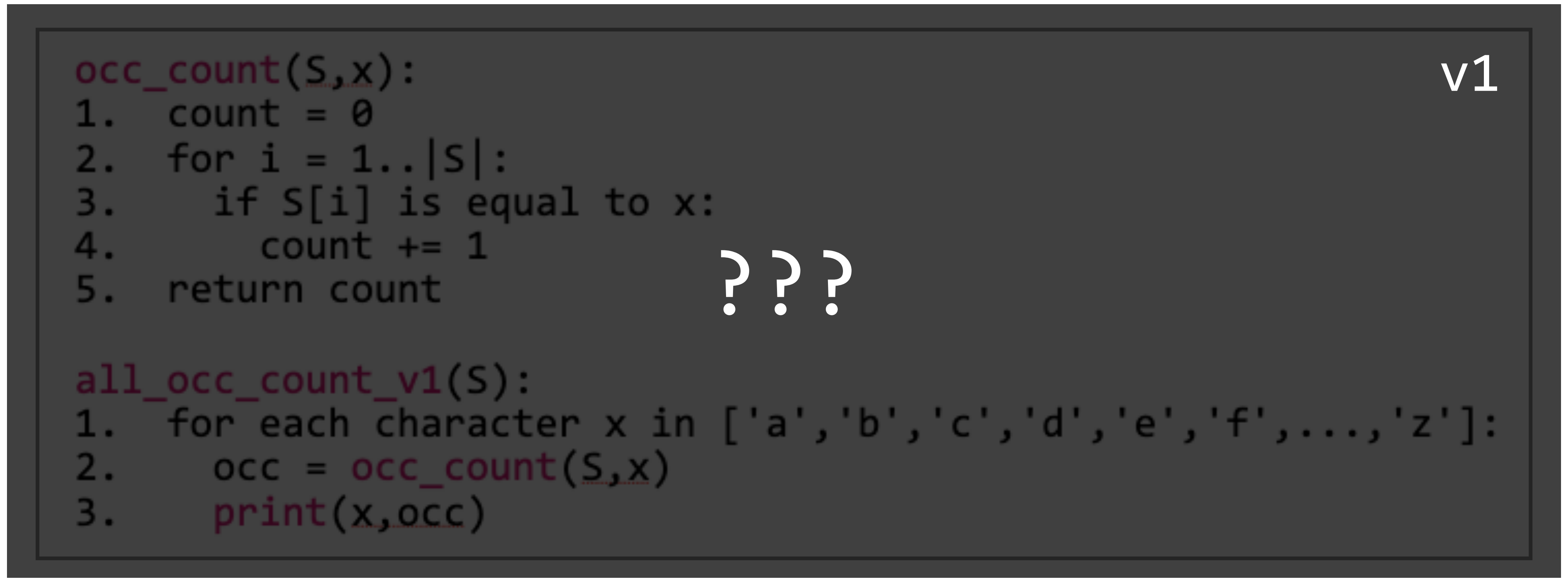


|S|	v1	v2
0.5M	118 ms	3 ms
1M	201 ms	6 ms
2M	372 ms	13 ms
4M	721 ms	26 ms

M = 1 million; 1 ms = 1/1000 sec

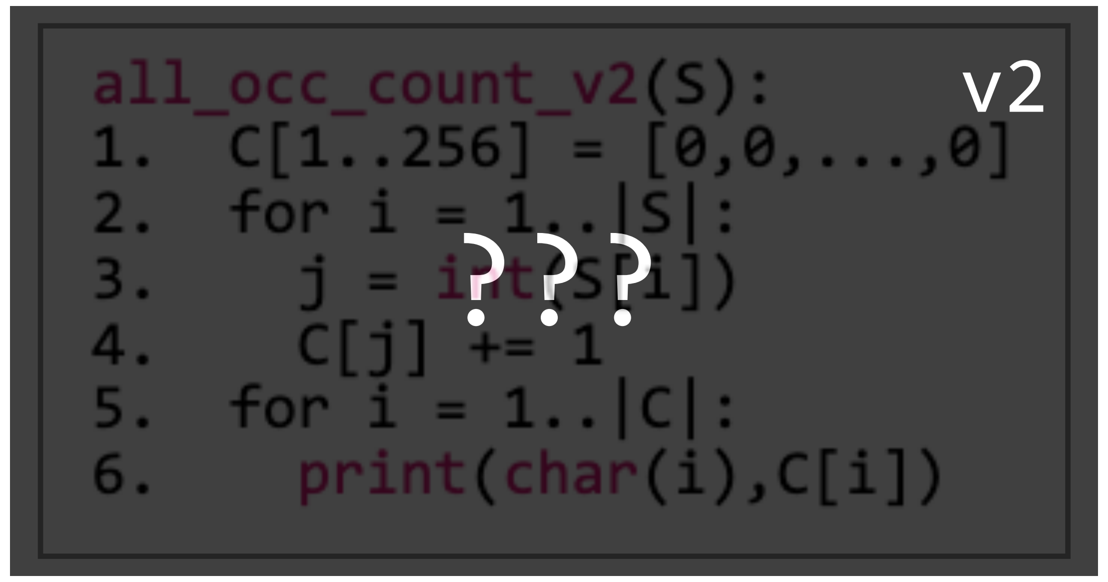
- First observation: as the **input doubles in size**, **also the running time** of both v1 and v2 **doubles**.
- First hypothesis: the running time has a **linear** dependency from the input size.

The running time — The scientific method



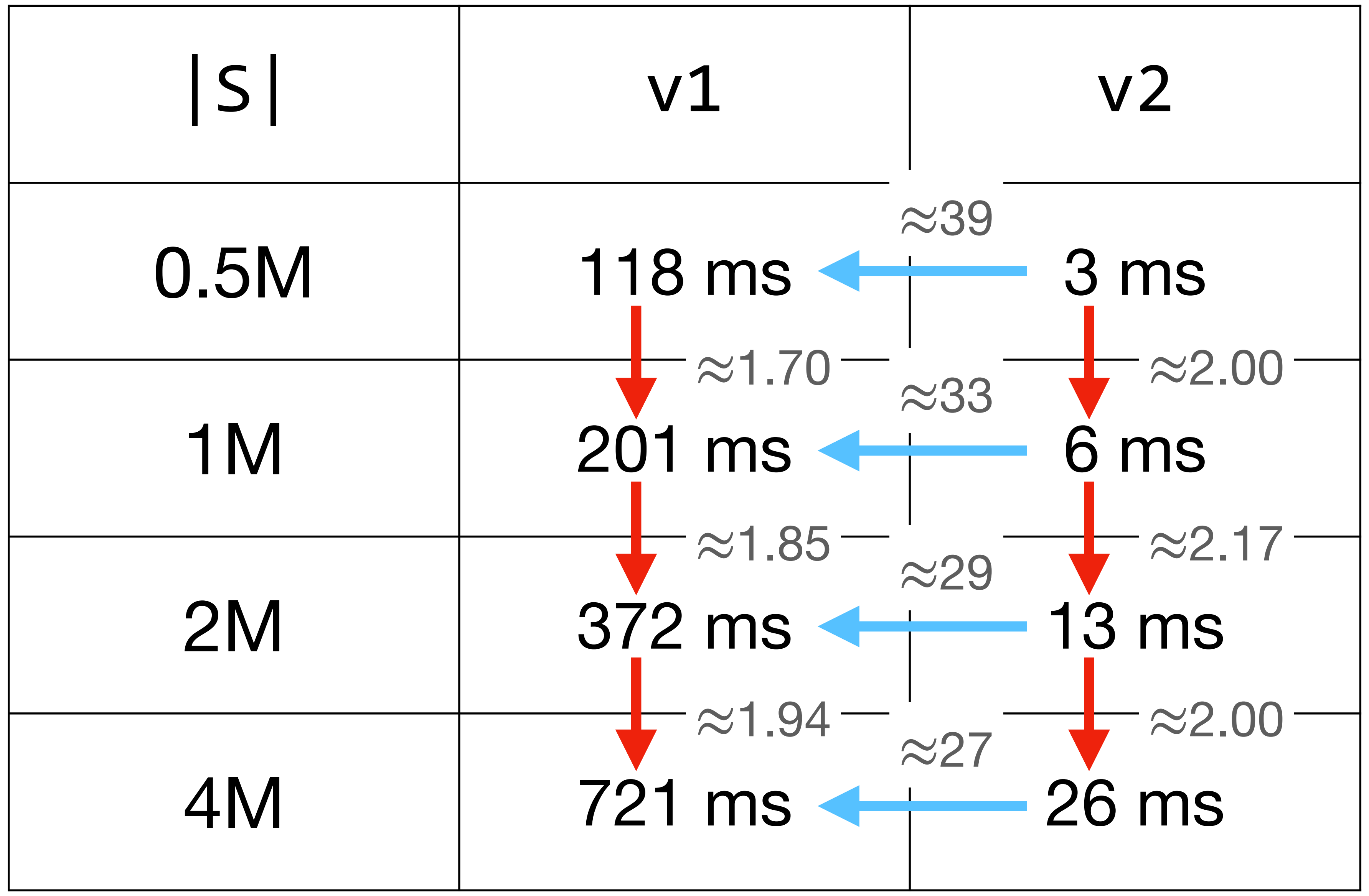
v1

```
occ_count(S,x):
  count = 0
  for i = 1..|S|:
    if S[i] is equal to x:
      count += 1
  return count

all_occ_count_v1(S):
  for each character x in ['a','b','c','d','e','f',...,'z']:
    occ = occ_count(S,x)
    print(x,occ)
```



v2

```
all_occ_count_v2(S):
  C[1..256] = [0,0,...,0]
  for i = 1..|S|:
    j = int(S[i])
    C[j] += 1
  for i = 1..|C|:
    print(char(i),C[i])
```



|S|	v1	v2
0.5M	118 ms	3 ms
1M	201 ms	6 ms
2M	372 ms	13 ms
4M	721 ms	26 ms

M = 1 million; 1 ms = 1/1000 sec

- First observation: as the **input doubles in size**, **also the running time** of both v1 and v2 **doubles**.
- First hypothesis: the running time has a **linear** dependency from the input size.
- Second observation: v1 tends to be ≈27-30✕ **slower** than v2 for large inputs.

The running time — The scientific method

- The scientific method is great to validate our hypotheses.
- But one should come up with an hypothesis first. We derived our hypothesis via **direct observation** of the running time.

The running time — The scientific method

- The scientific method is great to validate our hypotheses.
- But one should come up with an hypothesis first. We derived our hypothesis via **direct observation** of the running time.
- However, looking at the running time alone does not explain **what** the algorithm is doing.
- We would like to have a **model** to **predict the running time.**

The running time — Deriving a model

- Intuitively: the running time of an algorithm is the **sum of the costs of all the operations it executes**.

- **Q.** What is an "operation" ?

The running time — Deriving a model

- Intuitively: the running time of an algorithm is the **sum of the costs of all the operations it executes**.
- **Q.** What is an "operation" ?
- By "operation" we mean some **elementary operation** that a computer can execute, like: assignments, addition/subtraction, multiplication/division, read a cell of an array, comparing two integers/characters, etc.
- **Simplification**: such elementary operations take a (usually, very small) unit of time, say c.

```
Example 1:

x = 1
y = 2
z = x + y
```

4 ops

```
Example 2:

S[3] = 5
z = S[3] * 4
```

5 ops

```
Example 3:

for i = 1..|S|:
  x = i + 3
```

~2|S| ops

Counting occurrences — Analysis

- Let's **count the number of operations** our two algorithms perform. Let $n = |S|$.

```
occ_count(S,x):
count = 0
for i = 1..|S|:
  if S[i] is equal to x:
    count += 1
return count
```

```
all_occ_count_v1(S):
for each character x in ['a','b','c','d','e','f',...,'z']:
  occ = occ_count(S,x)
  print(x,occ)
```

```
all_occ_count_v2(S):
C[1..256] = [0,0,...,0]
for i = 1..|S|:
  j = int(S[i])
  C[j] += 1
for i = 1..|C|:
  print(char(i),C[i])
```

Counting occurrences — Analysis

- Let's **count the number of operations** our two algorithms perform. Let $n = |S|$.

```
occ_count(S,x):
  count = 0
  for i = 1..|S|:
    if S[i] is equal to x:
      count += 1
  return count
```

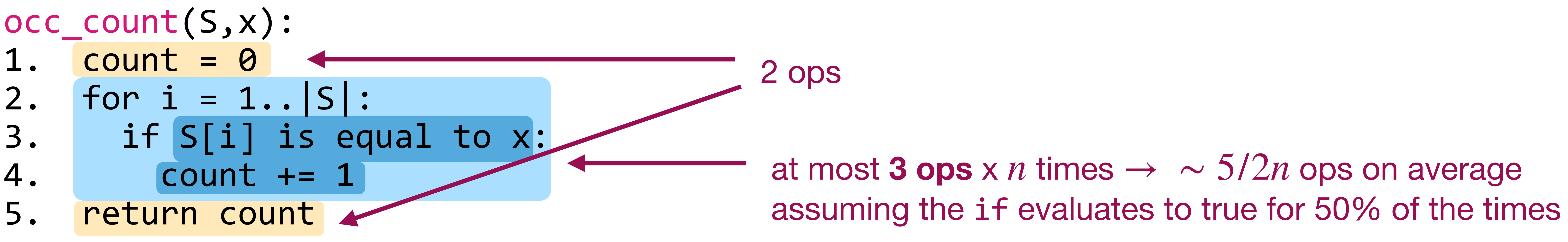
at most **3 ops** x n times → $\sim 5/2n$ ops on average
assuming the `if` evaluates to true for 50% of the times

```
all_occ_count_v1(S):
  for each character x in ['a','b','c','d','e','f',...,'z']:
    occ = occ_count(S,x)
    print(x,occ)
```

```
all_occ_count_v2(S):
  C[1..256] = [0,0,...,0]
  for i = 1..|S|:
    j = int(S[i])
    C[j] += 1
  for i = 1..|C|:
    print(char(i),C[i])
```

Counting occurrences — Analysis

- Let's **count the number of operations** our two algorithms perform. Let $n = |S|$.



```
occ_count(S,x):
  count = 0
  for i = 1..|S|:
    if S[i] is equal to x:
      count += 1
  return count
```

2 ops

at most **3 ops** x n times → $\sim 5/2n$ ops on average
assuming the `if` evaluates to true for 50% of the times

```
all_occ_count_v1(S):
  for each character x in ['a','b','c','d','e','f',...,'z']:
    occ = occ_count(S,x)
    print(x,occ)
```

```
all_occ_count_v2(S):
  C[1..256] = [0,0,...,0]
  for i = 1..|S|:
    j = int(S[i])
    C[j] += 1
  for i = 1..|C|:
    print(char(i),C[i])
```

Counting occurrences — Analysis

- Let's **count the number of operations** our two algorithms perform. Let $n = |S|$.

```
occ_count(S,x):
  count = 0
  for i = 1..|S|:
    if S[i] is equal to x:
      count += 1
  return count
```

2 ops

at most **3 ops** x n times → $\sim 5/2n$ ops on average
assuming the `if` evaluates to true for 50% of the times

```
all_occ_count_v1(S):
  for each character x in ['a','b','c','d','e','f',...,'z']:
    occ = occ_count(S,x)
    print(x,occ)
```

26 calls to the function `occ_count` that takes
$\sim 5/2n$ ops on average → $\sim 65n$ total ops

```
all_occ_count_v2(S):
  C[1..256] = [0,0,...,0]
  for i = 1..|S|:
    j = int(S[i])
    C[j] += 1
  for i = 1..|C|:
    print(char(i),C[i])
```

Counting occurrences — Analysis

- Let's **count the number of operations** our two algorithms perform. Let $n = |S|$.

```
occ_count(S,x):
    count = 0
    for i = 1..|S|:
        if S[i] is equal to x:
            count += 1
    return count
```

2 ops

at most **3 ops** x n times → $\sim 5/2n$ ops on average
assuming the `if` evaluates to true for 50% of the times

```
all_occ_count_v1(S):
    for each character x in ['a','b','c','d','e','f',...,'z']:
        occ = occ_count(S,x)
        print(x,occ)
```

26 calls to the function `occ_count` that takes
$\sim 5/2n$ ops on average → $\sim 65n$ total ops

```
all_occ_count_v2(S):
    C[1..256] = [0,0,...,0]
    for i = 1..|S|:
        j = int(S[i])
        C[j] += 1
    for i = 1..|C|:
        print(char(i),C[i])
```

256 ops

$\sim 3n$ ops

256 ops

a total of $\sim 3n$ + 256 x 2 ops $\approx 3n$
when n is large (e.g., n = 1 million)

Counting occurrences — Analysis

- To sum up.

	v1	v2
num. operations	$\sim 65n$	$\sim 3n$

- We can conclude that:
  - Both v1 and v2 have running time that grows **linearly** in n (the length of the input string).
  - But v2 executes way fewer operations, hence it is **much** faster (≈20-30X faster).

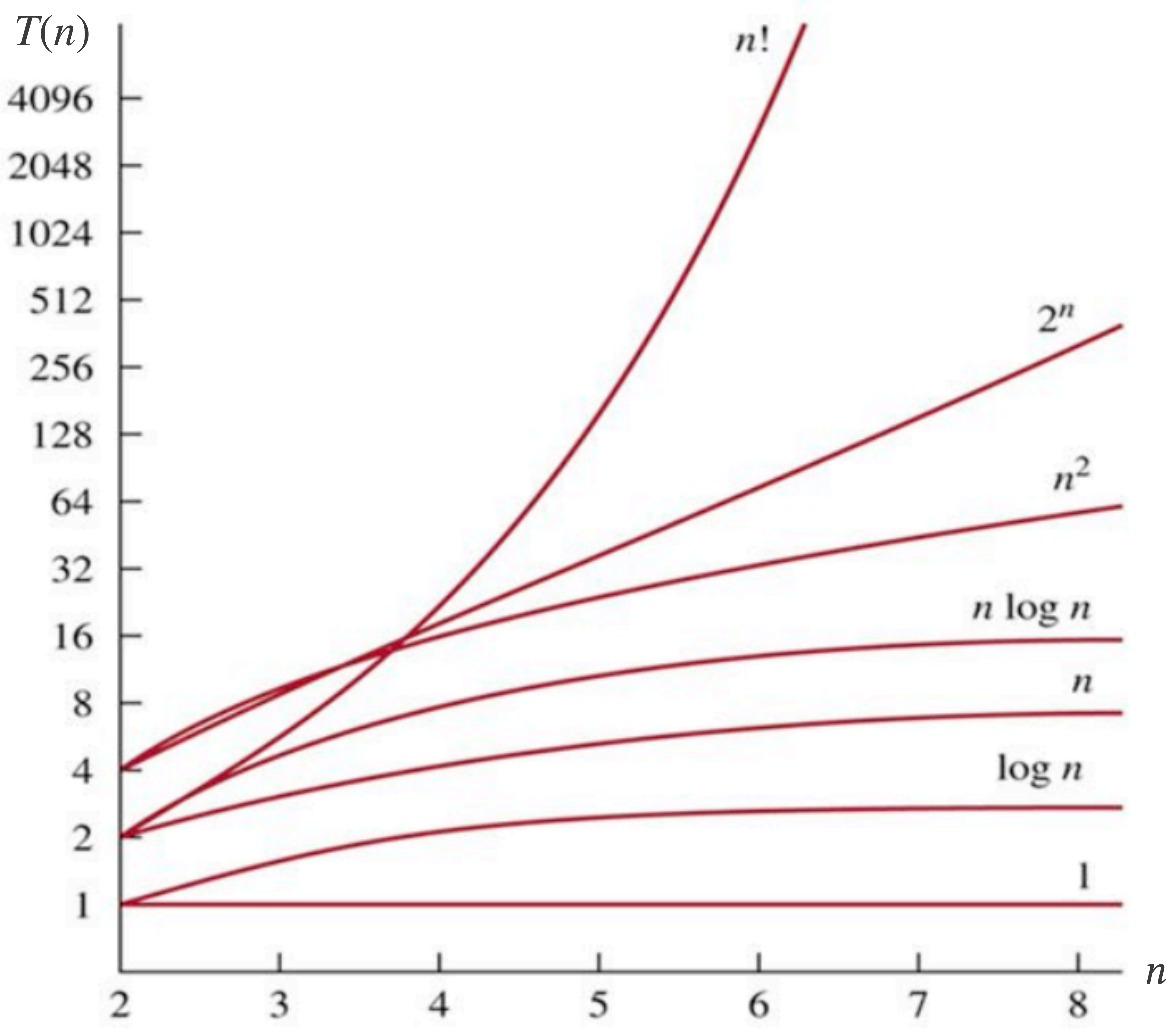
Counting occurrences — Analysis

- To sum up.

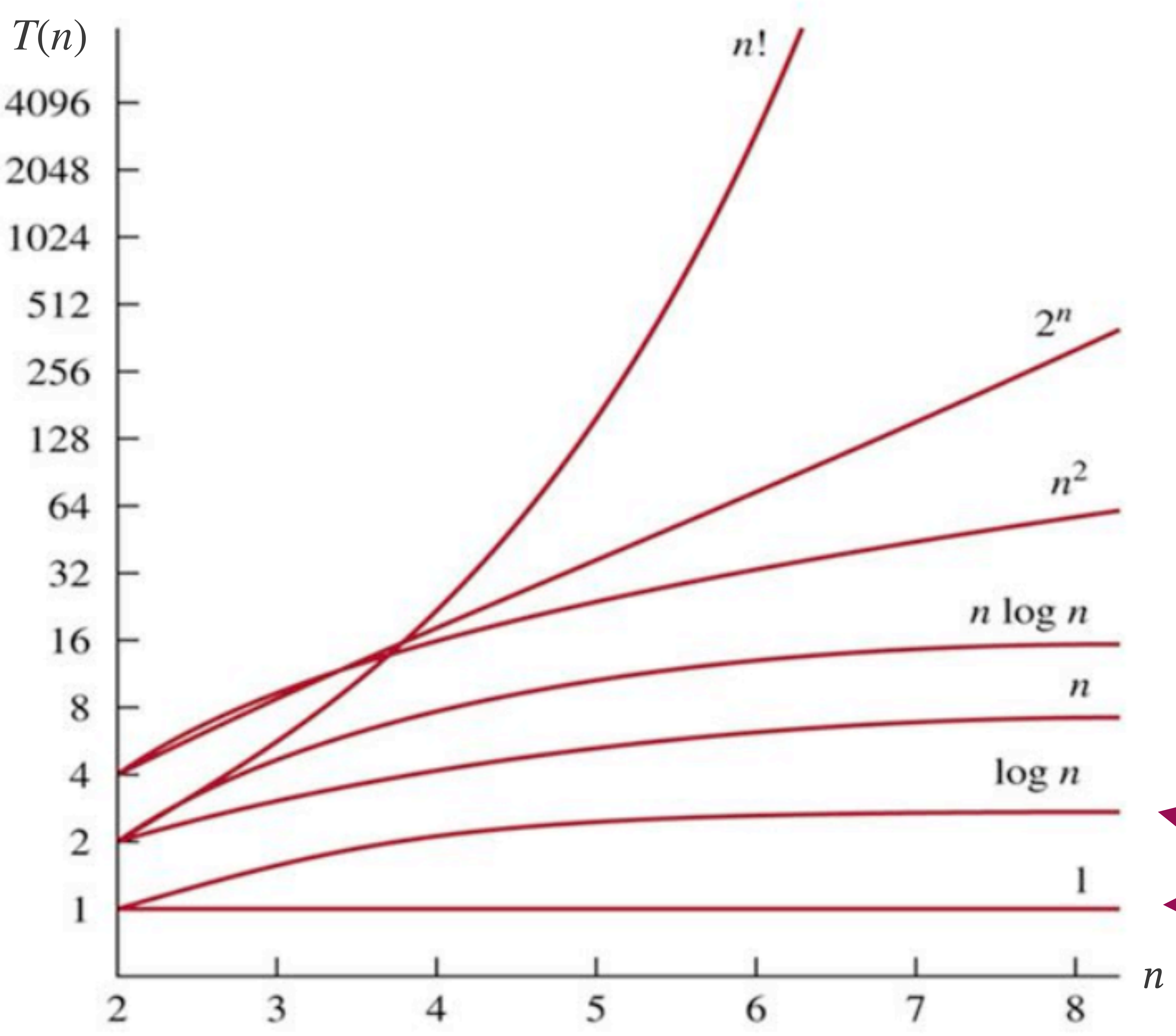
	v1	v2
num. operations	$\sim 65n$	$\sim 3n$

- We can conclude that:
    - Both v1 and v2 have running time that grows **linearly** in n (the length of the input string).
    - But v2 executes way fewer operations, hence it is **much** faster (≈20-30X faster).
- **Linear running time is not the only possibility!**

Growth of running time

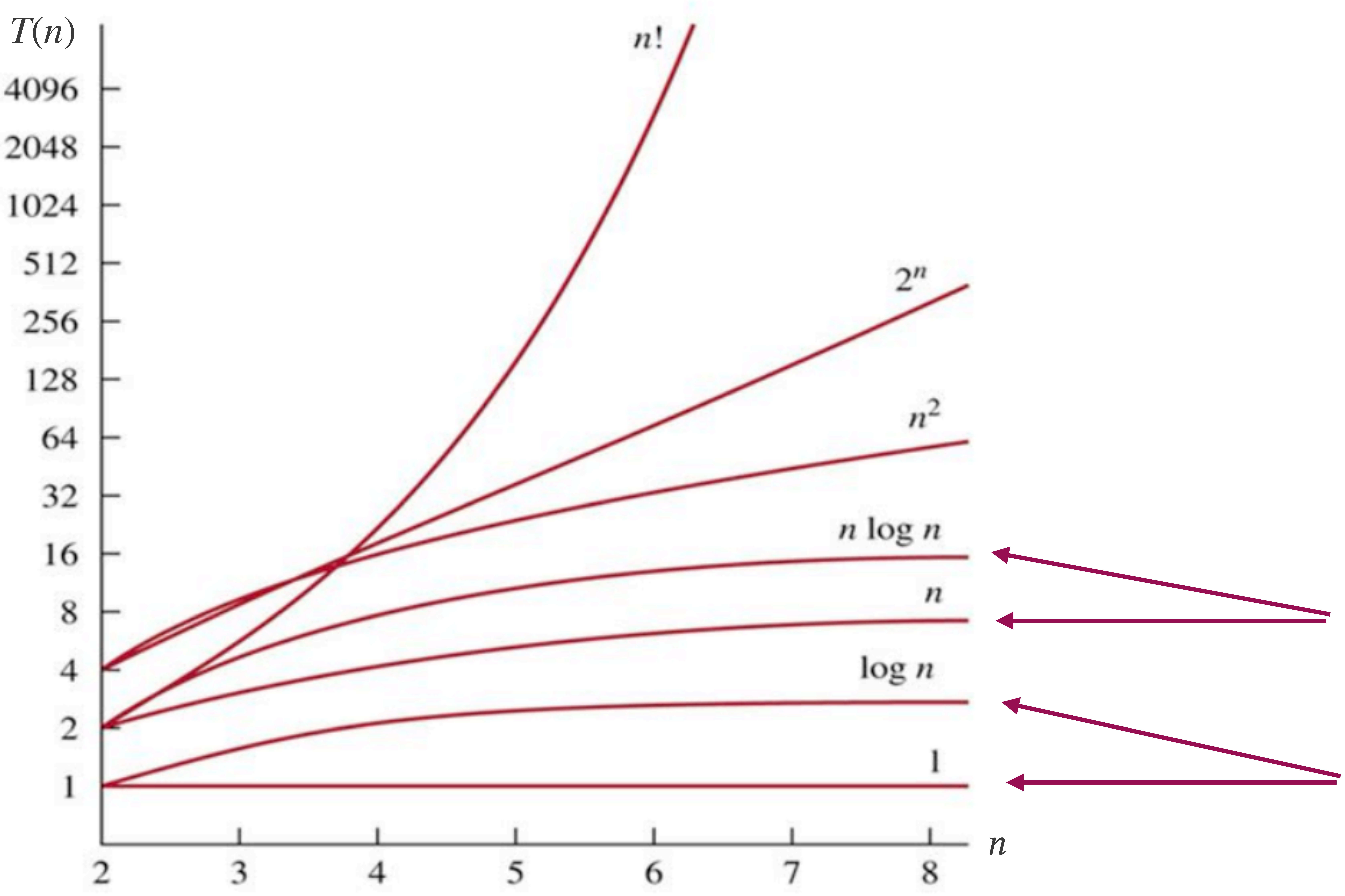


Growth of running time



very fast!

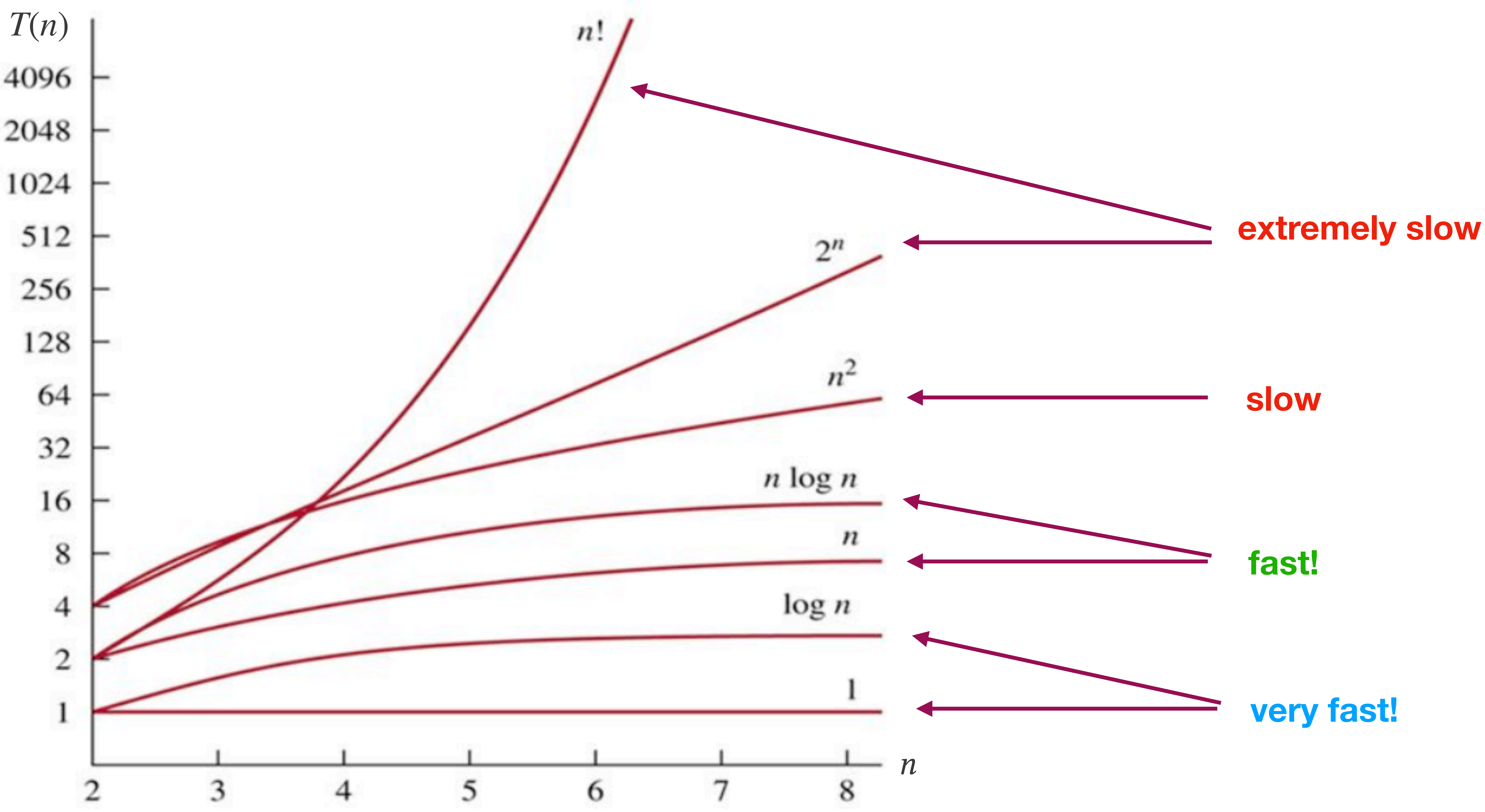
Growth of running time



fast!

very fast!

Growth of running time



extremely slow

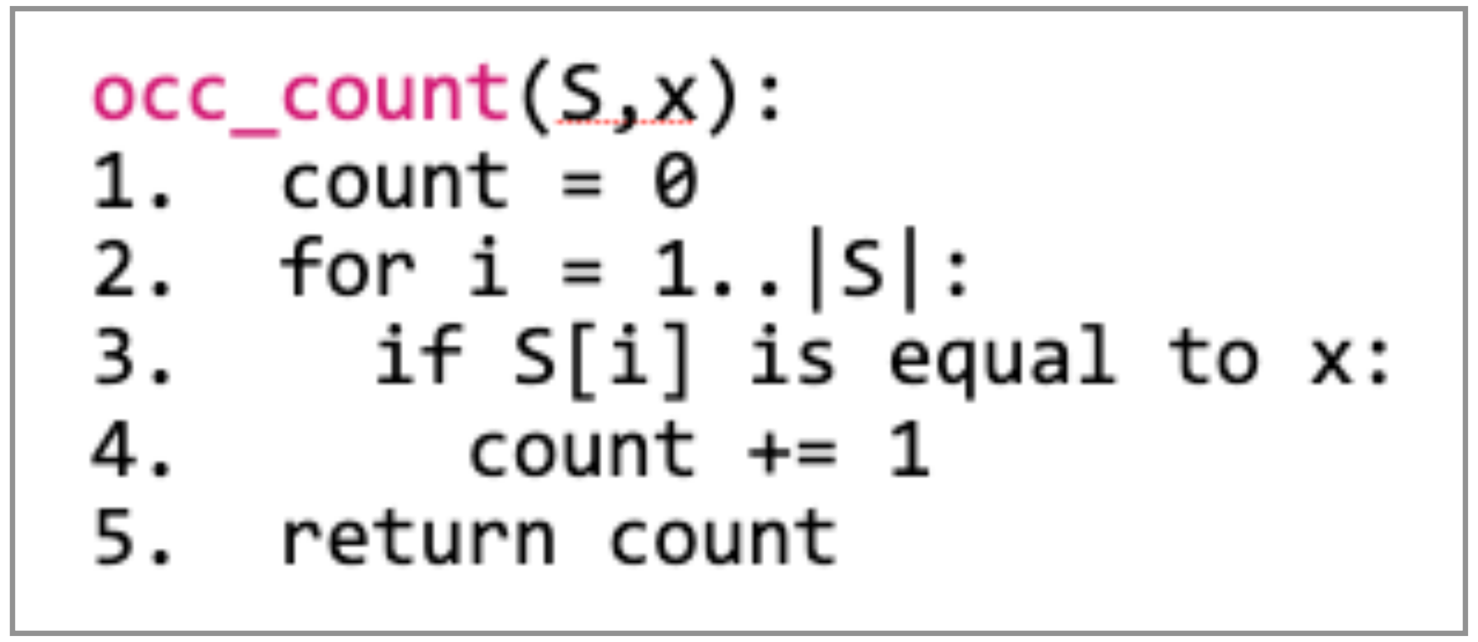
slow

fast!

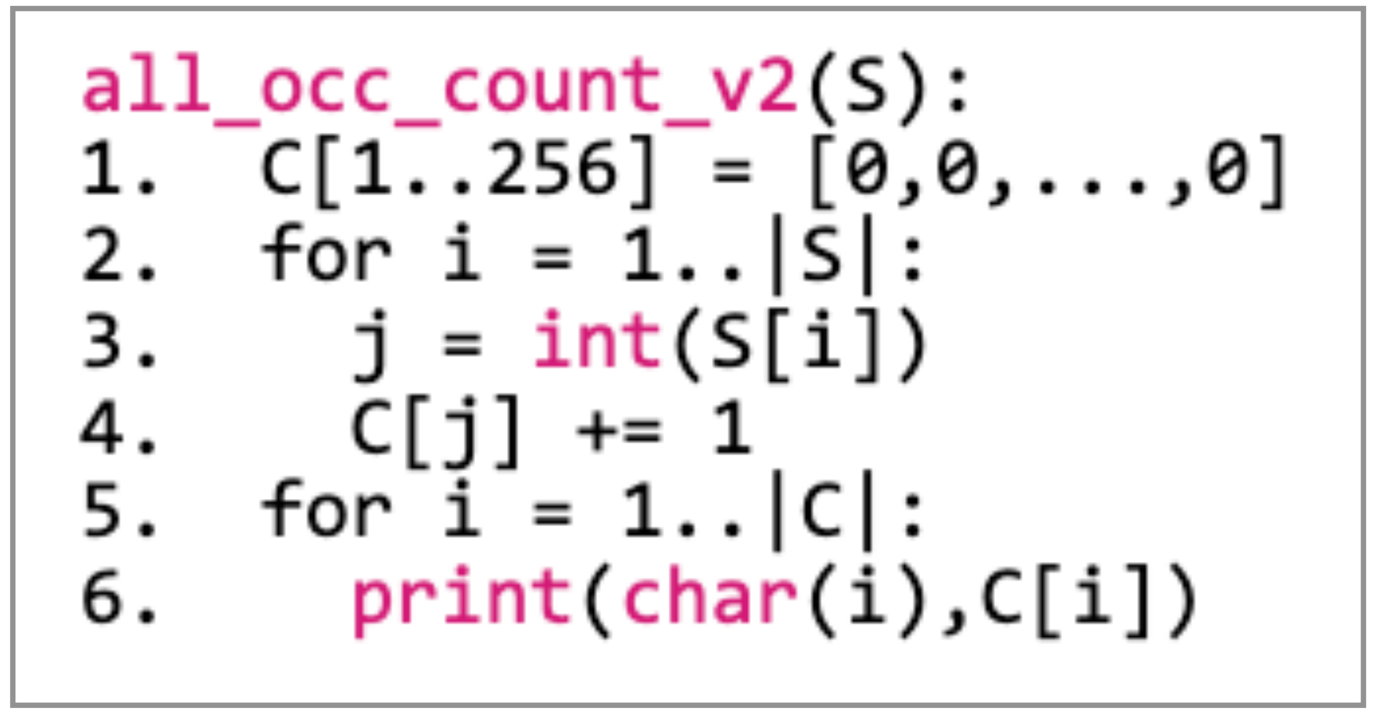
very fast!

The space

- Intuitively: all the bytes that are maintained/manipulated by the algorithm during its execution.



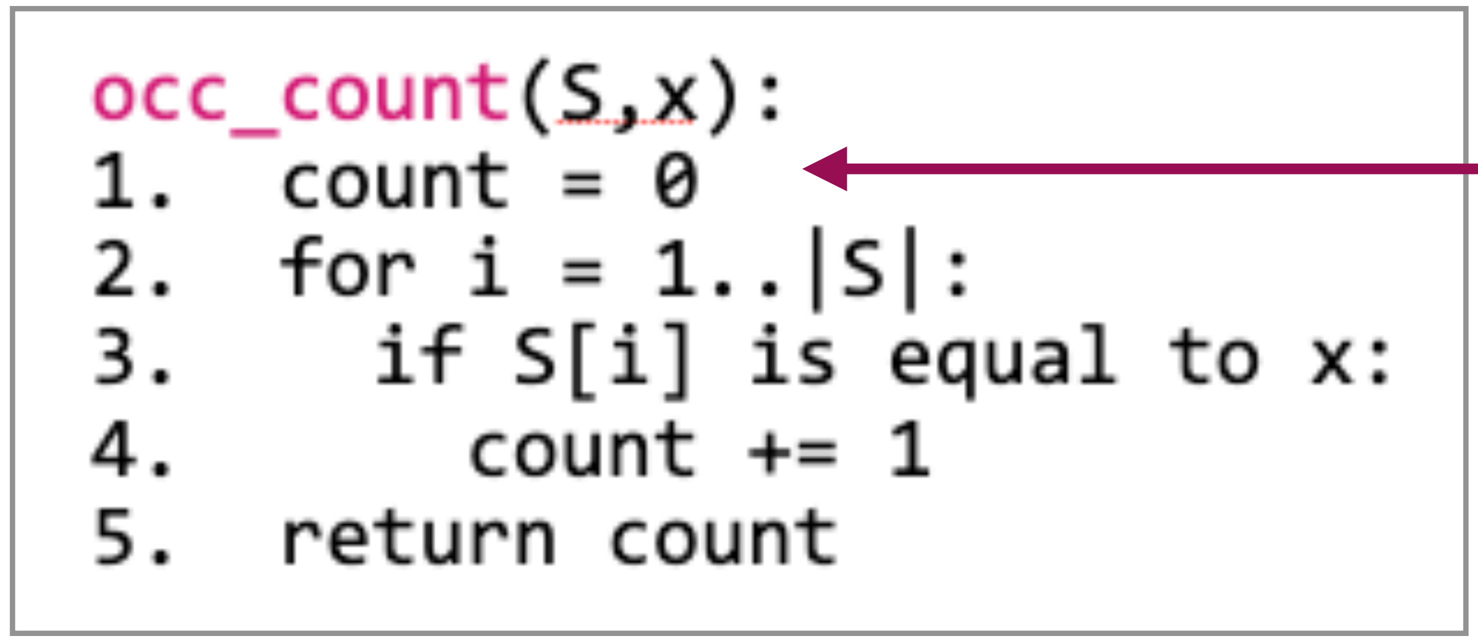
```
occ_count(S,x):
  count = 0
  for i = 1..|S|:
    if S[i] is equal to x:
      count += 1
  return count
```



```
all_occ_count_v2(S):
  C[1..256] = [0,0,...,0]
  for i = 1..|S|:
    j = int(S[i])
    C[j] += 1
  for i = 1..|C|:
    print(char(i),C[i])
```

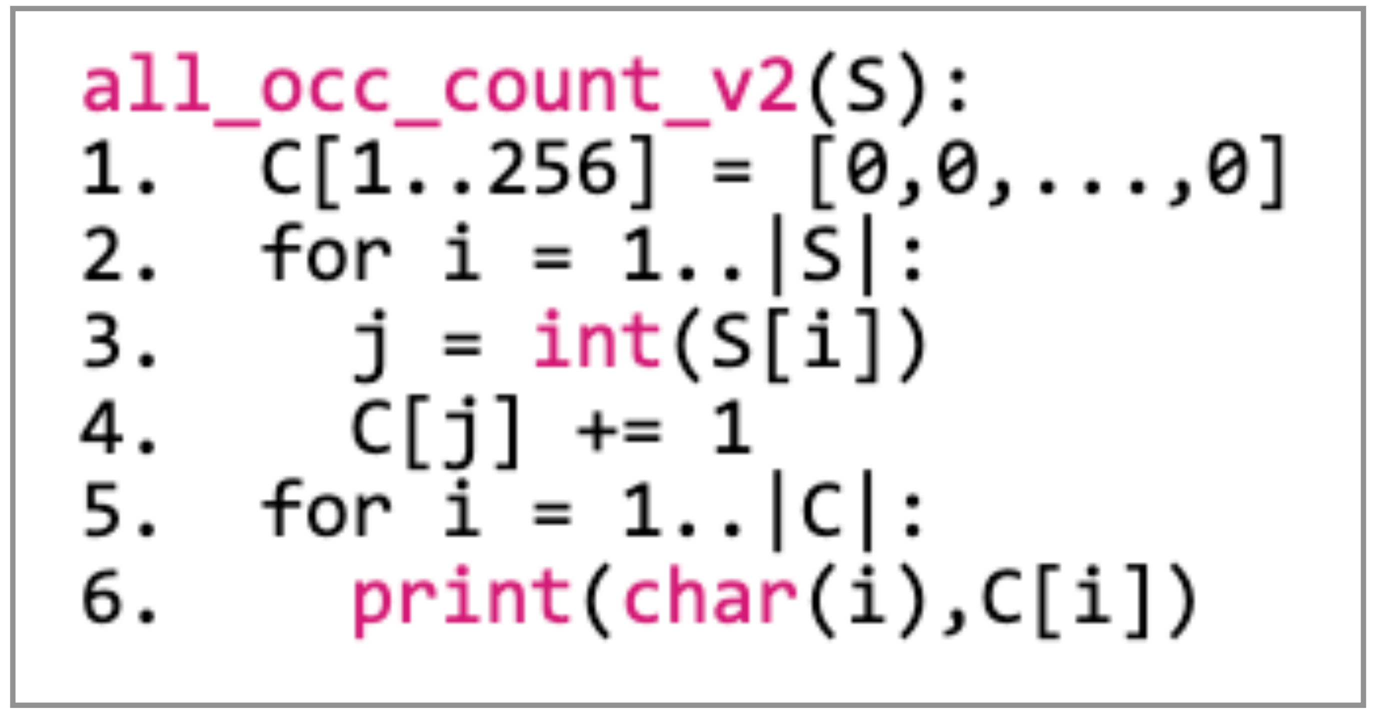
The space

- Intuitively: all the bytes that are maintained/manipulated by the algorithm during its execution.



```
occ_count(S,x):
  count = 0
  for i = 1..|S|:
    if S[i] is equal to x:
      count += 1
  return count
```

4-byte integer



```
all_occ_count_v2(S):
  C[1..256] = [0,0,...,0]
  for i = 1..|S|:
    j = int(S[i])
    C[j] += 1
  for i = 1..|C|:
    print(char(i),C[i])
```

4 bytes x 256 = **1024** bytes = 1 KiB

Part 2 — Summary

- Three good reasons to study algorithms:
  - understand; solve; earn.
- Analysis of algorithms:
  - scientific method is good to confirm/reject hypotheses;
  - we need a model to predict the running time and space consumed by an algorithm.
- Model: count the number of operations performed by an algorithm.
- Alg. `v2` is 30X faster than algorithm `v1` but also consumes 1KiB of extra memory.

Part 3 — Some example problems: integer search and sub-string search

Integer search

- **Problem.** We are given a **sorted** integer array `A`, say of length n, and an integer `x`. We want to determine whether `x` is in `A` and, if so, return its **position** in `A`.

Integer search

- **Problem.** We are given a **sorted** integer array `A`, say of length n, and an integer `x`. We want to determine whether `x` is in `A` and, if so, return its **position** in `A`.
- Example.

```
A = [3,5,7,13,14,15,34,45,66,78,123,443,601]
     1 2 3  4  5  6  7  8  9 10  11  12  13
```

`x=34` ✓ (return 7)

Integer search

- **Problem.** We are given a **sorted** integer array `A`, say of length n, and an integer `x`. We want to determine whether `x` is in `A` and, if so, return its **position** in `A`.
- Example.

```
A = [3,5,7,13,14,15,34,45,66,78,123,443,601]
     1 2 3  4  5  6  7  8  9 10  11  12  13
```

`x=34` ✓ (return 7)

`x=95` ✗ (return -1)

Integer search

- **Problem.** We are given a **sorted** integer array `A`, say of length n, and an integer `x`. We want to determine whether `x` is in `A` and, if so, return its **position** in `A`.
- Example.

```
A = [3,5,7,13,14,15,34,45,66,78,123,443,601]
     1 2 3  4  5  6  7  8  9 10  11  12  13
```

`x=34` ✓ (return 7)

`x=95` ✗ (return -1)

- We will see **two algorithms** to solve this problem, with **radically different** running times.

Linear search

- **Idea 1.** For each integer `A[i]`, `i = 1..n`, check if it is equal to `x`. If so, return `i`. If no integer is equal to `x`, then return `-1`.

- Pseudo code.

```
linear_search(A,x):
  for i = 1..n:
    if A[i] is equal to x:
      return i
  return -1
```

Linear search

- **Idea 1.** For each integer `A[i]`, `i = 1..n`, check if it is equal to `x`. If so, return `i`. If no integer is equal to `x`, then return `-1`.

- Pseudo code.

```
linear_search(A,x):
  for i = 1..n:
    if A[i] is equal to x:
      return i
  return -1
```

- Example.

```
A = [3,5,7,13,14,15,34,45,66,78,123,443,601]
     1 2 3  4  5  6  7  8  9 10  11  12  13

x = 34
```

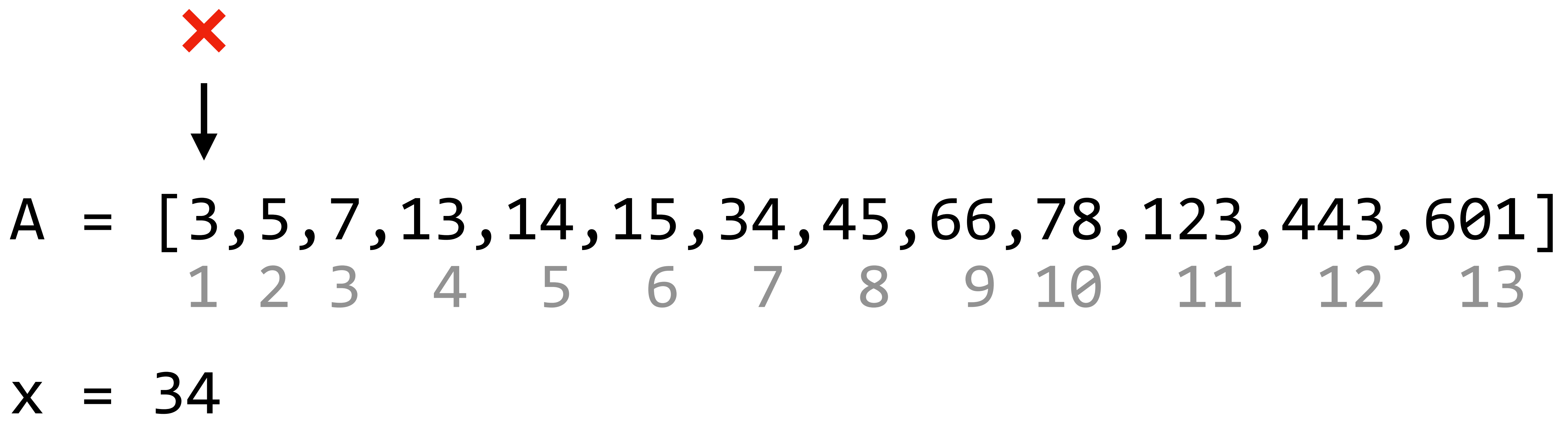
Linear search

- **Idea 1.** For each integer `A[i]`, `i = 1..n`, check if it is equal to `x`. If so, return `i`. If no integer is equal to `x`, then return `-1`.

- Pseudo code.

```
linear_search(A,x):
  for i = 1..n:
    if A[i] is equal to x:
      return i
  return -1
```

- Example.



```
A = [3,5,7,13,14,15,34,45,66,78,123,443,601]
     1 2 3  4  5  6  7  8  9 10  11  12  13

x = 34
```

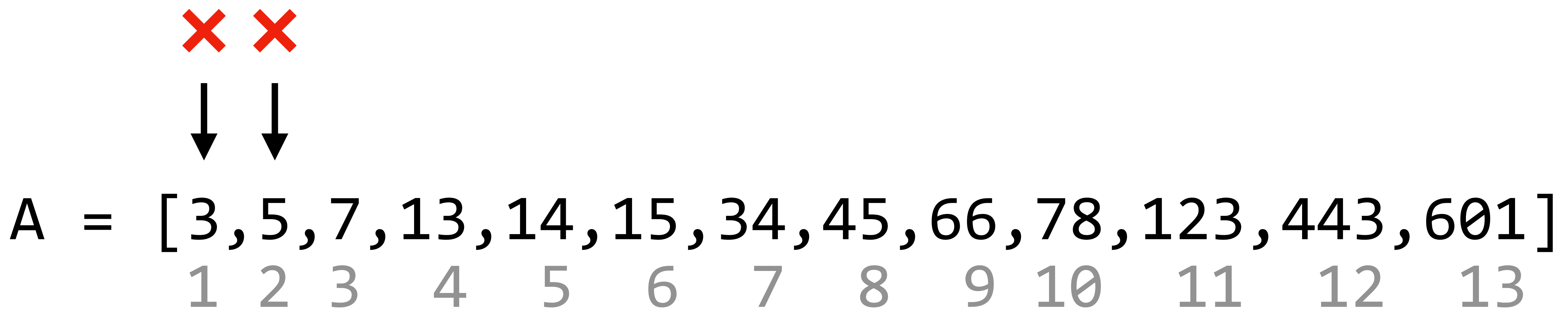
Linear search

- **Idea 1.** For each integer `A[i]`, `i = 1..n`, check if it is equal to `x`. If so, return `i`. If no integer is equal to `x`, then return `-1`.

- Pseudo code.

```
linear_search(A,x):
  for i = 1..n:
    if A[i] is equal to x:
      return i
  return -1
```

- Example.



```
A = [3,5,7,13,14,15,34,45,66,78,123,443,601]
     1 2 3  4  5  6  7  8  9 10  11  12  13
```

```
x = 34
```

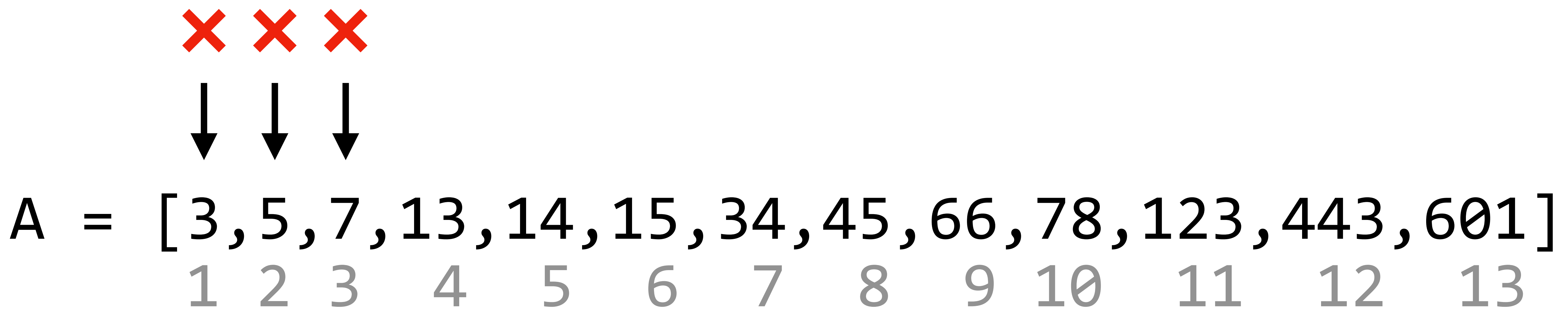
Linear search

- **Idea 1.** For each integer `A[i]`, `i = 1..n`, check if it is equal to x. If so, return `i`. If no integer is equal to x, then return `-1`.

- Pseudo code.

```
linear_search(A,x):
  for i = 1..n:
    if A[i] is equal to x:
      return i
  return -1
```

- Example.



```
A = [3,5,7,13,14,15,34,45,66,78,123,443,601]
     1 2 3  4  5  6  7  8  9 10  11  12  13
```

```
x = 34
```

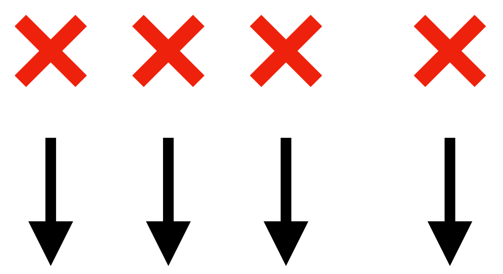
Linear search

- **Idea 1.** For each integer `A[i]`, `i = 1..n`, check if it is equal to `x`. If so, return `i`. If no integer is equal to `x`, then return `-1`.

- Pseudo code.

```
linear_search(A,x):
  for i = 1..n:
    if A[i] is equal to x:
      return i
  return -1
```

- Example.



```
A = [3,5,7,13,14,15,34,45,66,78,123,443,601]
     1 2 3  4  5  6  7  8  9 10  11  12  13

x = 34
```

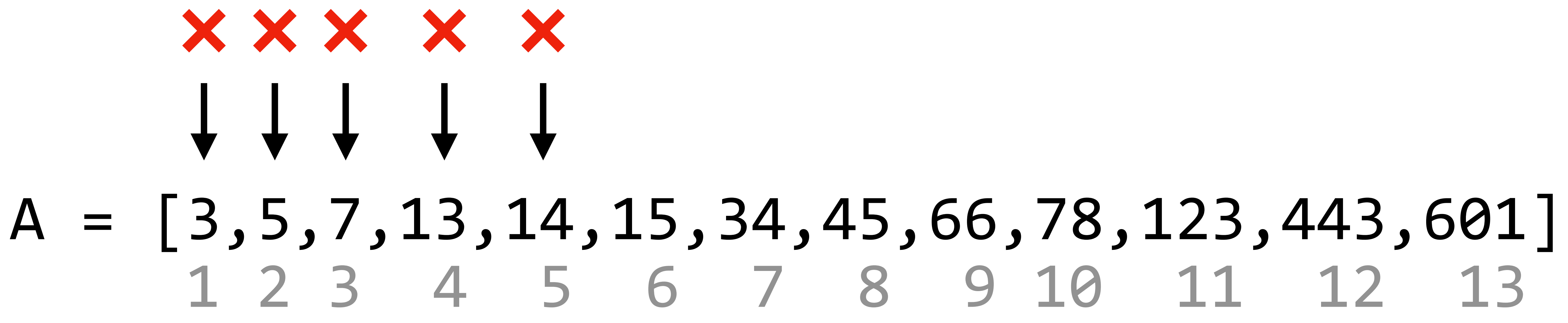
Linear search

- **Idea 1.** For each integer `A[i]`, `i = 1..n`, check if it is equal to `x`. If so, return `i`. If no integer is equal to `x`, then return `-1`.

- Pseudo code.

```
linear_search(A,x):
  for i = 1..n:
    if A[i] is equal to x:
      return i
  return -1
```

- Example.



```
A = [3,5,7,13,14,15,34,45,66,78,123,443,601]
     1 2 3 4  5  6  7  8  9  10 11  12  13
```

```
x = 34
```

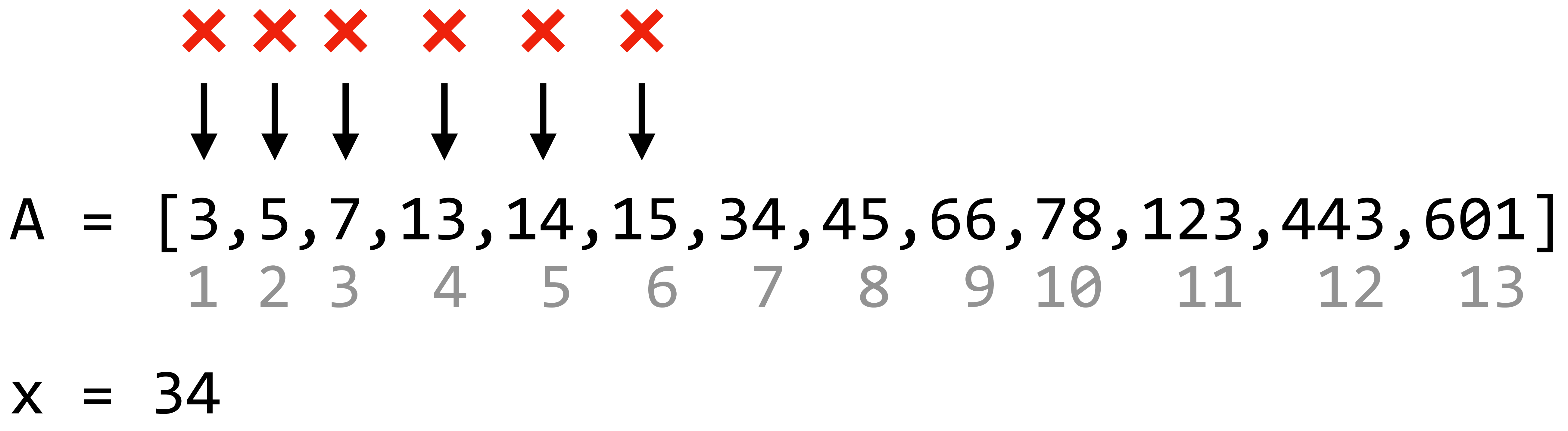
Linear search

- **Idea 1.** For each integer `A[i]`, `i = 1..n`, check if it is equal to `x`. If so, return `i`. If no integer is equal to `x`, then return `-1`.

- Pseudo code.

```
linear_search(A,x):
  for i = 1..n:
    if A[i] is equal to x:
      return i
  return -1
```

- Example.



```
A = [3,5,7,13,14,15,34,45,66,78,123,443,601]
     1 2 3  4  5  6  7  8  9 10  11  12  13

x = 34
```

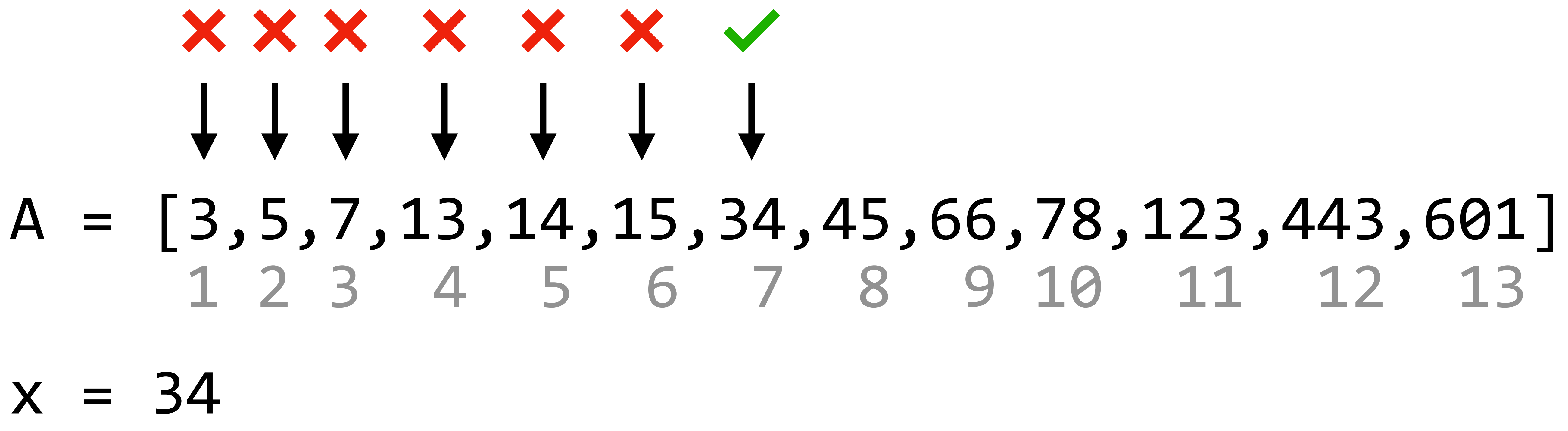
Linear search

- **Idea 1.** For each integer `A[i]`, `i = 1..n`, check if it is equal to x. If so, return `i`. If no integer is equal to x, then return `-1`.

- Pseudo code.

```
linear_search(A,x):
  for i = 1..n:
    if A[i] is equal to x:
      return i
  return -1
```

- Example.



```
A = [3,5,7,13,14,15,34,45,66,78,123,443,601]
     1 2 3  4  5  6  7  8  9 10  11  12  13

x = 34
```

Linear search

- **Idea 1.** For each integer `A[i]`, `i = 1..n`, check if it is equal to `x`. If so, return `i`. If no integer is equal to `x`, then return `-1`

- Pseudo code.

```
linear_search(A,x):
  for i = 1..n:
    if A[i] is equal to x:
      return i
  return -1
```

- **Q.** How many operations?

Linear search

- **Idea 1.** For each integer `A[i]`, `i = 1..n`, check if it is equal to `x`. If so, return `i`. If no integer is equal to `x`, then return `-1`

- Pseudo code.

```
linear_search(A,x):
  for i = 1..n:
    if A[i] is equal to x:
      return i
  return -1
```

- **Q.** How many operations?
  - **Best case**: `x` is found in first position (`i=1`), so just 2 ops.

Linear search

- **Idea 1.** For each integer `A[i]`, `i = 1..n`, check if it is equal to `x`. If so, return `i`. If no integer is equal to `x`, then return `-1`

- Pseudo code.

```
linear_search(A,x):
  for i = 1..n:
    if A[i] is equal to x:
      return i
  return -1
```

- **Q.** How many operations?
  - **Best case**: x is found in first position (`i=1`), so just 2 ops.
  - **Worst case**: x is **not found at all**, so $2n$ ops.

Linear search

- **Idea 1.** For each integer `A[i]`, `i = 1..n`, check if it is equal to `x`. If so, return `i`. If no integer is equal to `x`, then return `-1`

- Pseudo code.

```
linear_search(A,x):
  for i = 1..n:
    if A[i] is equal to x:
      return i
  return -1
```

- **Q.** How many operations?
  - **Best case**: x is found in first position (`i=1`), so just 2 ops.
  - **Worst case**: x is **not found at all**, so $2n$ ops.
  - **Average case**: $\sim 1/2 \cdot 2n = n$ ops.

Linear search

- **Idea 1.** For each integer `A[i]`, `i = 1..n`, check if it is equal to `x`. If so, return `i`. If no integer is equal to `x`, then return `-1`

- Pseudo code.

```
linear_search(A,x):
  for i = 1..n:
    if A[i] is equal to x:
      return i
  return -1
```

- **Q.** How many operations?
  - **Best case**: x is found in first position (`i=1`), so just 2 ops.
  - **Worst case**: x is **not found at all**, so $2n$ ops.
  - **Average case**: $\sim 1/2 \cdot 2n = n$ ops.

- So **the running time is linear** in the length of the array.

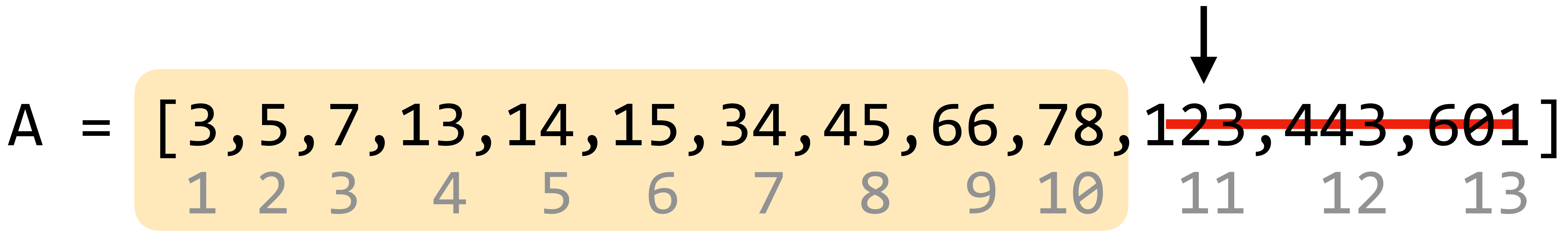
A better search strategy

- **Idea 2.** Exploit that fact that the array A is **sorted**.
- **Intuition**: Suppose you have `x=34` and you look at a random position in `A`, say at position `11`. What can you say about the position of `x`?

```
                                 ↓
A = [3,5,7,13,14,15,34,45,66,78,123,443,601]
     1 2 3  4  5  6  7  8  9 10  11  12  13

x = 34
```

A better search strategy

- **Idea 2.** Exploit that fact that the array A is **sorted**.
- **Intuition**: Suppose you have `x=34` and you look at a random position in `A`, say at position `11`. What can you say about the position of `x`?

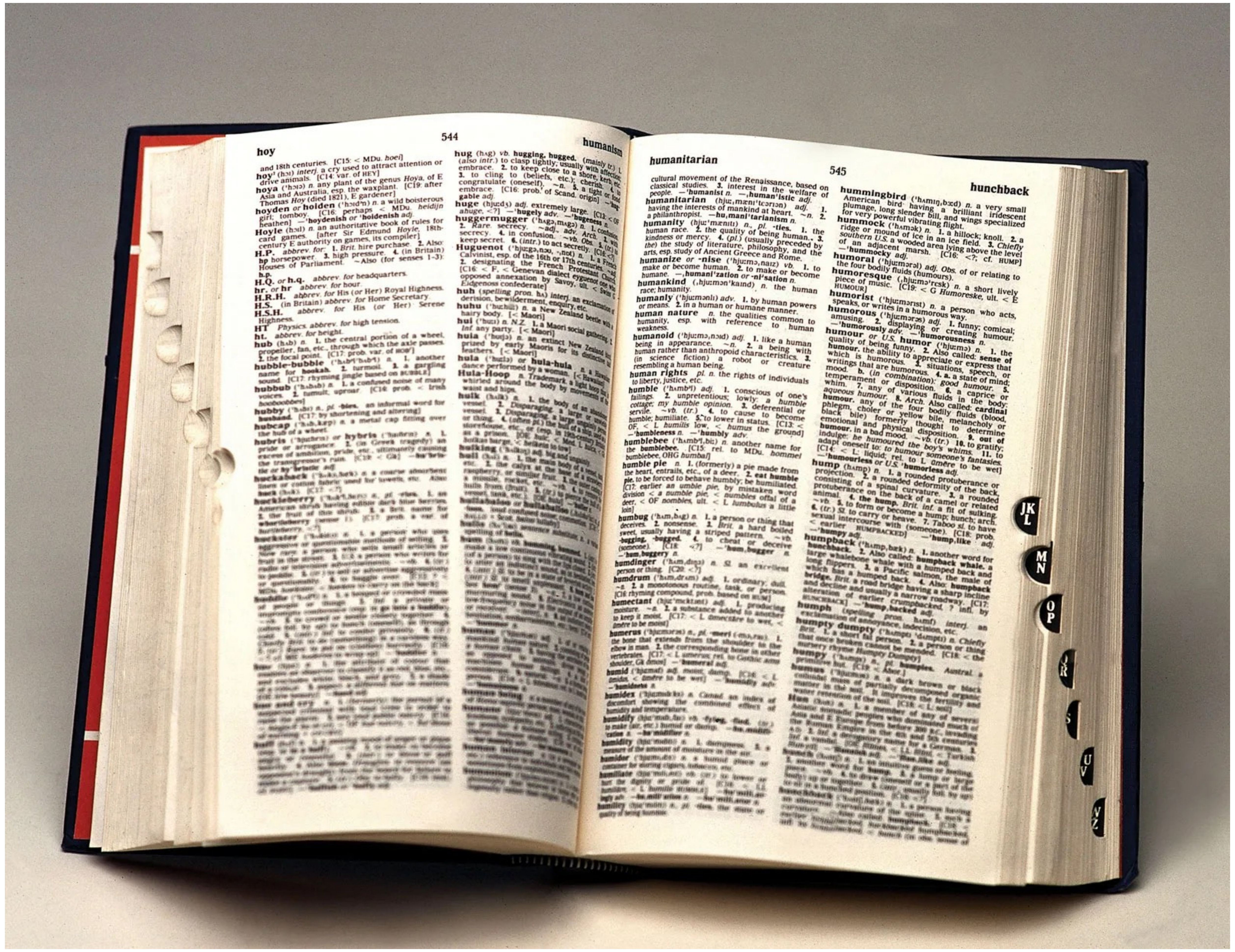


A = [3,5,7,13,14,15,34,45,66,78,~~123,443,601~~]

x = 34

A better search strategy

- If you think, this is exactly the way we search for a word in a dictionary!
- If we are searching for the word *"freshness"* we do not start from the beginning of the dictionary...but probably look for words that start with *f*.
- In fact, words in a vocabulary are **sorted** lexicographically...



Binary search

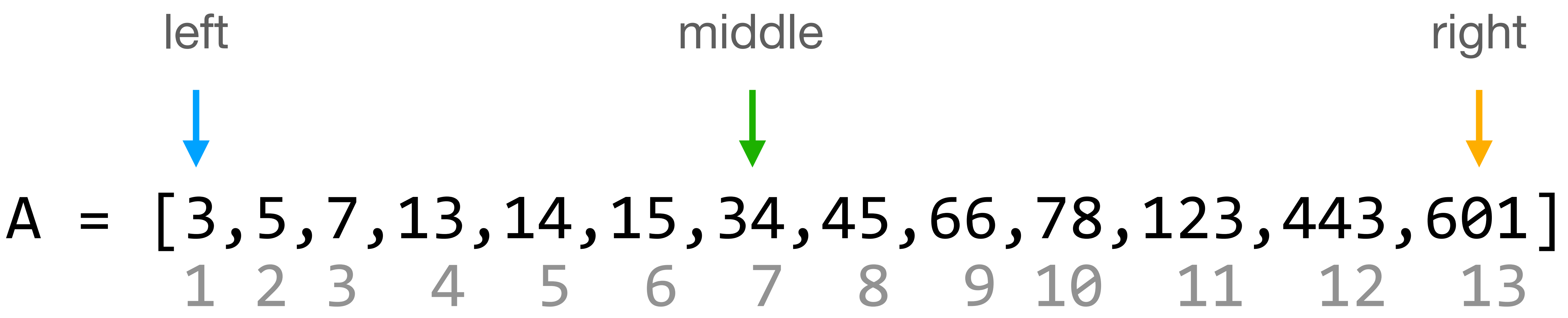
- **Our refined strategy.** Look at the element in **middle** position, `y=A[n/2]`:
  if `x = y`, then we are done;
  if `x < y`, then continue searching in the **left half** (i.e., `A[1..n/2-1]`);
  otherwise continue the search in the **right half** (i.e., `A[n/2+1..n]`).

- Example.

Binary search

- **Our refined strategy.** Look at the element in **middle** position, `y=A[n/2]`:
  if `x = y`, then we are done;
  if `x < y`, then continue searching in the **left half** (i.e., `A[1..n/2-1]`);
  otherwise continue the search in the **right half** (i.e., `A[n/2+1..n]`).

- Example.



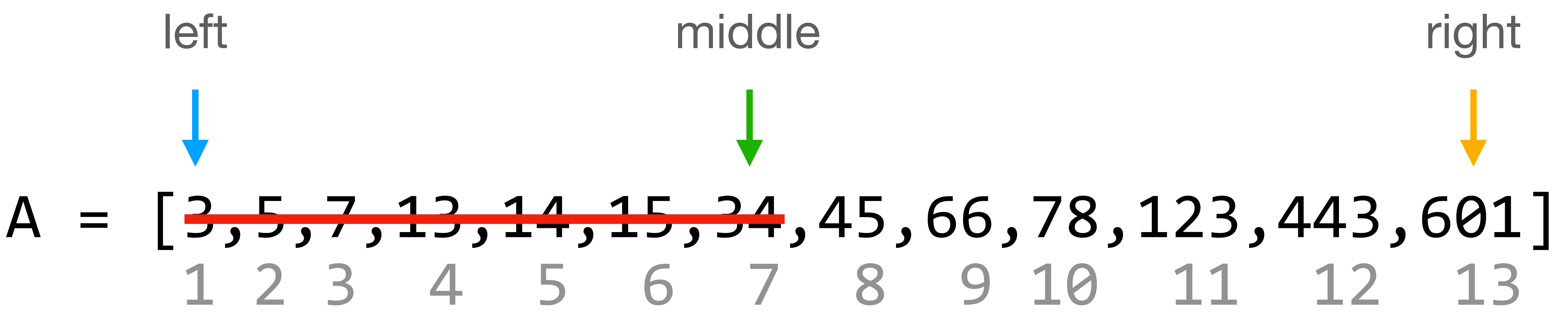
```
A = [3,5,7,13,14,15,34,45,66,78,123,443,601]
     1 2 3  4  5  6  7  8  9 10  11  12  13
```

```
x = 66
```

Binary search

- **Our refined strategy.** Look at the element in **middle** position, `y=A[n/2]`:
  if `x = y`, then we are done;
  if `x < y`, then continue searching in the **left half** (i.e., `A[1..n/2-1]`);
  otherwise continue the search in the **right half** (i.e., `A[n/2+1..n]`).

- Example.



A = [~~3,5,7,13,14,15,34~~,45,66,78,123,443,601]

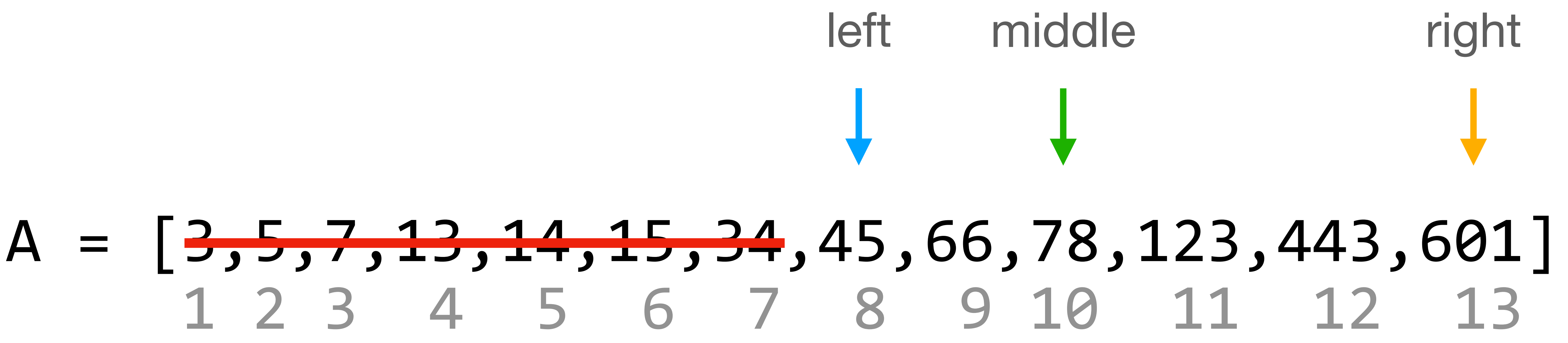
1 2 3 4 5 6 7 8 9 10 11 12 13

x = 66

Binary search

- **Our refined strategy.** Look at the element in **middle** position, `y=A[n/2]`:
  if `x = y`, then we are done;
  if `x < y`, then continue searching in the **left half** (i.e., `A[1..n/2-1]`);
  otherwise continue the search in the **right half** (i.e., `A[n/2+1..n]`).

- Example.



left middle right

A = [~~3,5,7,13,14,15,34~~,45,66,78,123,443,601]

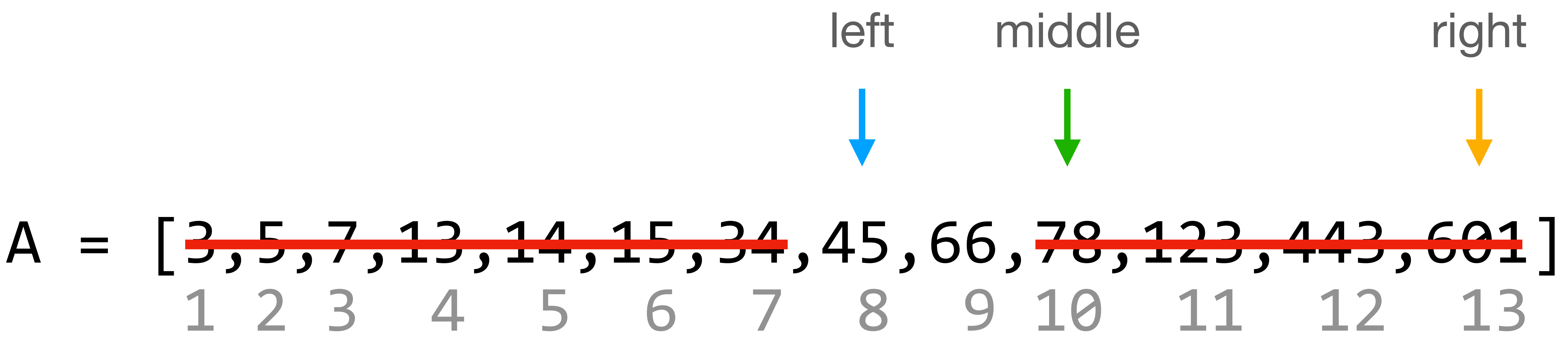
1 2 3 4 5 6 7 8 9 10 11 12 13

x = 66

Binary search

- **Our refined strategy.** Look at the element in **middle** position, `y=A[n/2]`:
  if `x = y`, then we are done;
  if `x < y`, then continue searching in the **left half** (i.e., `A[1..n/2-1]`);
  otherwise continue the search in the **right half** (i.e., `A[n/2+1..n]`).

- Example.



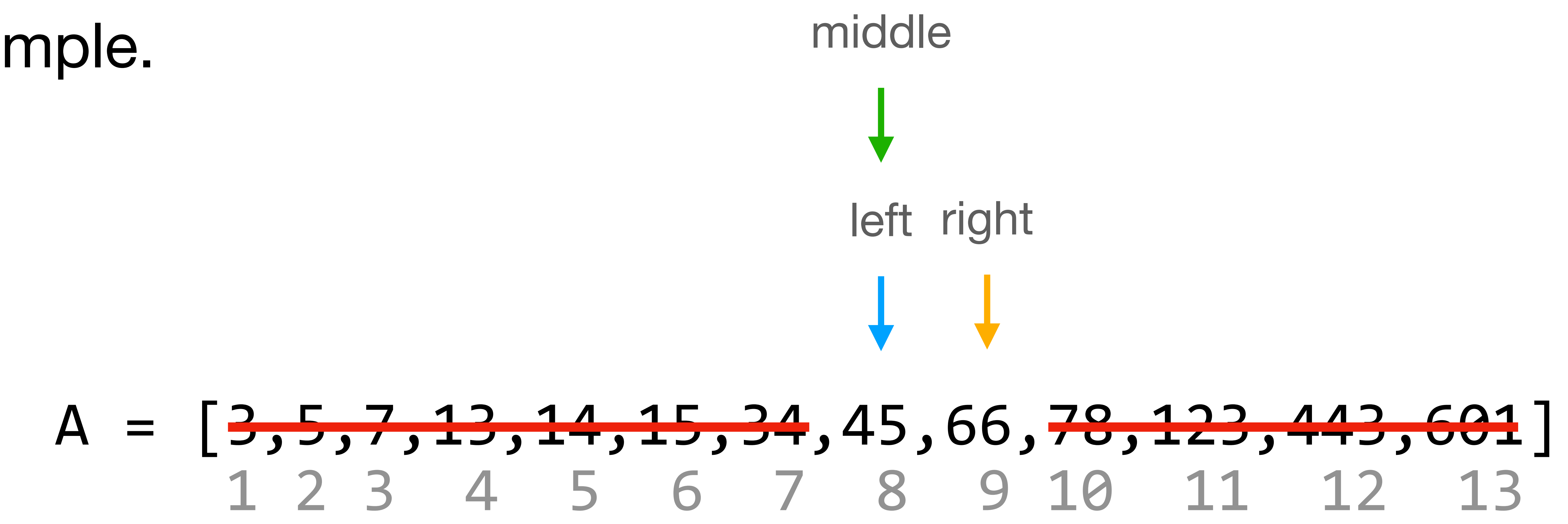
A = [~~3,5,7,13,14,15,34~~,45,66,~~78,123,443,601~~]
1 2 3 4 5 6 7 8 9 10 11 12 13

x = 66

Binary search

- **Our refined strategy.** Look at the element in **middle** position, `y=A[n/2]`:
  if `x = y`, then we are done;
  if `x < y`, then continue searching in the **left half** (i.e., `A[1..n/2-1]`);
  otherwise continue the search in the **right half** (i.e., `A[n/2+1..n]`).

- Example.



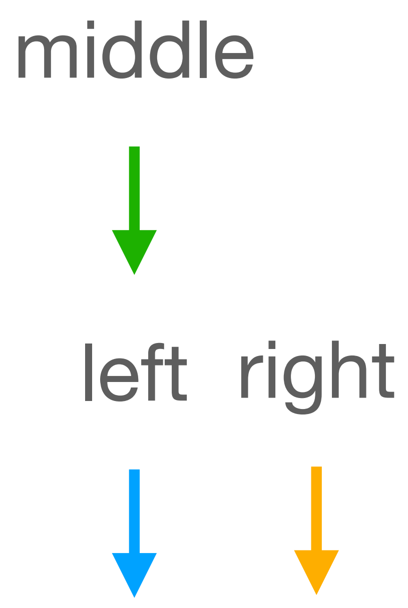
A = [~~3,5,7,13,14,15,34~~,45,66,~~78,123,443,601~~]
1 2 3 4 5 6 7 8 9 10 11 12 13

x = 66

Binary search

- **Our refined strategy.** Look at the element in **middle** position, `y=A[n/2]`:
  if `x = y`, then we are done;
  if `x < y`, then continue searching in the **left half** (i.e., `A[1..n/2-1]`);
  otherwise continue the search in the **right half** (i.e., `A[n/2+1..n]`).

- Example.



A = [~~3,5,7,13,14,15,34~~,~~45~~,66,~~78,123,443,601~~]
1 2 3 4 5 6 7 8 9 10 11 12 13

x = 66

Binary search

- **Our refined strategy.** Look at the element in **middle** position, `y=A[n/2]`:
  if `x = y`, then we are done;
  if `x < y`, then continue searching in the **left half** (i.e., `A[1..n/2-1]`);
  otherwise continue the search in the **right half** (i.e., `A[n/2+1..n]`).

- Example.

A = [~~3,5,7,13,14,15,34~~,~~45~~,66,~~78,123,443,601~~]
1 2 3 4 5 6 7 8 9 10 11 12 13

x = 66

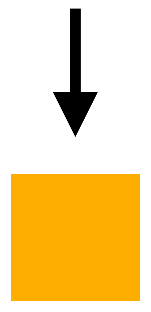
Binary search — Analysis

- **Q.** How many operations (comparisons) do we need to search an array of length n?

Binary search — Analysis

- **Q.** How many operations (comparisons) do we need to search an array of length n?

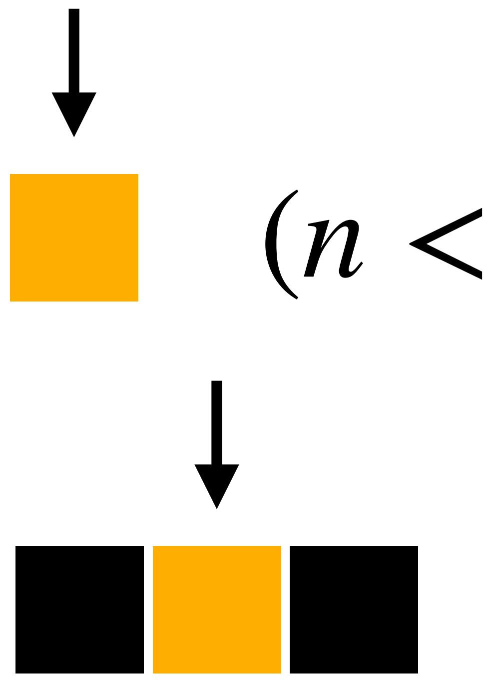
1 op:



$(n < 2)$

Binary search — Analysis

- **Q.** How many operations (comparisons) do we need to search an array of length n?

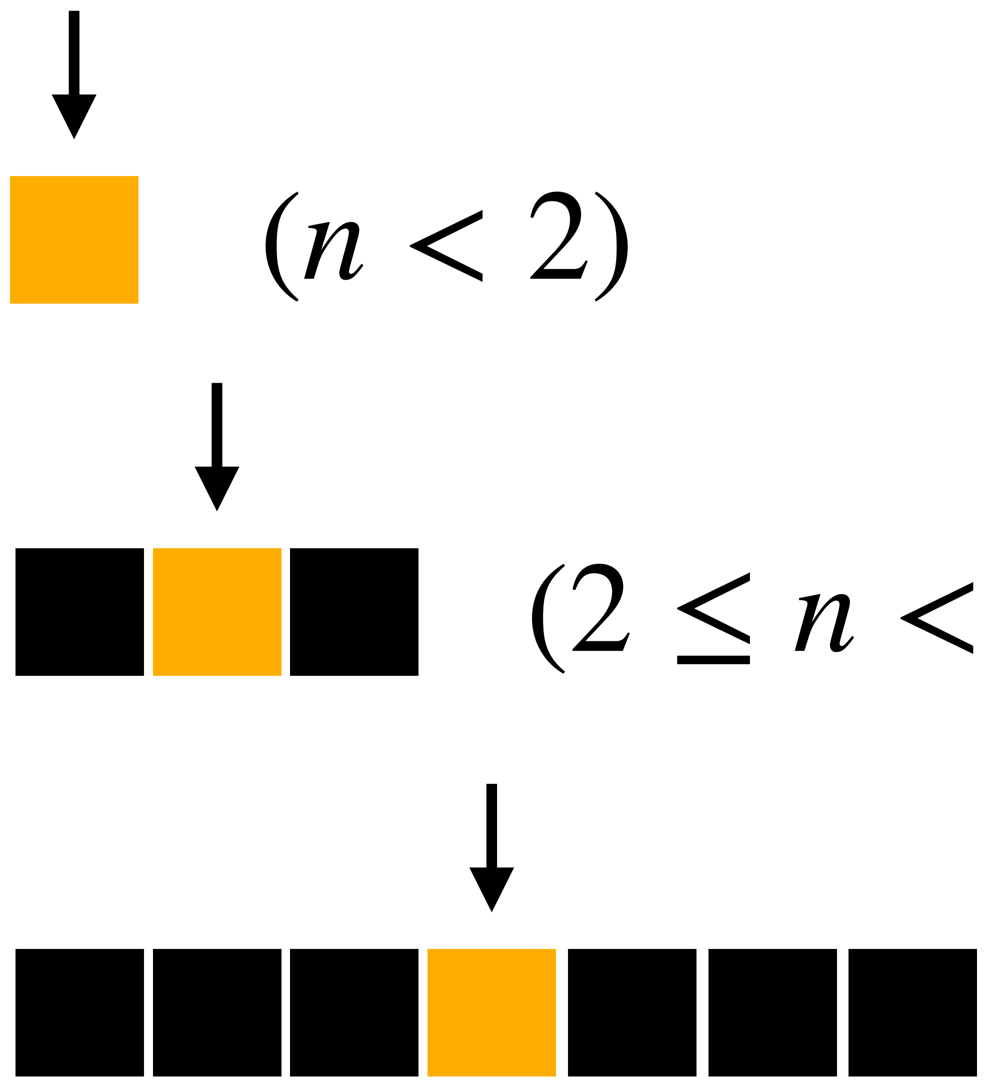


1 op: $(n < 2)$

2 ops: $(2 \leq n < 4)$

Binary search — Analysis

- **Q.** How many operations (comparisons) do we need to search an array of length n?



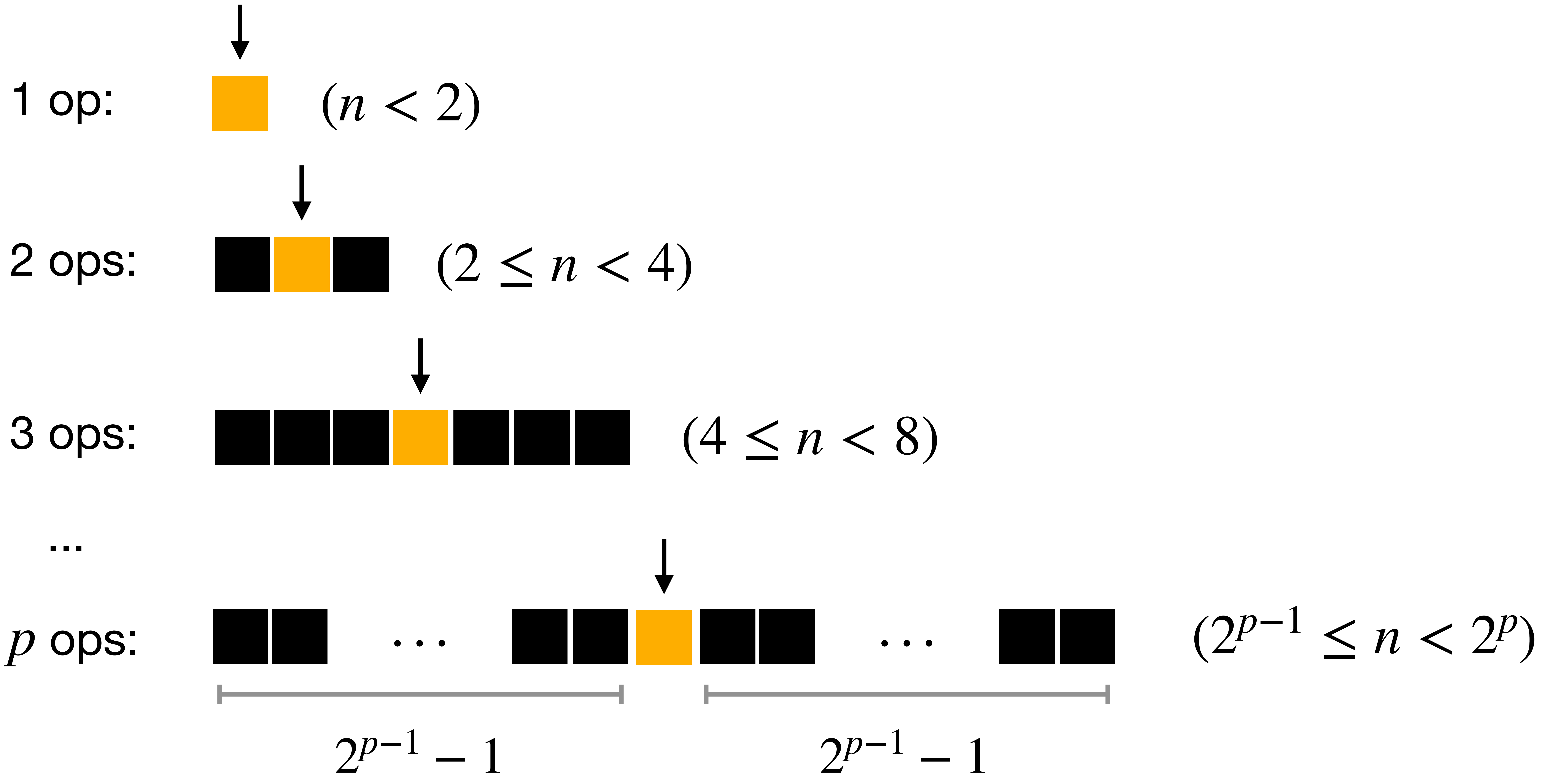
1 op: $(n < 2)$

2 ops: $(2 \leq n < 4)$

3 ops: $(4 \leq n < 8)$

Binary search — Analysis

- **Q.** How many operations (comparisons) do we need to search an array of length n?



1 op: $(n < 2)$

2 ops: $(2 \leq n < 4)$

3 ops: $(4 \leq n < 8)$

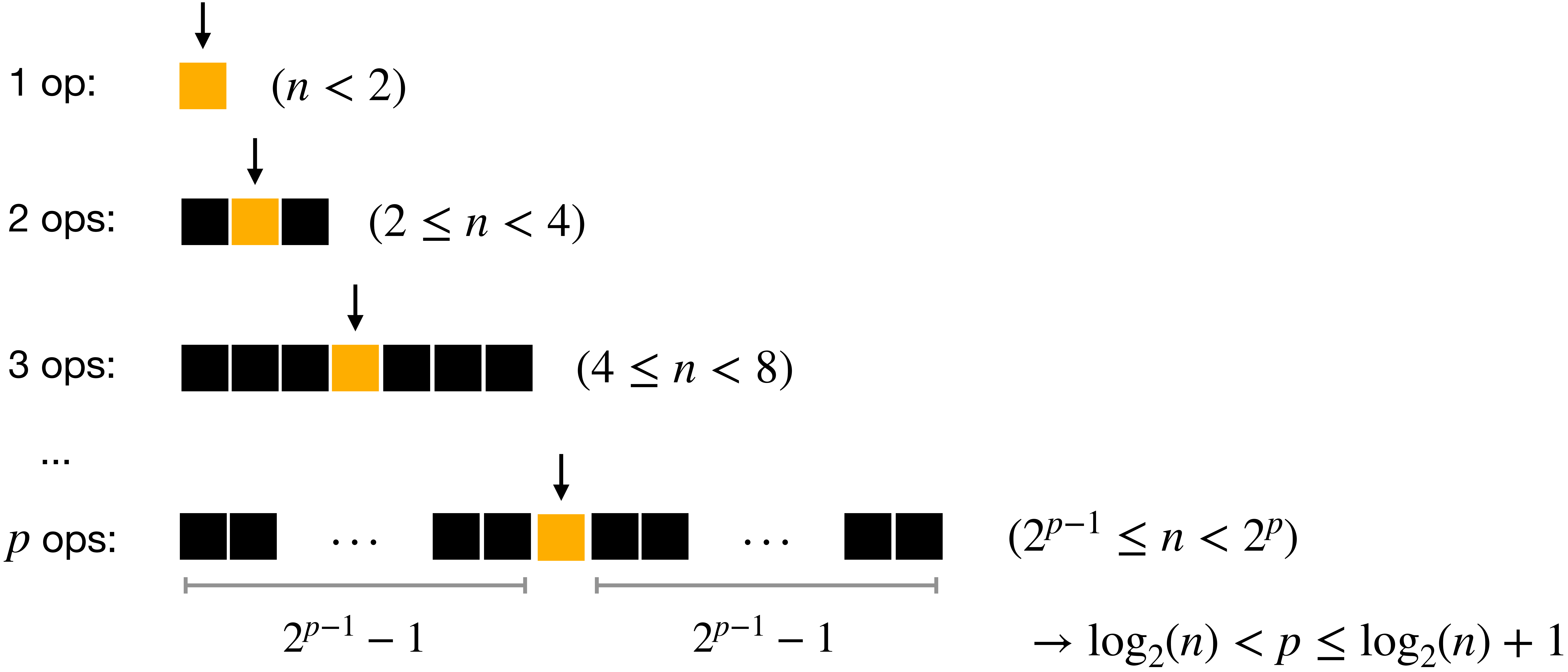
...

p ops: $(2^{p-1} \leq n < 2^p)$

$2^{p-1} - 1$ $2^{p-1} - 1$

Binary search — Analysis

- **Q.** How many operations (comparisons) do we need to search an array of length n?



1 op: $(n < 2)$

2 ops: $(2 \le n < 4)$

3 ops: $(4 \le n < 8)$

...

p ops: $\cdots$ $\cdots$ $(2^{p-1} \le n < 2^p)$

$2^{p-1} - 1$ $2^{p-1} - 1$

$\rightarrow \log_2(n) < p \le \log_2(n) + 1$

Linear search vs. binary search

	num. operations	**n = 100,000**	**n = 1,000,000**	**n = 10,000,000**
Linear search	$\sim n$	305 ms	3,400 ms	36,000 ms
Binary search	$\sim \log_2(n)$	0 ms	1 ms	3 ms

Running time to search for 10,000 integers.

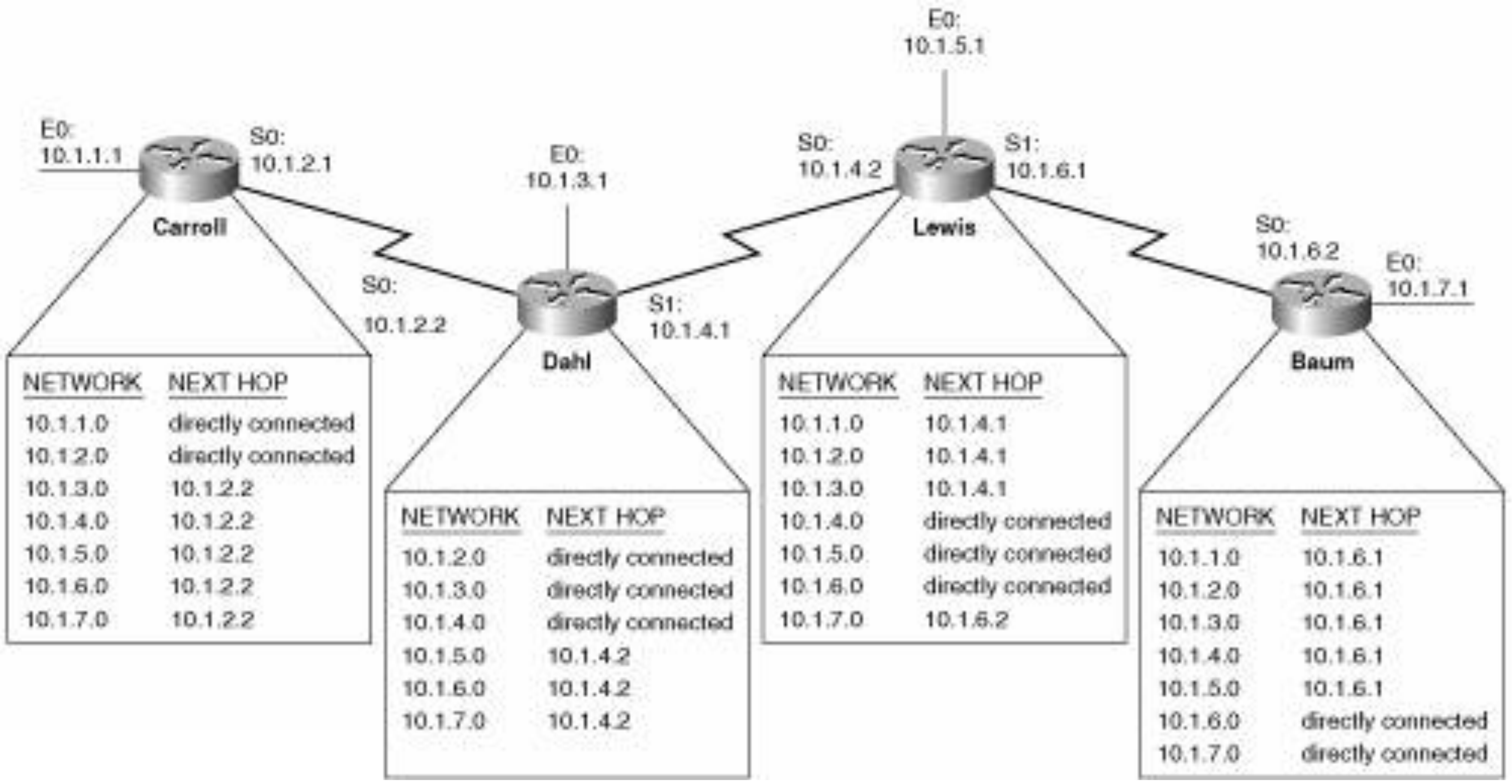
IP address lookup

- Each packet has an IP destination address which is a big integer number.

- This number is searched, at each hop, in a sorted table of destinations IP addresses.

- **Search is done via binary search.**

- Hence binary search is probably **the most run algorithm in the world!**



NETWORK	NEXT HOP
10.1.1.0	directly connected
10.1.2.0	directly connected
10.1.3.0	10.1.2.2
10.1.4.0	10.1.2.2
10.1.5.0	10.1.2.2
10.1.6.0	10.1.2.2
10.1.7.0	10.1.2.2

NETWORK	NEXT HOP
10.1.2.0	directly connected
10.1.3.0	directly connected
10.1.4.0	directly connected
10.1.5.0	10.1.4.2
10.1.6.0	10.1.4.2
10.1.7.0	10.1.4.2

NETWORK	NEXT HOP
10.1.1.0	10.1.4.1
10.1.2.0	10.1.4.1
10.1.3.0	10.1.4.1
10.1.4.0	directly connected
10.1.5.0	directly connected
10.1.6.0	directly connected
10.1.7.0	10.1.6.2

NETWORK	NEXT HOP
10.1.1.0	10.1.6.1
10.1.2.0	10.1.6.1
10.1.3.0	10.1.6.1
10.1.4.0	10.1.6.1
10.1.5.0	10.1.6.1
10.1.6.0	directly connected
10.1.7.0	directly connected

Sub-string search

- **Problem.** We are given two strings, T and P, respectively of length n and m, with usually $n \gg m$, and we are asked to find all the occurrences of P in T.
- T is also called the *text* and P is called the *pattern*.
- Example.

```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
```

Sub-string search

- **Problem.** We are given two strings, T and P, respectively of length n and m, with usually $n \gg m$, and we are asked to find all the occurrences of P in T.
- T is also called the *text* and P is called the *pattern*.
- Example.

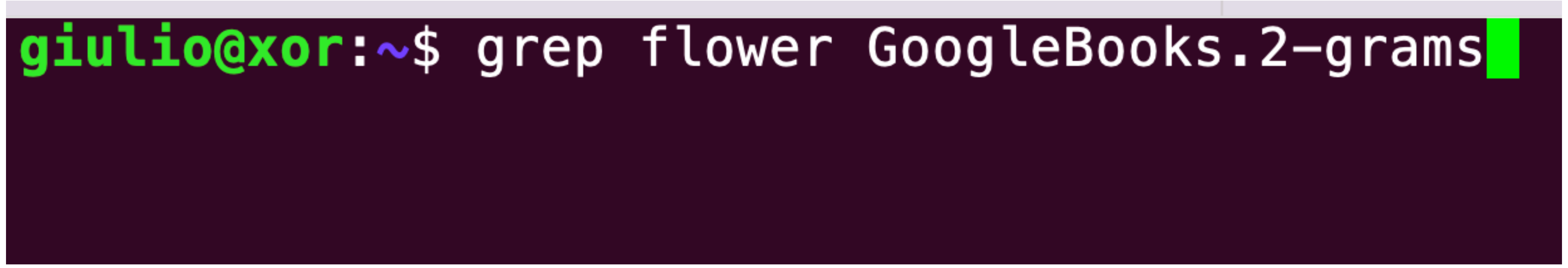
P = S I P

T = M I S S I S S I P P I L I P P I S I P

7 17

The Linux utility `grep`

```
giulio@xor:~$ grep --help
Usage: grep [OPTION]... PATTERNS [FILE]...
Search for PATTERNS in each FILE.
Example: grep -i 'hello world' menu.h main.c
PATTERNS can contain multiple patterns separated by newlines.
```



```
giulio@xor:~$ grep flower GoogleBooks.2-grams
```

search for all occurrences of "**`flower`**"
in the file "**`GoogleBooks.2-grams`**"

Brute-force algorithm

- **Idea 1.** Compare every sub-string of T of length m, $T[i\,..\,i+m-1]$, for $1 \leq i \leq n-m+1$, with P and check if they are equal.

```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
```

Brute-force algorithm

- **Idea 1.** Compare every sub-string of T of length m, $T[i\,..\,i+m-1]$, for $1 \leq i \leq n-m+1$, with P and check if they are equal.

```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
      S I P
```

Brute-force algorithm

- **Idea 1.** Compare every sub-string of T of length m, $T[i\,..\,i+m-1]$, for $1 \leq i \leq n-m+1$, with P and check if they are equal.

```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
    S I P
      S I P
```

Brute-force algorithm

- **Idea 1.** Compare every sub-string of T of length m, $T[i\,..\,i+m-1]$, for $1 \leq i \leq n-m+1$, with P and check if they are equal.

```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
    S I P
      S I P
        S I P
```

Brute-force algorithm

- **Idea 1.** Compare every sub-string of T of length m, $T[i\,..\,i+m-1]$, for $1 \leq i \leq n-m+1$, with P and check if they are equal.

```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
    S I P
      S I P
        S I P
          S I P
```

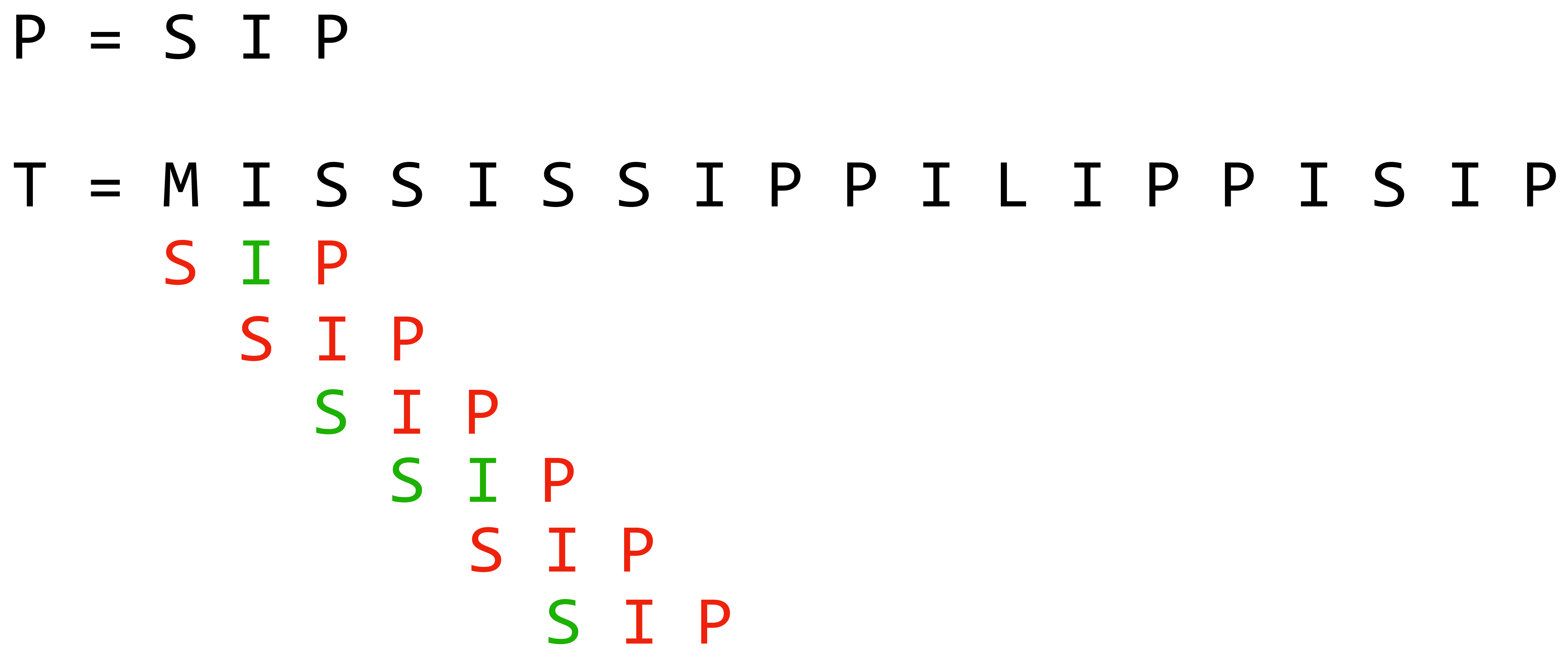
Brute-force algorithm

- **Idea 1.** Compare every sub-string of T of length m, $T[i\,..\,i+m-1]$, for $1 \leq i \leq n-m+1$, with P and check if they are equal.

```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
    S I P
      S I P
        S I P
          S I P
            S I P
```

Brute-force algorithm

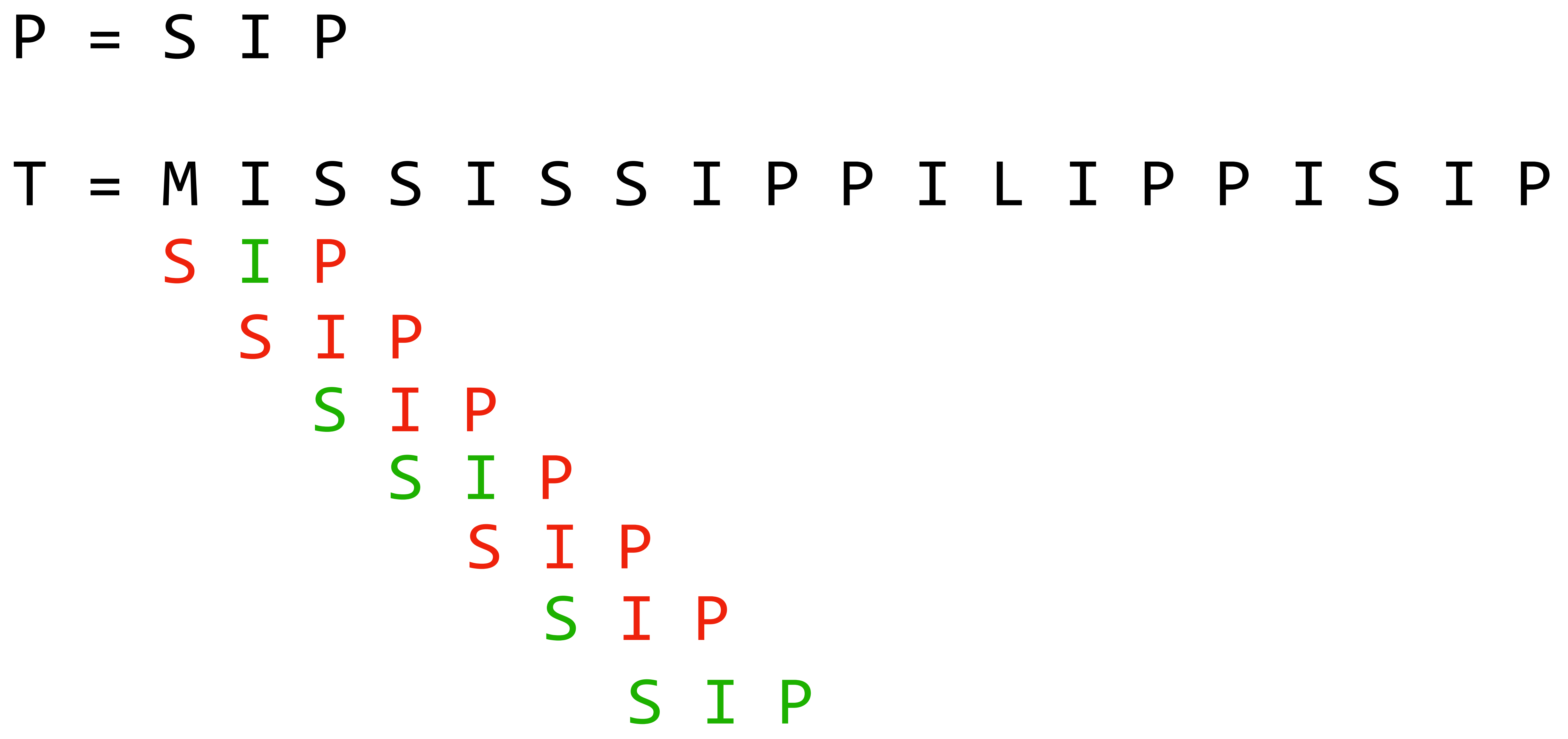
- **Idea 1.** Compare every sub-string of T of length m, $T[i \mathinner{..} i+m-1]$, for $1 \leq i \leq n-m+1$, with P and check if they are equal.



```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
    S I P
      S I P
        S I P
          S I P
            S I P
              S I P
```

Brute-force algorithm

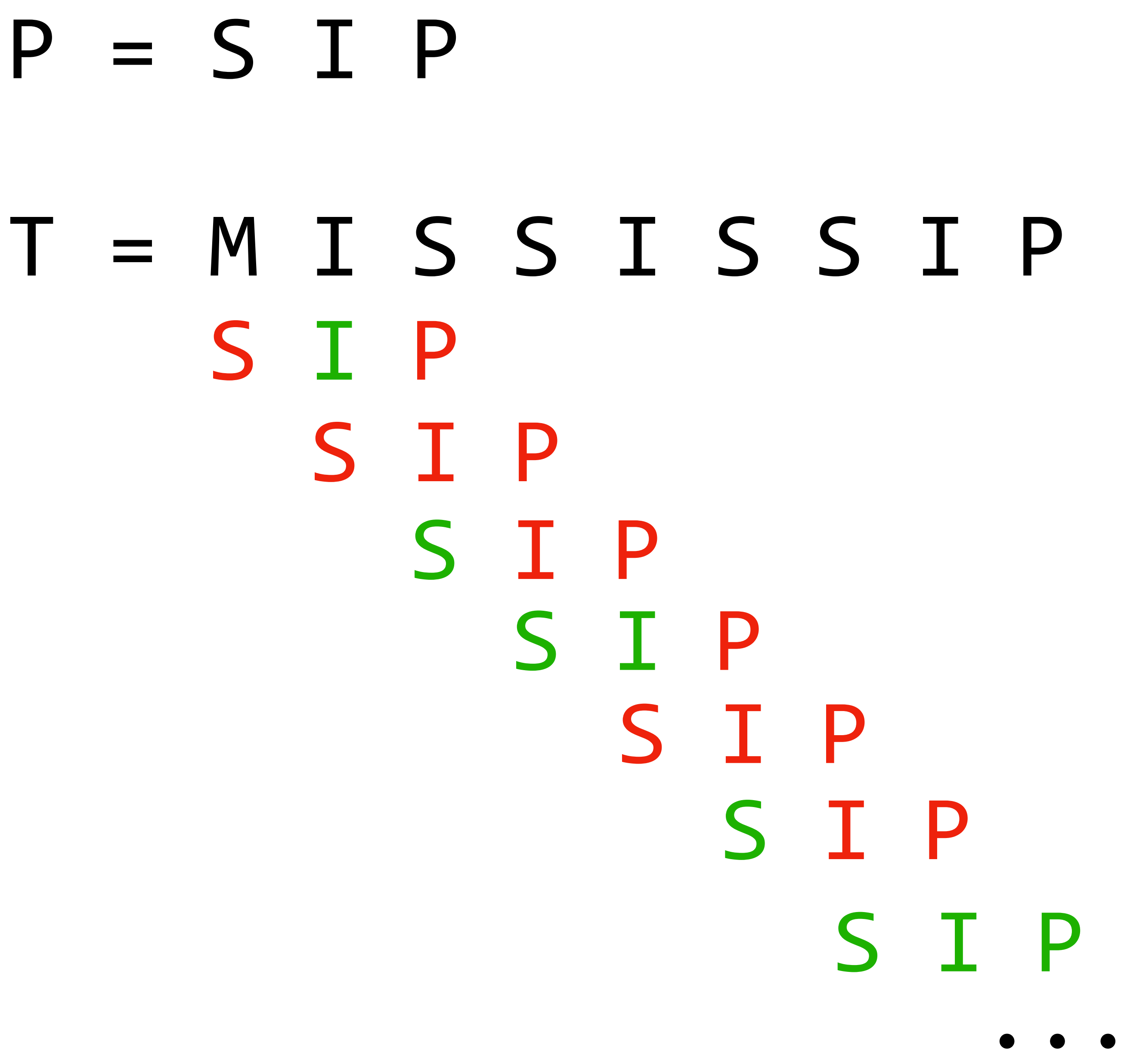
- **Idea 1.** Compare every sub-string of T of length m, $T[i \mathinner{..} i+m-1]$, for $1 \leq i \leq n-m+1$, with P and check if they are equal.



```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
    S I P
      S I P
        S I P
          S I P
            S I P
              S I P
                S I P
```

Brute-force algorithm

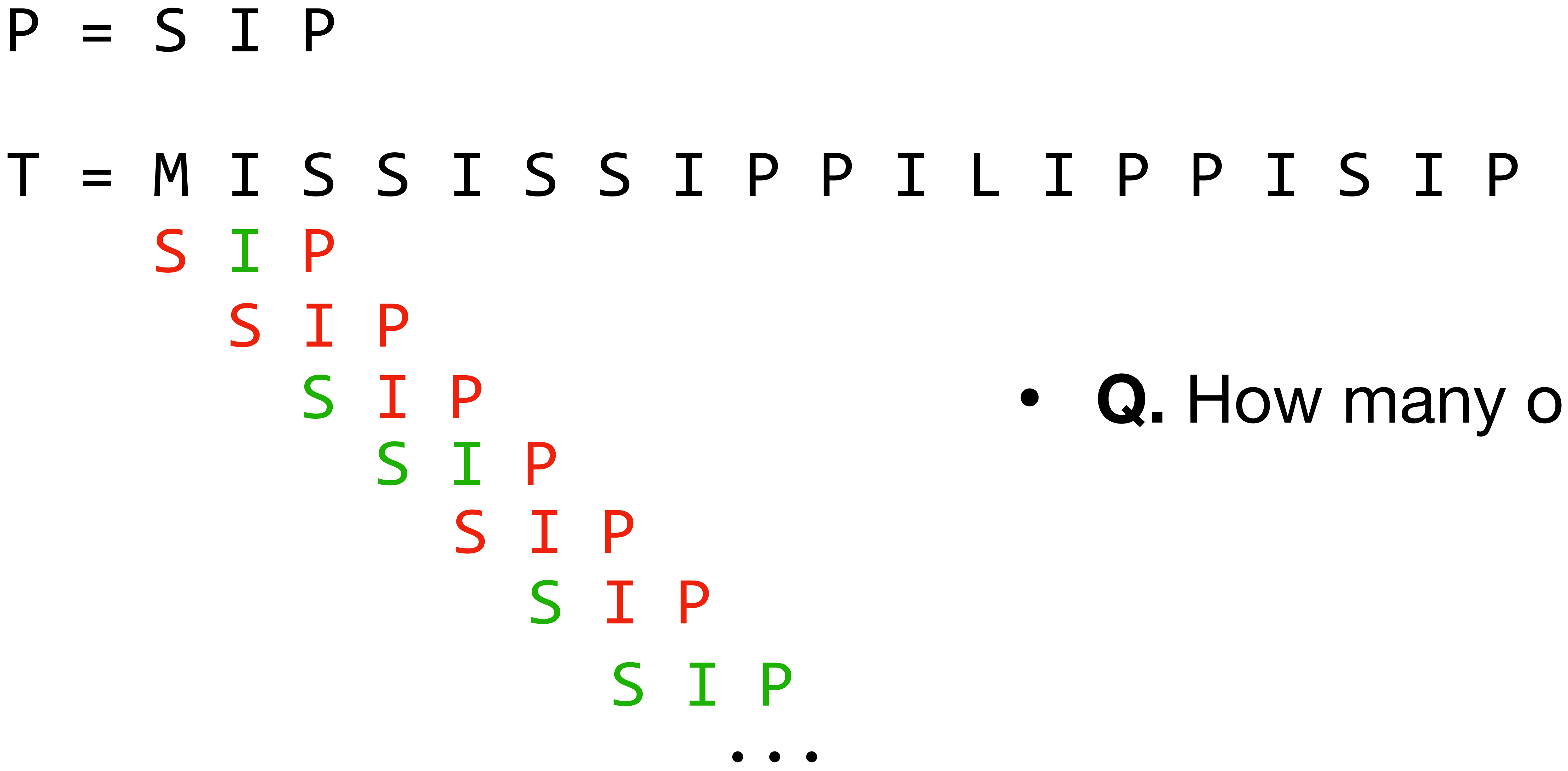
- **Idea 1.** Compare every sub-string of T of length m, $T[i\,..\,i+m-1]$, for $1 \leq i \leq n-m+1$, with P and check if they are equal.



```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
    S I P
      S I P
        S I P
          S I P
            S I P
              S I P
                S I P
                   . . .
```

Brute-force algorithm

- **Idea 1.** Compare every sub-string of T of length m, $T[i\,..\,i+m-1]$, for $1 \leq i \leq n-m+1$, with P and check if they are equal.

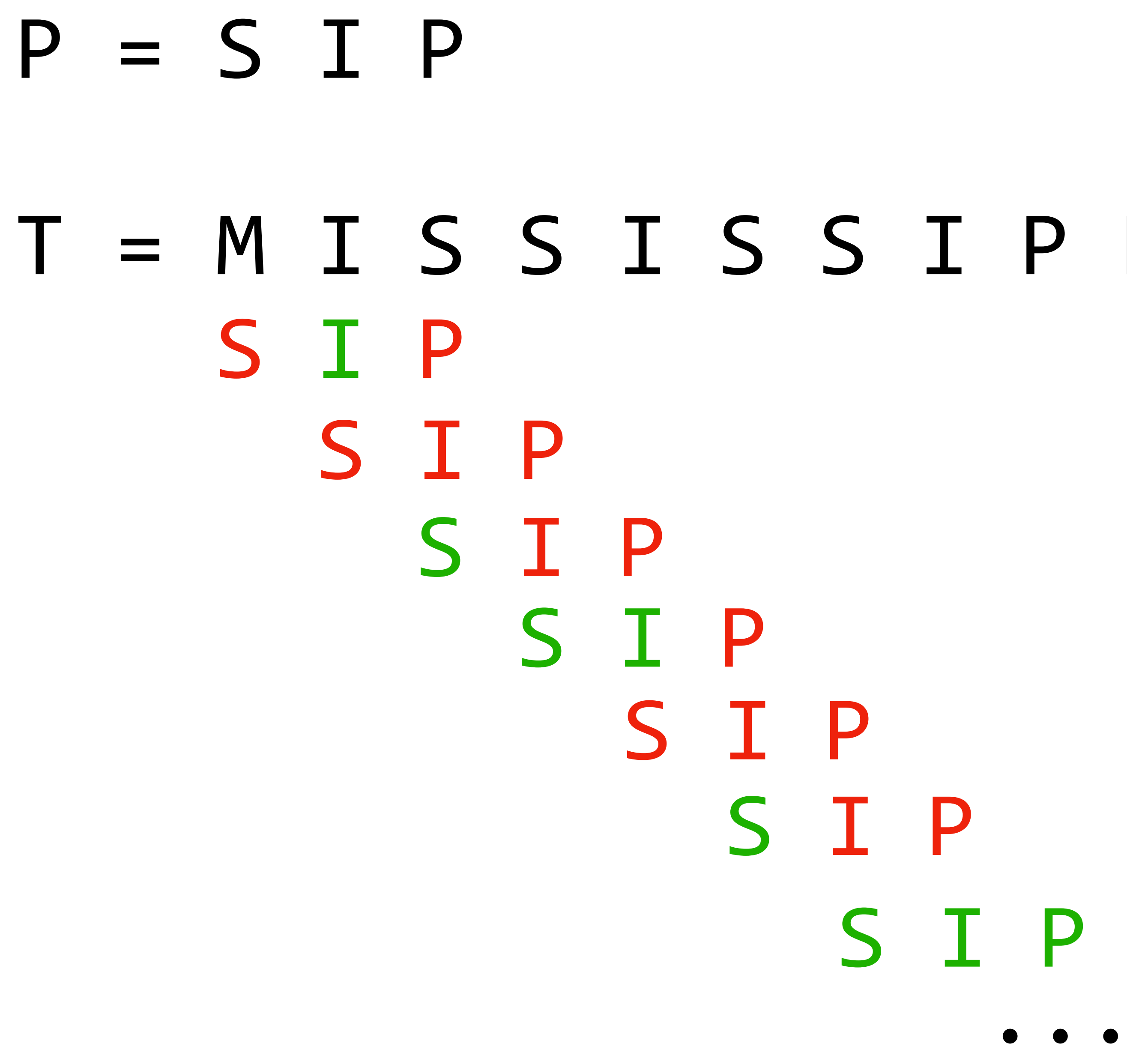


```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
    S I P
      S I P
        S I P
          S I P
            S I P
              S I P
                S I P
                  ...
```

- **Q.** How many operations?

Brute-force algorithm

- **Idea 1.** Compare every sub-string of T of length m, $T[i\,..\,i+m-1]$, for $1 \leq i \leq n-m+1$, with P and check if they are equal.



```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
    S I P
      S I P
        S I P
          S I P
            S I P
              S I P
                S I P
                  . . .
```

- **Q.** How many operations?
  - We compare two sub-strings of length m spending $\sim m$ operations.
  - We have a total of $n-m+1$ total sub-string comparisons, which is $\approx n$ when $n \gg m$.
  - Hence, a total of $\sim mn$ operations.

Brute-force algorithm

- **Summary.** Compare P to $T[i\,..\,i+m-1]$, from **left to right**, for every $1 \leq i \leq n-m+1$.
- Very easy to implement; analysis is straightforward.
- Usually sufficiently fast if m is small.

Brute-force algorithm

- **Summary.** Compare P to $T[i\,..\,i+m-1]$, from **left to right**, for every $1 \leq i \leq n-m+1$.
- Very easy to implement; analysis is straightforward.
- Usually sufficiently fast if m is small.
- Could be **slow** if m is sufficiently long.
- **Q.** How to make it faster?

Boyer-Moore algorithm

- **Intuition.** Compare **from right to left**. If the last character does **not** match, then stop comparing and jump ahead.

```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
```

Boyer-Moore algorithm

- **Intuition.** Compare **from right to left**. If the last character does **not** match, then stop comparing and jump ahead.

```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
    S I P
```

Boyer-Moore algorithm

- **Intuition.** Compare **from right to left**. If the last character does **not** match, then stop comparing and jump ahead.

```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
    S I P
        S I P
```

Boyer-Moore algorithm

- **Intuition.** Compare **from right to left**. If the last character does **not** match, then stop comparing and jump ahead.

```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
    S I P
        S I P
          S I P
```

Boyer-Moore algorithm

- **Intuition.** Compare **from right to left**. If the last character does **not** match, then stop comparing and jump ahead.

```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
    S I P
        S I P
          S I P
              S I P
```

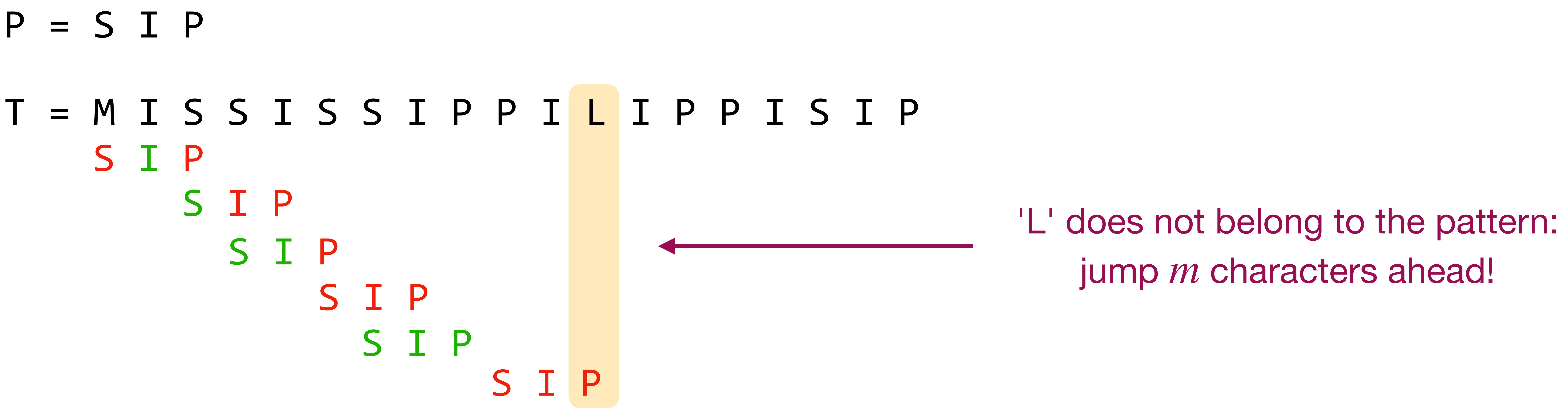
Boyer-Moore algorithm

- **Intuition.** Compare **from right to left**. If the last character does **not** match, then stop comparing and jump ahead.

```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
    S I P
        S I P
          S I P
              S I P
                S I P
```

Boyer-Moore algorithm

- **Intuition.** Compare **from right to left**. If the last character does **not** match, then stop comparing and jump ahead.

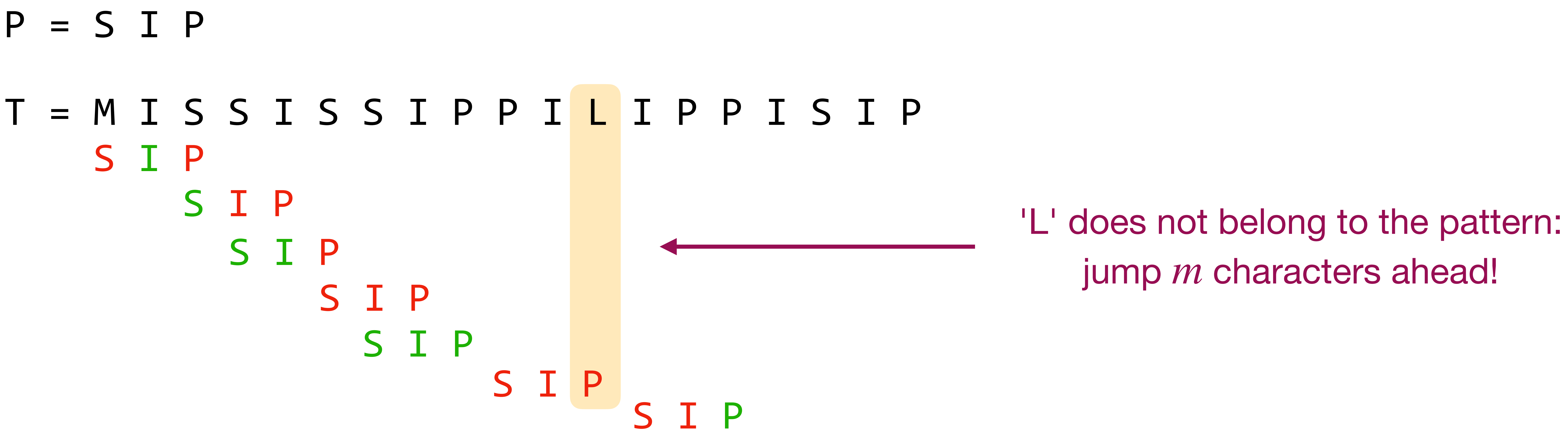


```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
    S I P
        S I P
          S I P
              S I P
                S I P
                      S I P
```

'L' does not belong to the pattern:
jump m characters ahead!

Boyer-Moore algorithm

- **Intuition.** Compare **from right to left**. If the last character does **not** match, then stop comparing and jump ahead.

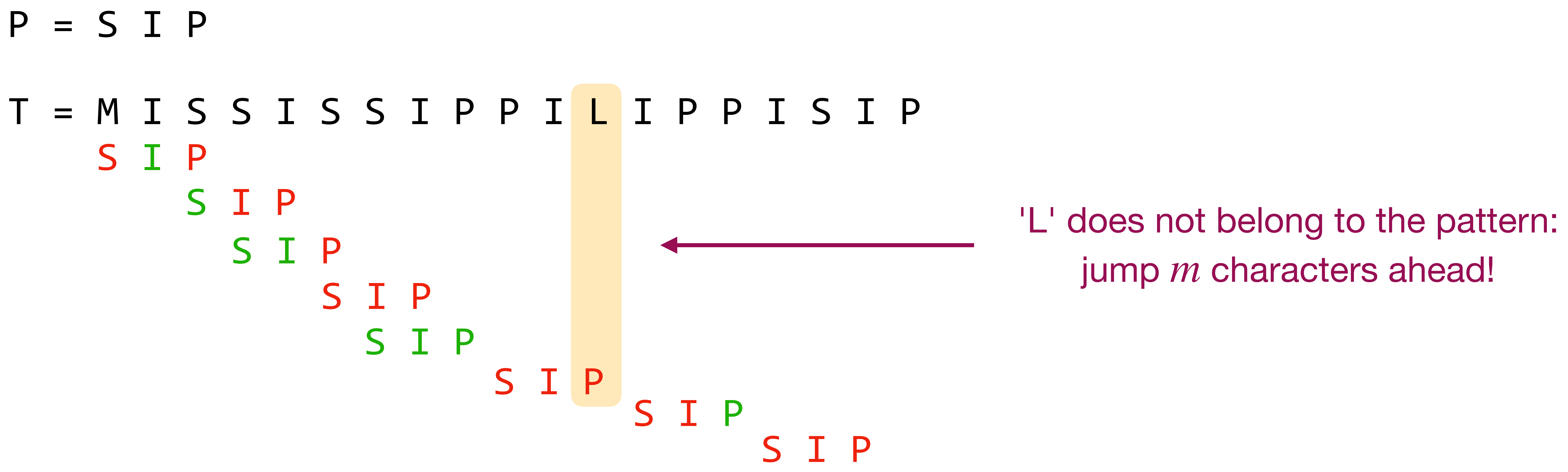


```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
    S I P
        S I P
          S I P
              S I P
                S I P
                      S I P
                            S I P
```

'L' does not belong to the pattern:
jump m characters ahead!

Boyer-Moore algorithm

- **Intuition.** Compare **from right to left**. If the last character does **not** match, then stop comparing and jump ahead.

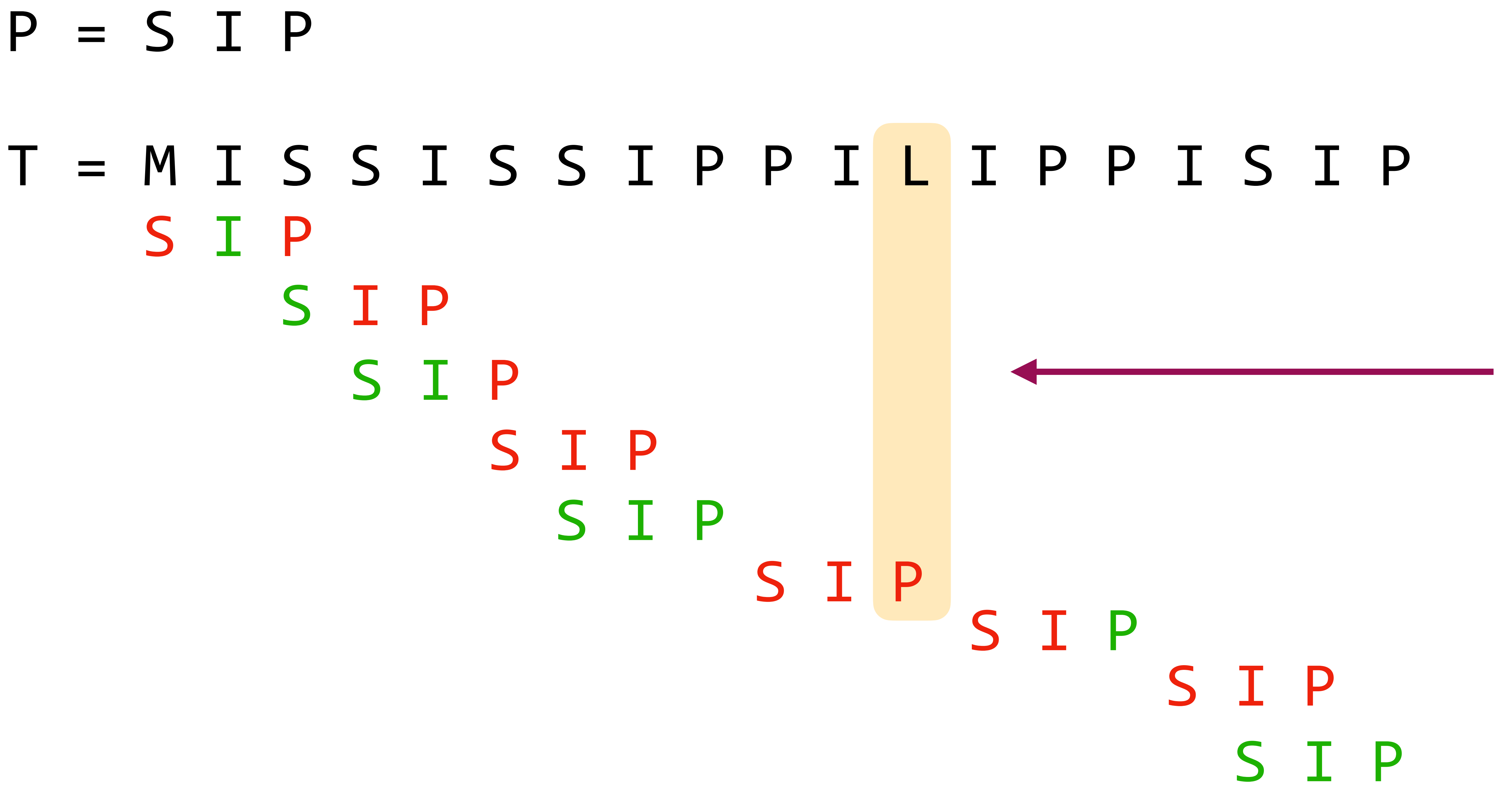


```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
    S I P
        S I P
          S I P
              S I P
                S I P
                      S I P
                            S I P
                                  S I P
```

'L' does not belong to the pattern: jump m characters ahead!

Boyer-Moore algorithm

- **Intuition.** Compare **from right to left**. If the last character does **not** match, then stop comparing and jump ahead.

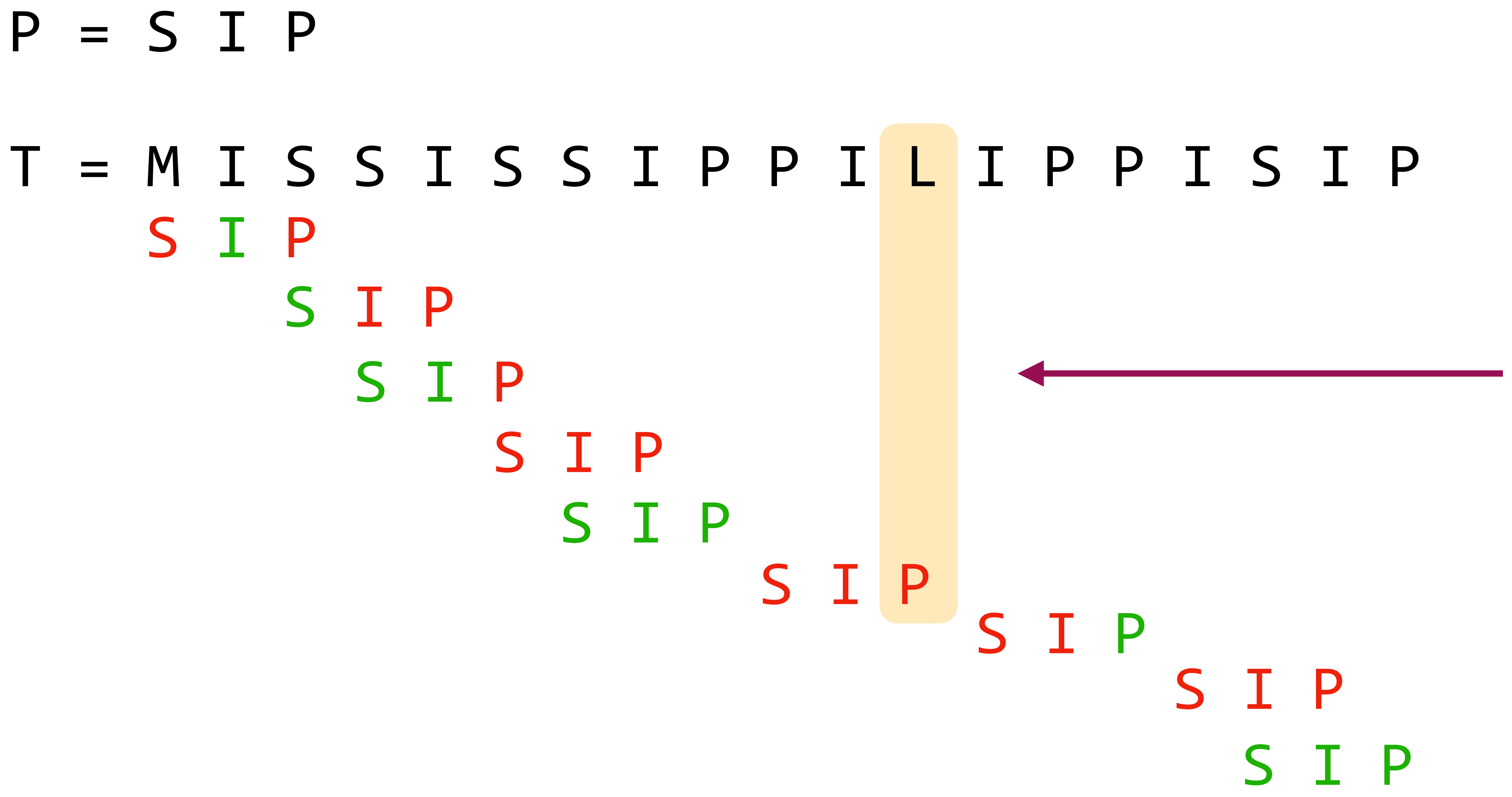


```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
    S I P
        S I P
          S I P
              S I P
                S I P
                      S I P
                            S I P
                                  S I P
                                    S I P
```

'L' does not belong to the pattern:
jump m characters ahead!

Boyer-Moore algorithm

- **Intuition.** Compare **from right to left**. If the last character does **not** match, then stop comparing and jump ahead.



```
P = S I P

T = M I S S I S S I P P I L I P P I S I P
    S I P
        S I P
          S I P
              S I P
                S I P
                      S I P
                            S I P
                                  S I P
                                    S I P
```

'L' does not belong to the pattern: jump m characters ahead!

- If the above case is frequent (as it usually **is** in practice), then we perform $\sim n/m$ operations!

Karp-Rabin algorithm

- **Idea.** Calculate a function $h(P)$ that returns an **integer number** and compare this number to $h(T[i\mathinner{.\,.}i+m-1])$. If the two numbers are equal, then we have found a match.
- Two integers can be compared with 1 operation, which is much faster than doing a string comparison ($\sim m$ operations).

Karp-Rabin algorithm

- **Idea.** Calculate a function $h(P)$ that returns an **integer number** and compare this number to $h(T[i\,..\,i+m-1])$. If the two numbers are equal, then we have found a match.

- Two integers can be compared with 1 operation, which is much faster than doing a string comparison ($\sim m$ operations).

- **Key.** Calculate the function h efficiently for every sub-string $T[i\,..\,i+m-1]$, using a constant number of operations, and not m operations.

- **Note.** Function h is called a *hash* function.

Karp-Rabin algorithm — Rolling hash function

Karp-Rabin algorithm — Rolling hash function

- How to compute h, i.e., obtain an integer number from a string?
- Remember the **ASCII** table (e.g., of size 127), mapping characters to integers.
- Each string can be treated as a "large" number in base $b = 127$.

Karp-Rabin algorithm — Rolling hash function

- How to compute h, i.e., obtain an integer number from a string?
- Remember the **ASCII** table (e.g., of size 127), mapping characters to integers.
- Each string can be treated as a "large" number in base $b = 127$.

```
     P =  S  I  P

ASCII    83 73 80
```

Karp-Rabin algorithm — Rolling hash function

- How to compute h, i.e., obtain an integer number from a string?
- Remember the **ASCII** table (e.g., of size 127), mapping characters to integers.
- Each string can be treated as a "large" number in base $b = 127$.

```
    P =  S  I  P
ASCII   83 73 80
```

$\rightarrow h(P) = 83 \times b^2 + 73 \times b + 80 = 1{,}348{,}058$ for $b = 127$.

Karp-Rabin algorithm — Rolling hash function

- **Key.** Calculate the function h efficiently for every sub-string $T[i\,..\,i+m-1]$, using a constant number of operations, and not m operations.

- **Problem.** How to calculate

  $$h(T[i+1..i+m]) = T[i+1]\cdot b^{m-1} + T[i+2]\cdot b^{m-2} + T[i+3]\cdot b^{m-3} + \cdots + T[i+m]$$

  from

  $$h(T[i\,..\,i+m-1]) = T[i]\cdot b^{m-1} + T[i+1]\cdot b^{m-2} + T[i+2]\cdot b^{m-3} + \cdots + T[i+m-1]$$

  using a **constant number of operations** ?

Karp-Rabin algorithm — Rolling hash function

- Let's consider an example.

`T = M I S S I S S I P P I L I P P I S I P`

`T[1..3] = M I S` → $h(T[1..3]) = T[1] \cdot b^2 + T[2] \cdot b + T[3]$

`T[2..4] = I S S` → $h(T[2..4]) = T[2] \cdot b^2 + T[3] \cdot b + T[4]$

Karp-Rabin algorithm — Rolling hash function

- Let's consider an example.

`T = M I S S I S S I P P I L I P P I S I P`

`T[1..3] = M I S` → $h(T[1..3]) = T[1] \cdot b^2 + T[2] \cdot b + T[3]$

`T[2..4] = I S S` → $h(T[2..4]) = T[2] \cdot b^2 + T[3] \cdot b + T[4]$

Karp-Rabin algorithm — Rolling hash function

- Let's consider an example.

`T = M I S S I S S I P P I L I P P I S I P`

subtract

`T[1..3] = M I S` → $h(T[1..3]) = T[1] \cdot b^2 + T[2] \cdot b + T[3]$

`T[2..4] = I S S` → $h(T[2..4]) = T[2] \cdot b^2 + T[3] \cdot b + T[4]$

add

Karp-Rabin algorithm — Rolling hash function

- Let's consider an example.

`T = M I S S I S S I P P I L I P P I S I P`

`T[1..3] = M I S` → $h(T[1..3]) = \overbrace{T[1] \cdot b^2}^{\text{subtract}} + T[2] \cdot b + T[3]$

`T[2..4] = I S S` → $h(T[2..4]) = T[2] \cdot b^2 + T[3] \cdot b + \underbrace{T[4]}_{\text{add}}$

- Hence, it is easy to derive that

$$h(T[i+1..i+m]) = (h(T[i\,..\,i+m-1]) - T[i] \cdot b^{m-1}) \cdot b + T[i+m].$$

Karp-Rabin algorithm — Rolling hash function

- Let's consider an example.

`T = M I S S I S S I P P I L I P P I S I P`

subtract

`T[1..3] = M I S` → $h(T[1..3]) = T[1]\cdot b^2 + T[2]\cdot b + T[3]$

`T[2..4] = I S S` → $h(T[2..4]) = T[2]\cdot b^2 + T[3]\cdot b + T[4]$

add

- Hence, it is easy to derive that

$$h(T[i+1..i+m]) = (h(T[i\,..\,i+m-1]) - T[i]\cdot b^{m-1})\cdot b + T[i+m].$$

- Just **4 operations** (not m) !

b^{m-1} can be pre-computed

Karp-Rabin algorithm

- The function h is computed using a constant number of operations for each sub-string: this leads to a **simple linear-time algorithm** $\rightarrow \ \sim n$ operations.

P = S I P $\rightarrow h(P) =$ 1348058

T = M I S S I S S I P P I L I P P I S I P

Karp-Rabin algorithm

- The function h is computed using a constant number of operations for each sub-string: this leads to a **simple linear-time algorithm** $\rightarrow\ \sim n$ operations.

```
P = S I P  → h(P) = 1348058

T = M I S S I S S I P P I L I P P I S I P
    M I S h(MIS) = 1251287
```

Karp-Rabin algorithm

- The function h is computed using a constant number of operations for each sub-string: this leads to a **simple linear-time algorithm** → $\sim n$ operations.

```
P = S I P  → h(P) = 1348058

T = M I S S I S S I P P I L I P P I S I P
    M I S h(MIS) = 1251287
      I S S h(ISS) = 1188041
```

Karp-Rabin algorithm

- The function h is computed using a constant number of operations for each sub-string: this leads to a **simple linear-time algorithm** → $\sim n$ operations.

P = S I P → $h(P) = 1348058$

T = M I S S I S S I P P I L I P P I S I P

M I S h(MIS) = 1251287

I S S h(ISS) = 1188041

S S I h(SSI) = 1349321

S I S h(SIS) = 1348061

I S S h(ISS) = 1188041

S S I h(SSI) = 1349321

S I P h(SIP) = 1348058

...

Karp-Rabin algorithm

- The function h is computed using a constant number of operations for each sub-string: this leads to a **simple linear-time algorithm** → $\sim n$ operations.

`P = S I P` $\rightarrow h(P) = 1348058$

```
T = M I S S I S S I P P I L I P P I S I P
    M I S h(MIS) = 1251287
      I S S h(ISS) = 1188041
        S S I h(SSI) = 1349321
          S I S h(SIS) = 1348061
            I S S h(ISS) = 1188041
              S S I h(SSI) = 1349321
                S I P h(SIP) = 1348058
                   ...
```

- **Caveat.** When m increases, the integers output by h increase as well. Thus we take the $h \bmod p$, where p is a *big prime* number.

Summary of sub-string search

	num. operations	**space**	**Moby Dick** (1.3 MB)	**Sherlock Holmes** (6.5 MB)
Brute force	$\sim mn$	constant	3.5 ms	15.1 ms
Boyer-Moore	$\sim n/m$	$\sim k$	0.9 ms	4.5 ms
Karp-Rabin	$\sim 4n$	constant	1.3 ms	6.3 ms

k is the alphabet size

time to search all occurrences of the pattern P = "*not only all that*"

This is not the end of the story...

- There are **many more** string search algorithms!
- So far, we have considered solutions to the sub-string search problem that do **not** use a data structure built from the text.

This is not the end of the story...

- There are **many more** string search algorithms!
- So far, we have considered solutions to the sub-string search problem that do **not** use a data structure built from the text.
- **Intuition**: if we pre-process the text T into a data structure, we can find the occurrences of the pattern P faster.
- Clear **trade-off between space and time** of the solution.
- These trade-offs are at the heart of all problems in Computer Science.

The Suffix Array data structure

- **Idea.** Build a data structure from the text T to allow faster pattern search.
- We will build a data structure known as the **suffix array** (SA) of T.

The Suffix Array data structure

- **Idea.** Build a data structure from the text T to allow faster pattern search.
- We will build a data structure known as the **suffix array** (SA) of T.
- Example.

	1	2	3	4	5	6	7	8	9	10	11	12
T =	**m**	**i**	**s**	**s**	**i**	**s**	**s**	**i**	**p**	**p**	**i**	**$**

The Suffix Array data structure

- **Idea.** Build a data structure from the text T to allow faster pattern search.
- We will build a data structure known as the **suffix array** (SA) of T.
- Example.

```
     1   2   3   4   5   6   7   8   9  10  11  12
T =  m   i   s   s   i   s   s   i   p   p   i   $  1
         i   s   s   i   s   s   i   p   p   i   $  2
             s   s   i   s   s   i   p   p   i   $  3
                 s   i   s   s   i   p   p   i   $  4
                     i   s   s   i   p   p   i   $  5
                         s   s   i   p   p   i   $  6
                             s   i   p   p   i   $  7
                                 i   p   p   i   $  8
                                     p   p   i   $  9
                                         p   i   $  10
                                             i   $  11
                                                 $  12
```

Step **1**: we take all the suffixes of T.
('`$`' is the smallest character.)

The Suffix Array data structure

- **Idea.** Build a data structure from the text T to allow faster pattern search.
- We will build a data structure known as the **suffix array** (SA) of T.
- Example.

```
       1  2  3  4  5  6  7  8  9 10 11 12
$T$ =  m  i  s  s  i  s  s  i  p  p  i  $  1
          i  s  s  i  s  s  i  p  p  i  $  2
             s  s  i  s  s  i  p  p  i  $  3
                s  i  s  s  i  p  p  i  $  4
                   i  s  s  i  p  p  i  $  5
                      s  s  i  p  p  i  $  6
                         s  i  p  p  i  $  7
                            i  p  p  i  $  8
                               p  p  i  $  9
                                  p  i  $  10
                                     i  $  11
                                        $  12
```

→

```
12  $
11  i  $
8   i  p  p  i  $
5   i  s  s  i  p  p  i  $
2   i  s  s  i  s  s  i  p  p  i  $
1   m  i  s  s  i  s  s  i  p  p  i  $
10  p  i  $
9   p  p  i  $
7   s  i  p  p  i  $
4   s  i  s  s  i  p  p  i  $
6   s  s  i  p  p  i  $
3   s  s  i  s  s  i  p  p  i  $
```

Step **1**: we take all the suffixes of T.
('$' is the smallest character.)

Step **2**: we **sort** them lexicographically.

The Suffix Array data structure

- **Idea.** Build a data structure from the text T to allow faster pattern search.

- We will build a data structure known as the **suffix array** (SA) of T.

- Example.

	1	2	3	4	5	6	7	8	9	10	11	12	
T =	m	i	s	s	i	s	s	i	p	p	i	$	1
		i	s	s	i	s	s	i	p	p	i	$	2
			s	s	i	s	s	i	p	p	i	$	3
				s	i	s	s	i	p	p	i	$	4
					i	s	s	i	p	p	i	$	5
						s	s	i	p	p	i	$	6
							s	i	p	p	i	$	7
								i	p	p	i	$	8
									p	p	i	$	9
										p	i	$	10
											i	$	11
												$	12

Step **1**: we take all the suffixes of T.
('$' is the smallest character.)

Step **2**: we **sort** them lexicographically.

→

SA												
12	$											
11	i	$										
8	i	p	p	i	$							
5	i	s	s	i	p	p	i	$				
2	i	s	s	i	s	s	i	p	p	i	$	
1	m	i	s	s	i	s	s	i	p	p	i	$
10	p	i	$									
9	p	p	i	$								
7	s	i	p	p	i	$						
4	s	i	s	s	i	p	p	i	$			
6	s	s	i	p	p	i	$					
3	s	s	i	s	s	i	p	p	i	$		

The Suffix Array data structure

- The SA of T looks like this.
- Examples.

  `SA[3] = 8`

  means that the 3-rd smallest suffix of T begins at position 8;

  `SA[6] = 1`

  means that the 6-th smallest suffix of T begins at position 1.
- Let's now see how, with SA and T, we can **search** for a pattern P.

	1	2	3	4	5	6	7	8	9	10	11	12
T =	**m**	**i**	**s**	**s**	**i**	**s**	**s**	**i**	**p**	**p**	**i**	**$**

SA = `[12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]`

1	2	3	4	5	6	7	8	9	10	11	12
$	i	i	i	i	m	p	p	s	s	s	s
	$	p	s	s	i	i	p	i	i	s	s
		p	s	s	s	$	i	p	s	i	i
		i	i	i	s		$	p	s	p	s
		$	p	s	i			i	i	p	s
			p	s	s			$	p	i	i
			i	i	s				p	$	p
			$	p	i				i		p
				p	p				$		i
				i	p						$
				$	i						
					$						

Searching with the Suffix Array

- With T and SA we can search for P by **binary search**:
  1. compare P with the string starting at $T[SA[\lfloor n/2 \rfloor]]$
  2. if **equal**, then a match if found in T at $SA[\lfloor n/2 \rfloor]$
  3. if **smaller**, recurse on $SA[1..\lfloor n/2 \rfloor - 1]$
  4. **otherwise**, recurse on $SA[\lfloor n/2 \rfloor + 1..n]$

- Example.

$P =$ `ssi`

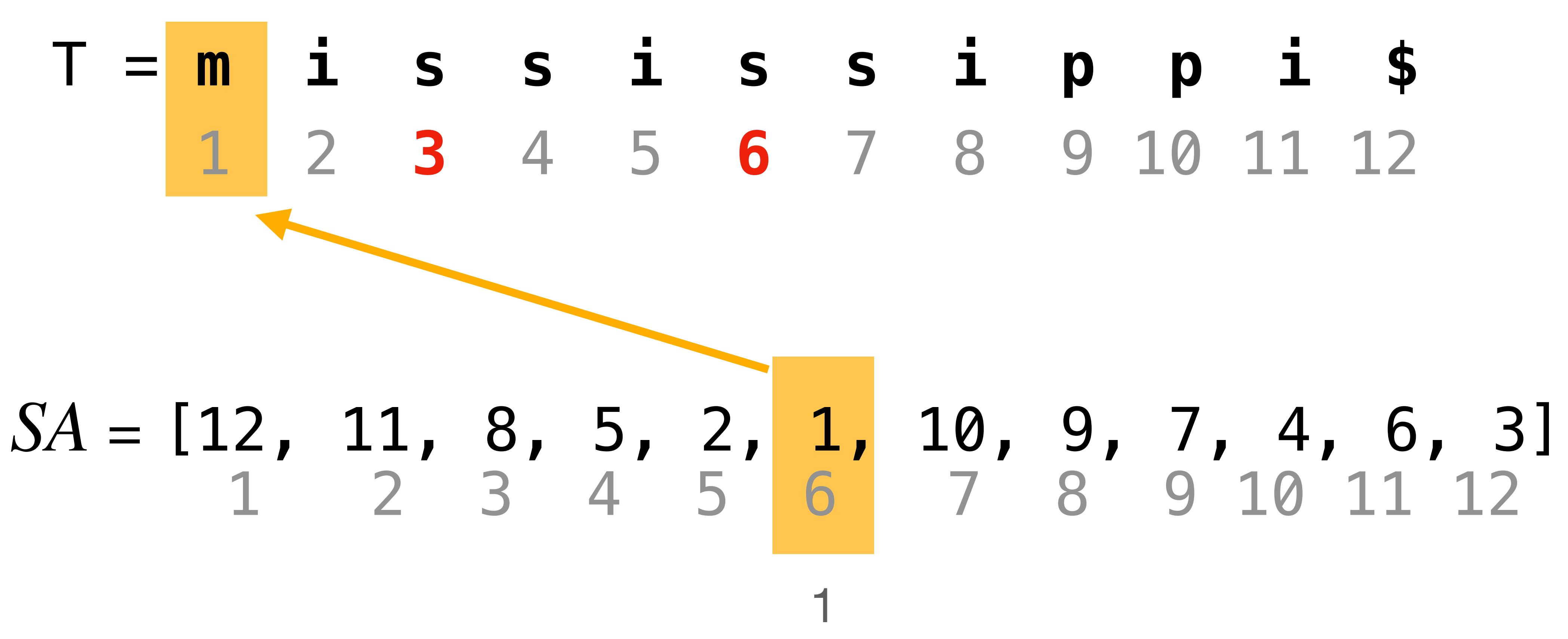
T =	**m**	**i**	**s**	**s**	**i**	**s**	**s**	**i**	**p**	**p**	**i**	**$**
	1	2	**3**	4	5	**6**	7	8	9	10	11	12

SA =	[12,	11,	8,	5,	2,	1,	10,	9,	7,	4,	6,	3]
	1	2	3	4	5	6	7	8	9	10	11	12

Searching with the Suffix Array

- With T and SA we can search for P by **binary search**:
  1. compare P with the string starting at $T[SA[\lfloor n/2 \rfloor]]$
  2. if **equal**, then a match if found in T at $SA[\lfloor n/2 \rfloor]$
  3. if **smaller**, recurse on $SA[1..\lfloor n/2 \rfloor - 1]$
  4. **otherwise**, recurse on $SA[\lfloor n/2 \rfloor + 1..n]$

- Example.



$P = \texttt{ssi}$

```
T = m i s s i s s i p p i $
    1 2 3 4 5 6 7 8 9 10 11 12
```

```
SA = [12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]
       1   2  3  4  5  6   7  8  9 10 11 12
```

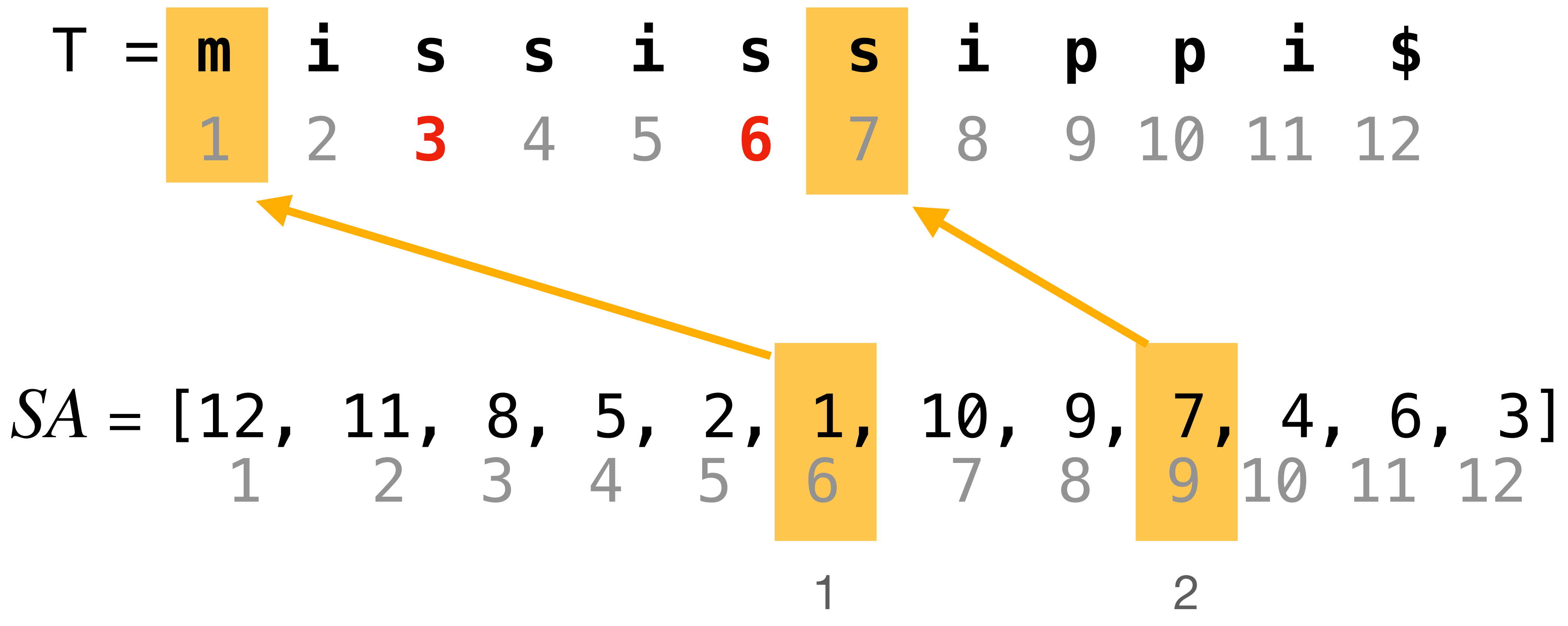
1

Searching with the Suffix Array

- With T and SA we can search for P by **binary search**:
  1. compare P with the string starting at $T[SA[\lfloor n/2 \rfloor]]$
  2. if **equal**, then a match if found in T at $SA[\lfloor n/2 \rfloor]$
  3. if **smaller**, recurse on $SA[1..\lfloor n/2 \rfloor - 1]$
  4. **otherwise**, recurse on $SA[\lfloor n/2 \rfloor + 1..n]$

- Example.

$P = \texttt{ssi}$



T = **m i s s i s s i p p i $**

1 2 3 4 5 6 7 8 9 10 11 12

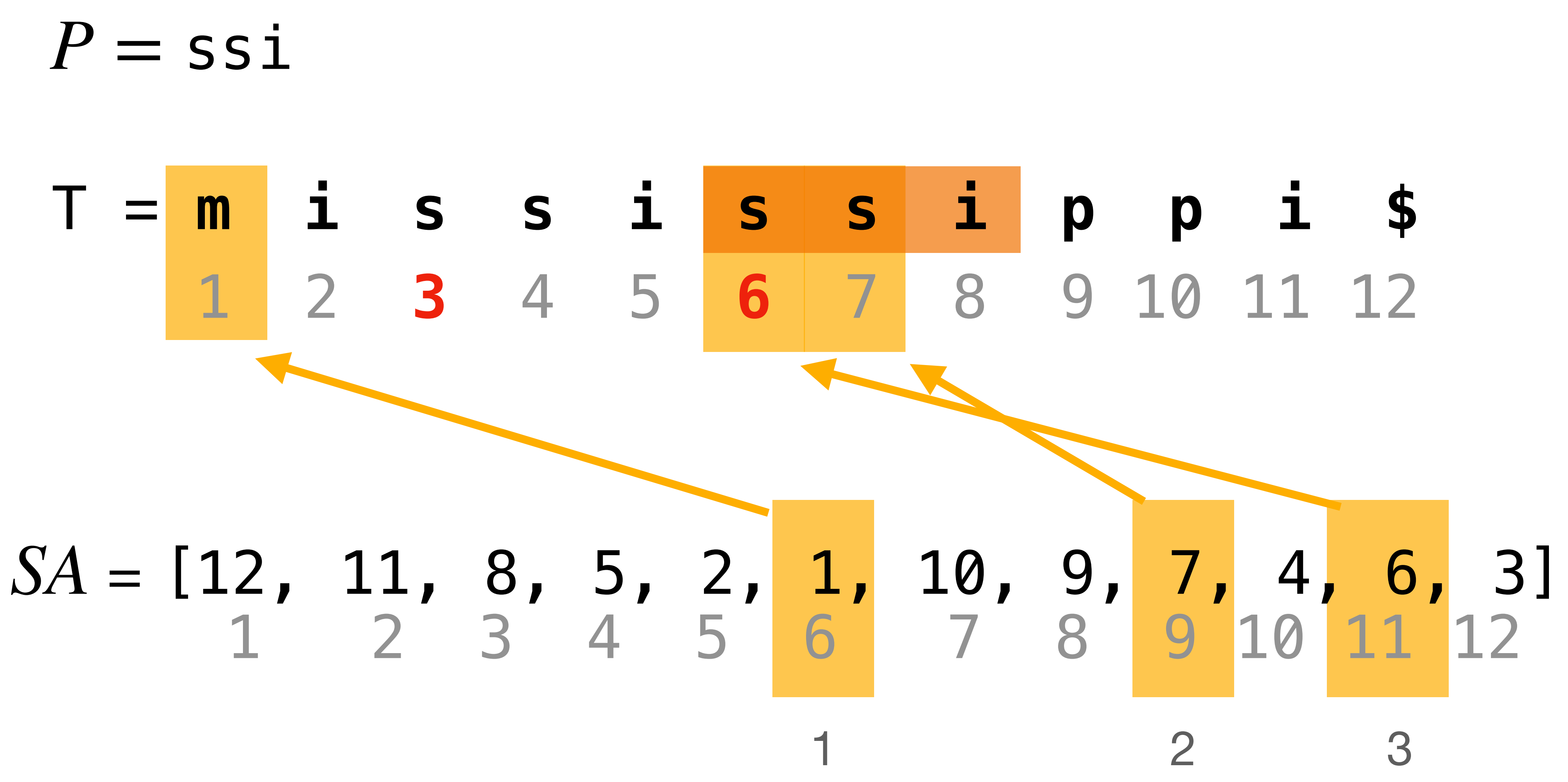
SA = [12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]

1 2 3 4 5 6 7 8 9 10 11 12

Searching with the Suffix Array

- With T and SA we can search for P by **binary search**:
  1. compare P with the string starting at $T[SA[\lfloor n/2 \rfloor]]$
  2. if **equal**, then a match if found in T at $SA[\lfloor n/2 \rfloor]$
  3. if **smaller**, recurse on $SA[1..\lfloor n/2 \rfloor - 1]$
  4. **otherwise**, recurse on $SA[\lfloor n/2 \rfloor + 1..n]$

- Example.



$P =$ `ssi`

T = m i s s i s s i p p i $

1 2 3 4 5 6 7 8 9 10 11 12

SA = [12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]

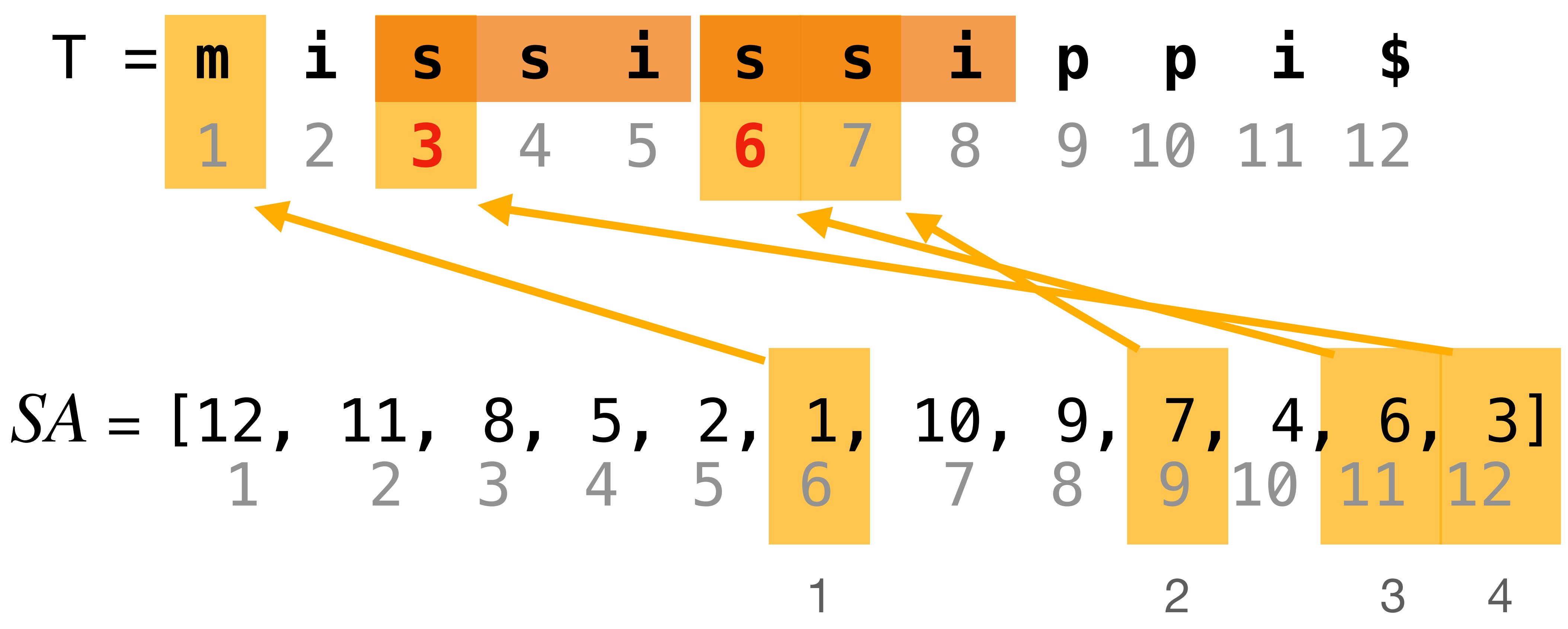
1 2 3 4 5 6 7 8 9 10 11 12

Searching with the Suffix Array

- With T and SA we can search for P by **binary search**:
  1. compare P with the string starting at $T[SA[\lfloor n/2 \rfloor]]$
  2. if **equal**, then a match if found in T at $SA[\lfloor n/2 \rfloor]$
  3. if **smaller**, recurse on $SA[1..\lfloor n/2 \rfloor - 1]$
  4. **otherwise**, recurse on $SA[\lfloor n/2 \rfloor + 1..n]$

- Example.

$P = \texttt{ssi}$



```
T = m i s s i s s i p p i $
    1 2 3 4 5 6 7 8 9 10 11 12
```

$SA =$ `[12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]`

```
  1   2  3  4  5  6   7  8  9 10 11 12
                  1         2     3  4
```

Searching with the Suffix Array — Analysis

- **Q.** Space and time of this solution?

Searching with the Suffix Array — Analysis

- **Q.** Space and time of this solution?
  - Recall that binary search takes $\sim \log_2(n)$ operations to search an array of length n.
  - Each string comparison, between P and $T[i\,..\,i+m-1]$, takes at most m operations.
  - Hence P can be searched in $\sim m\log_2(n)$ operations.

Searching with the Suffix Array — Analysis

- **Q.** Space and time of this solution?
  - Recall that binary search takes $\sim \log_2(n)$ operations to search an array of length n.
  - Each string comparison, between P and $T[i\,..\,i+m-1]$, takes at most m operations.
  - Hence P can be searched in $\sim m\log_2(n)$ operations.
  - Space?

Searching with the Suffix Array — Analysis

- **Q.** Space and time of this solution?
  - Recall that binary search takes $\sim \log_2(n)$ operations to search an array of length n.
  - Each string comparison, between P and $T[i\,..\,i+m-1]$, takes at most m operations.
  - Hence P can be searched in $\sim m\log_2(n)$ operations.
  - Space?
  - The SA is an integer array; each integer takes a value in the range $[1..n]$ and therefore requires $\lceil \log_2(n) \rceil$ bits to be represented.
  - Hence, the SA takes a total of $n\lceil \log_2(n) \rceil$ bits. (More than the text itself!)

Summary of sub-string search — Update

	num. operations	**space**	**Moby Dick** (1.3 MB)	**Sherlock Holmes** (6.5 MB)
Brute force	$\sim mn$	constant	3.5 ms	15.1 ms
Boyer-Moore	$\sim n/m$	$\sim k$	0.9 ms	4.5 ms
Karp-Rabin	$\sim 4n$	constant	1.3 ms	6.3 ms
Suffix Array	$\sim m \log_2(n)$	$n \log_2(n)$	**0.001 ms**	**0.001 ms**

k is the alphabet size

time to search all occurrences of the pattern P = "*not only all that*"

Thank you!

Questions?