ClearCode

プログラムの計算量と実行性能について

結城洋志

株式会社クリアコード

リーダブルコード演習

2022-08-04

講師紹介

結城洋志（ゆうき ひろし）
aka Piro

- ✓ 株式会社クリアコード所属
- ✓ FirefoxやThunderbirdの法人サポートに従事
  - ✓ トラブルの原因や対策を探るためソースコードを調査することが多い

アジェンダ

- ✓ プログラムの実行性能に影響する要素について
- ✓ 計算量について
- ✓ 実際に確かめてみる

プログラムは「軽い」「速い」方がいい

「軽い」「速い」とは？

- ✓ **目的が素早く達成される**
  例：データ変換プログラム
  - ✓ 同じ時間で多くの件数のデータを変換できる
  - ✓ データを1件変換する所要時間が短い

性能の指標と測定方法

- ✓ 他の方の講義で詳しく扱われるそうなので割愛

性能に影響する要素

- 外部要因
  - 通信待ち
  - ディスクアクセス待ち
    など
- 内部要因
  - **計算量**

計算量

実行される処理
の
回数の大小

例

「データを10件取得する」

1件ずつ取得する

```
SELECT * FROM table OFFSET 0 LIMIT 1;
SELECT * FROM table OFFSET 1 LIMIT 1;
...
SELECT * FROM table OFFSET 8 LIMIT 1;
SELECT * FROM table OFFSET 9 LIMIT 1;
```

→処理の実行回数は**10回**

10件まとめて取得する

```
SELECT * FROM table OFFSET 0 LIMIT 10;
```

→処理の実行回数は1回

計算量は
少ない方
がいい！

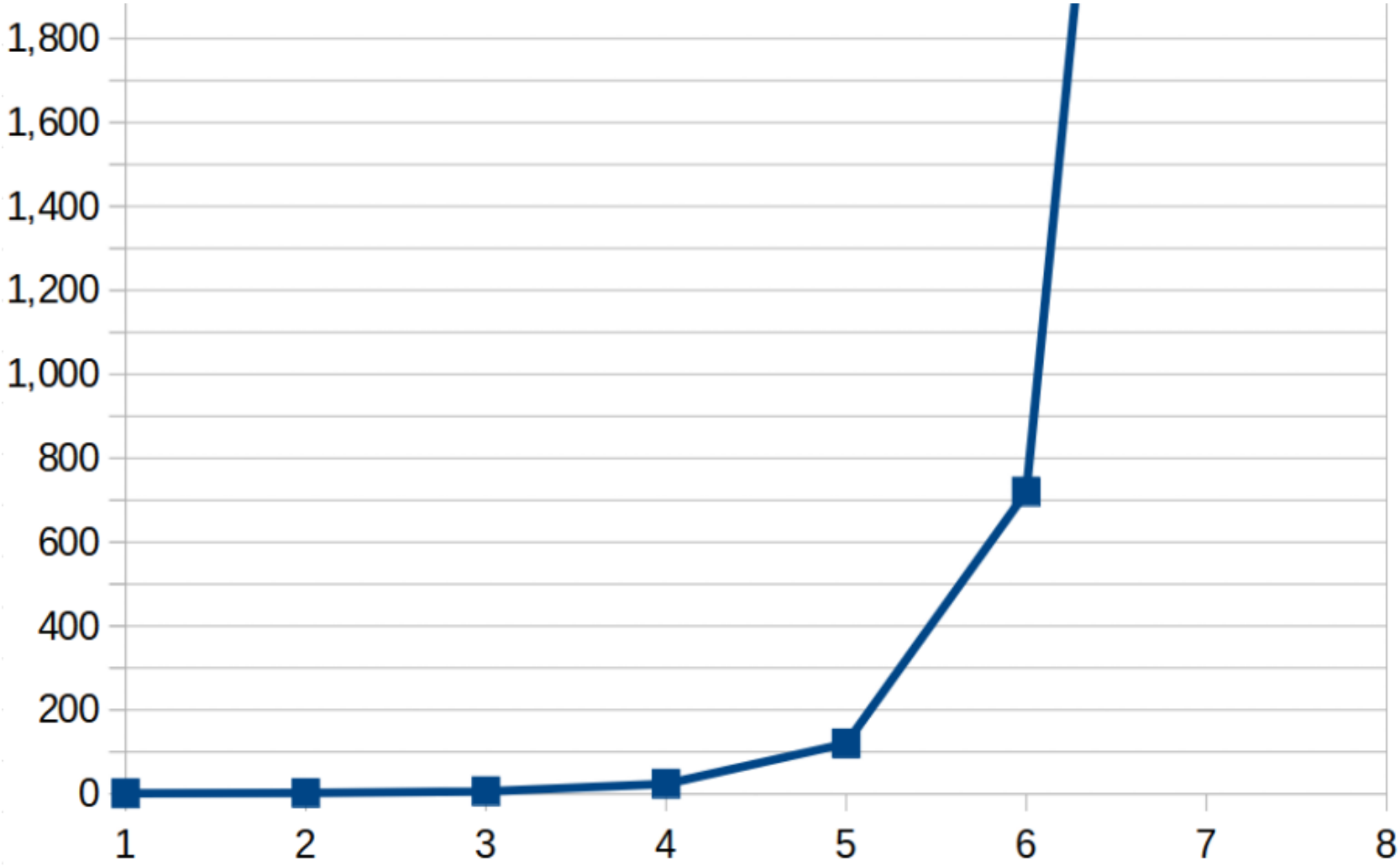
アルゴリズム(1/3)

- ✓ 計算量は**計算方法**で決まる
- ✓ 計算(処理)方法＝**アルゴリズム**
  - ✓ **より少ない計算量**で同じ結果を得られる
    ＝より優秀なアルゴリズム

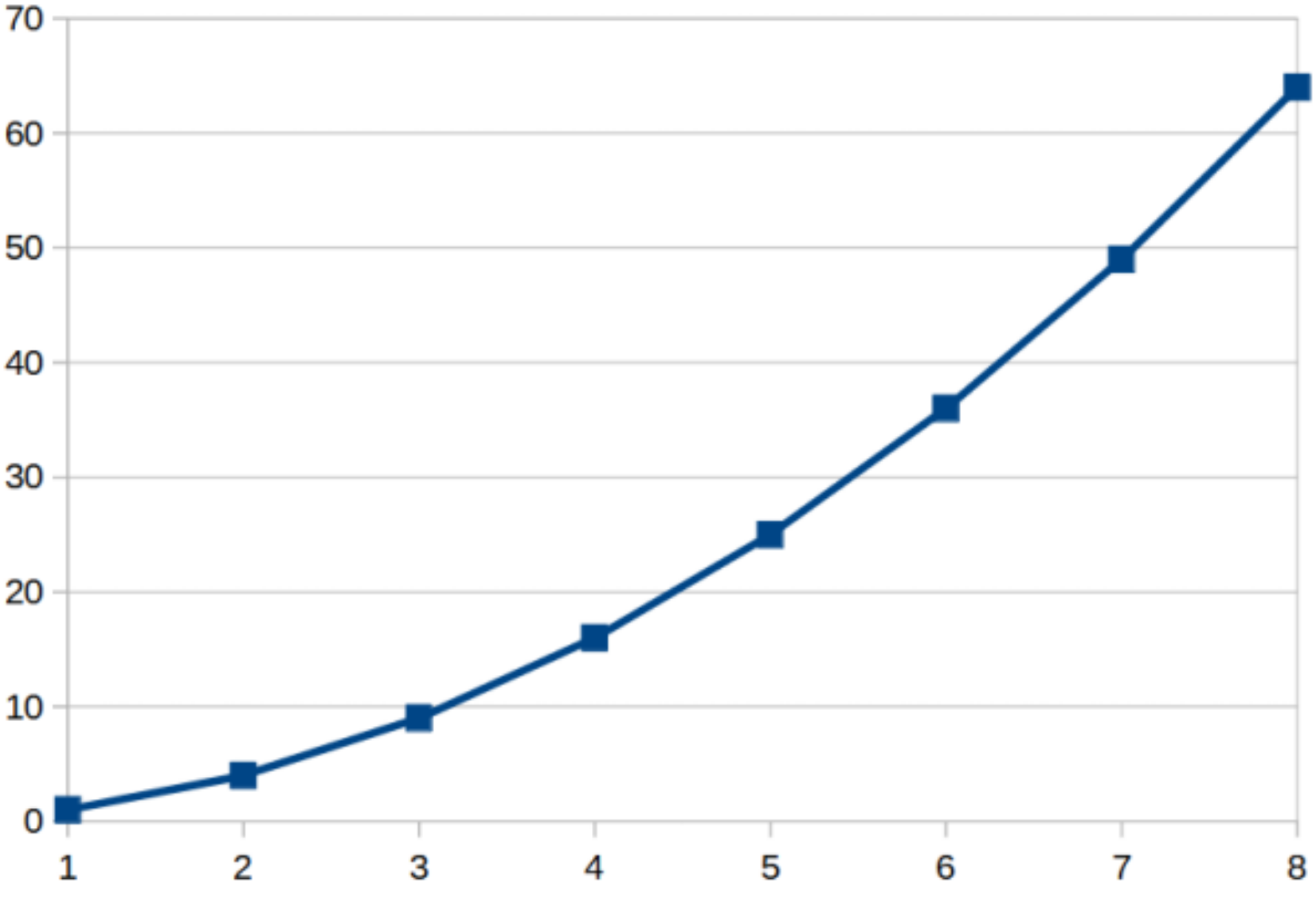
アルゴリズム(2/3)

- データ件数が増えれば
  計算量も増える（ことが多い）
  - データの件数の増加度合いに対して
    計算量の増加度合いが小さい
    ＝より優秀なアルゴリズム

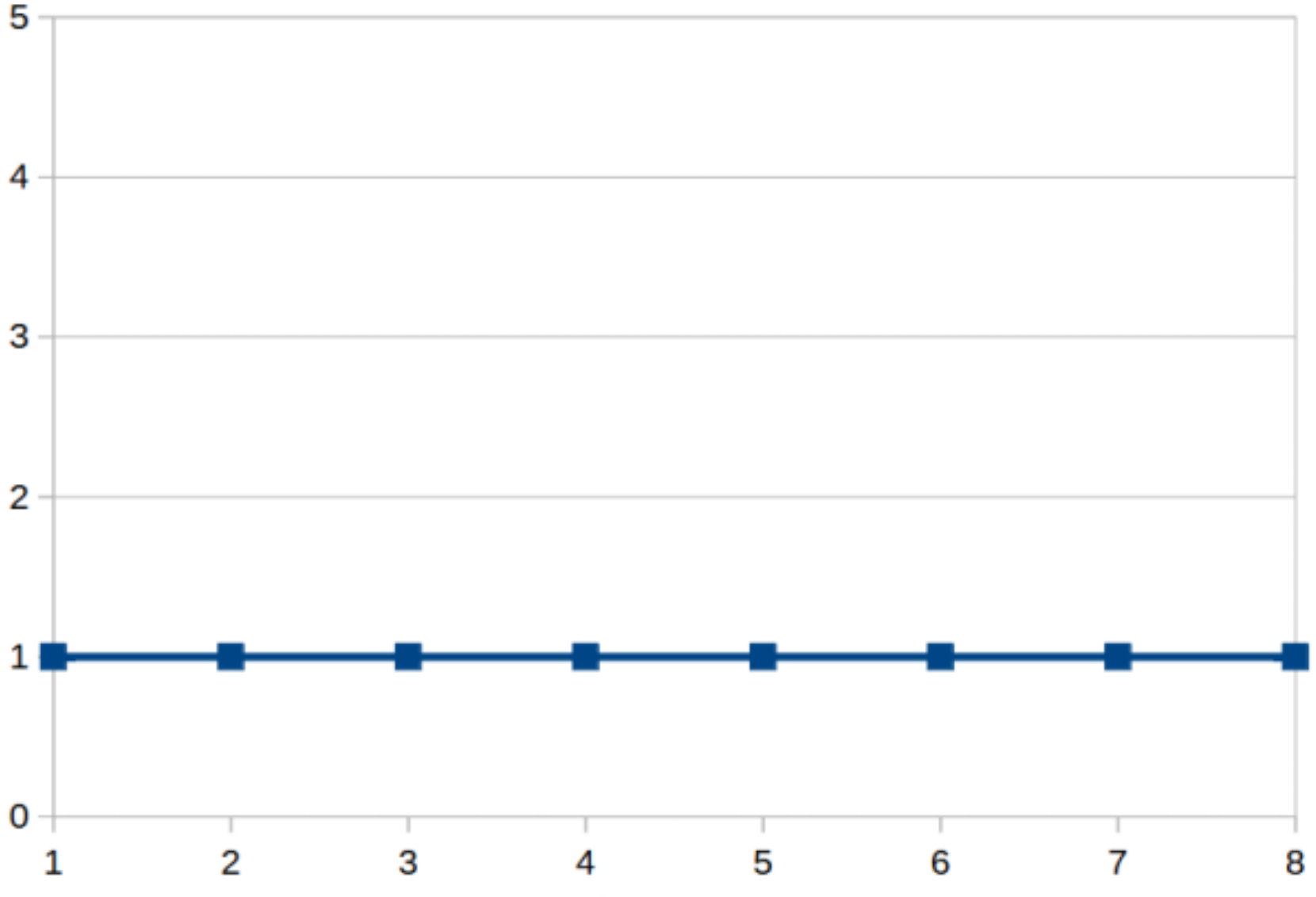
こういうのよりは



こういうのの方が嬉しい



こうだとなお嬉しい



アルゴリズム(3/3)

- ✓ 計算量は、**実行しなくても**
  アルゴリズムから見積もれる

どうやって見積もる？

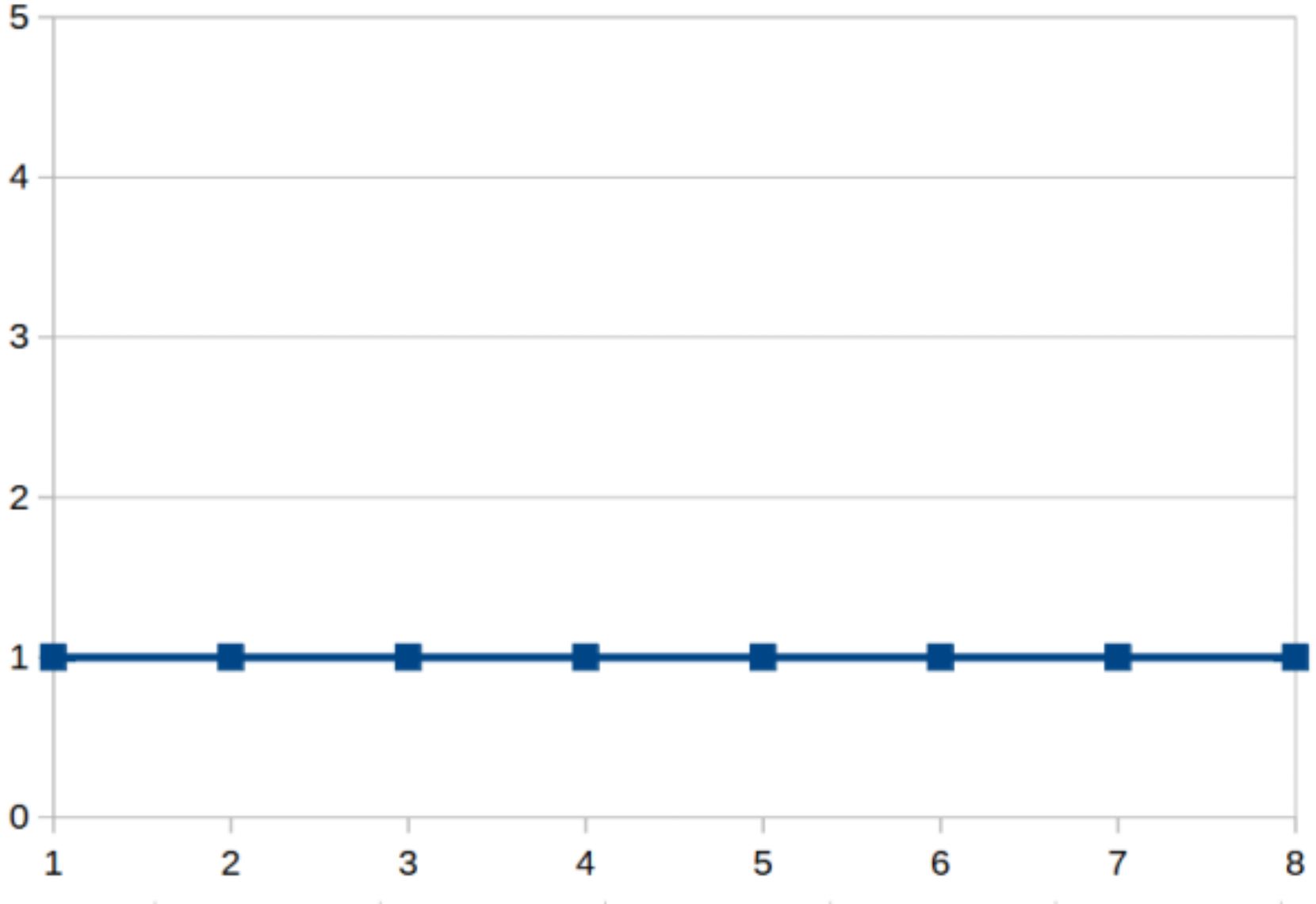
計算量の見積もり方・表し方

- O記法（オーダー）
  - O(1)
  - O(N)
  - O(N^2)
  - O(N!)
  - O(logN)

O(1)：定数時間

```
// 色の名前とカラーコードのデータ（ハッシュ構造）
const COLOR_CODE_BY_NAME = {
  blue:   0x0000FF,
  green:  0x00FF00,
  red:    0xFF0000,
  yellow: 0xFFFF00,
};
// 色の名前でカラーコードを特定する
let color = COLOR_CODE_BY_NAME['red'];
console.log(color); // => 0xFF0000
```

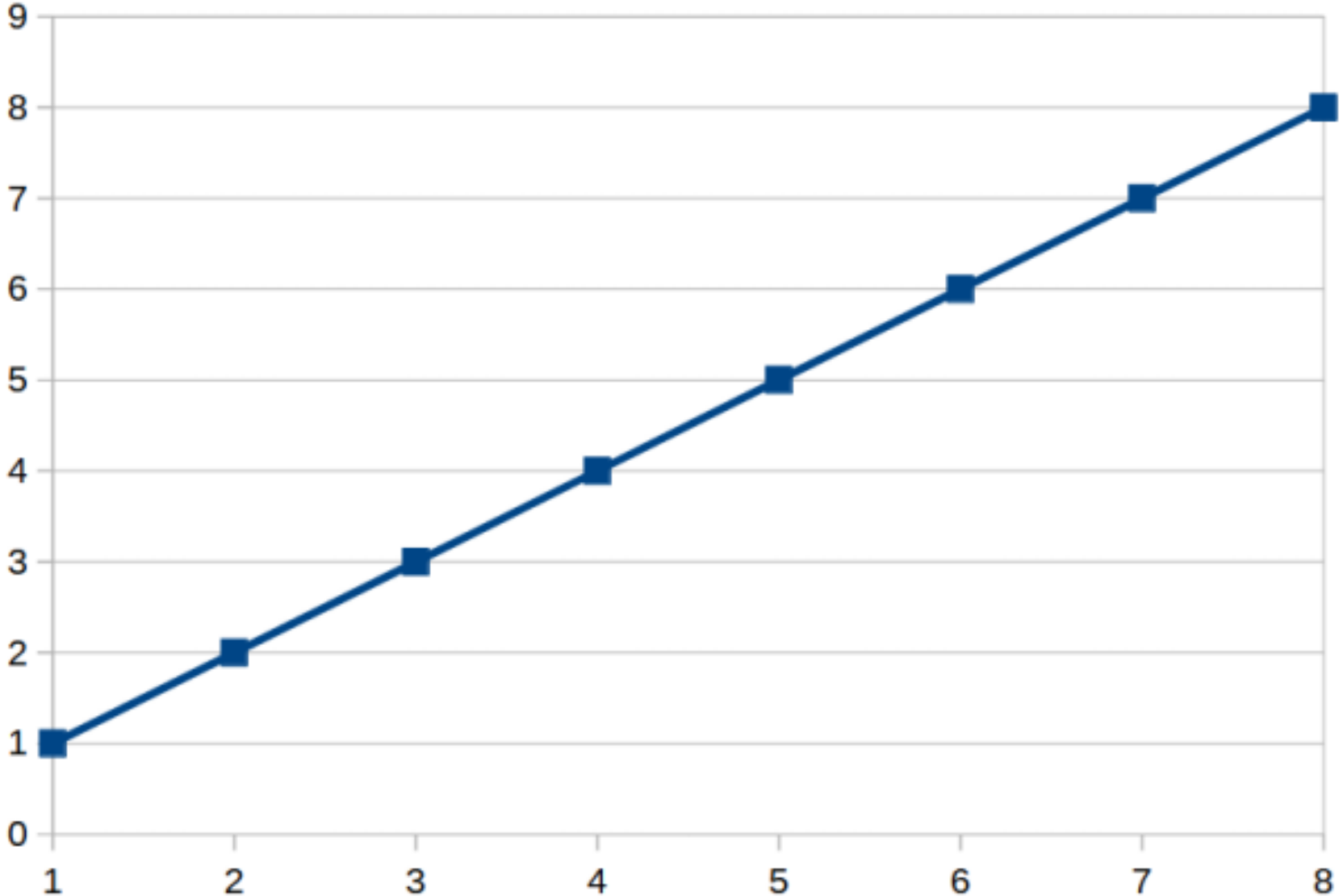
O(1)：定数時間



O(N)：線形関数

```
// 色の名前とカラーコードのデータ（配列構造）
const COLOR_CODES = [
  { name: 'blue',   code: 0x0000FF },
  { name: 'green',  code: 0x00FF00 },
  { name: 'red',    code: 0xFF0000 },
  { name: 'yellow', code: 0xFFFF00 },
];
// 色の名前でカラーコードを特定する
let color;
for (const item of COLOR_CODES) {
  if (item.name == 'red') {
    color = item.code;
    break;
  }
}
console.log(color); // => 0xFF0000
```

O(N)：線形関数



ループを使ってなくても安心できない

```
[a,b,c,d,e].indexOf(c) // => 2

[a,b,c,d,e].includes(c) // => true
```

メソッドの中身が線形関数な場合がある(1/2)

```
function indexOf() {
  const items = [a,b,c,d,e];
  for (let i = 0; i < items.length; i++) {
    if (item == c) {
      return i; // => 2
    }
  }
  return -1;
}
```

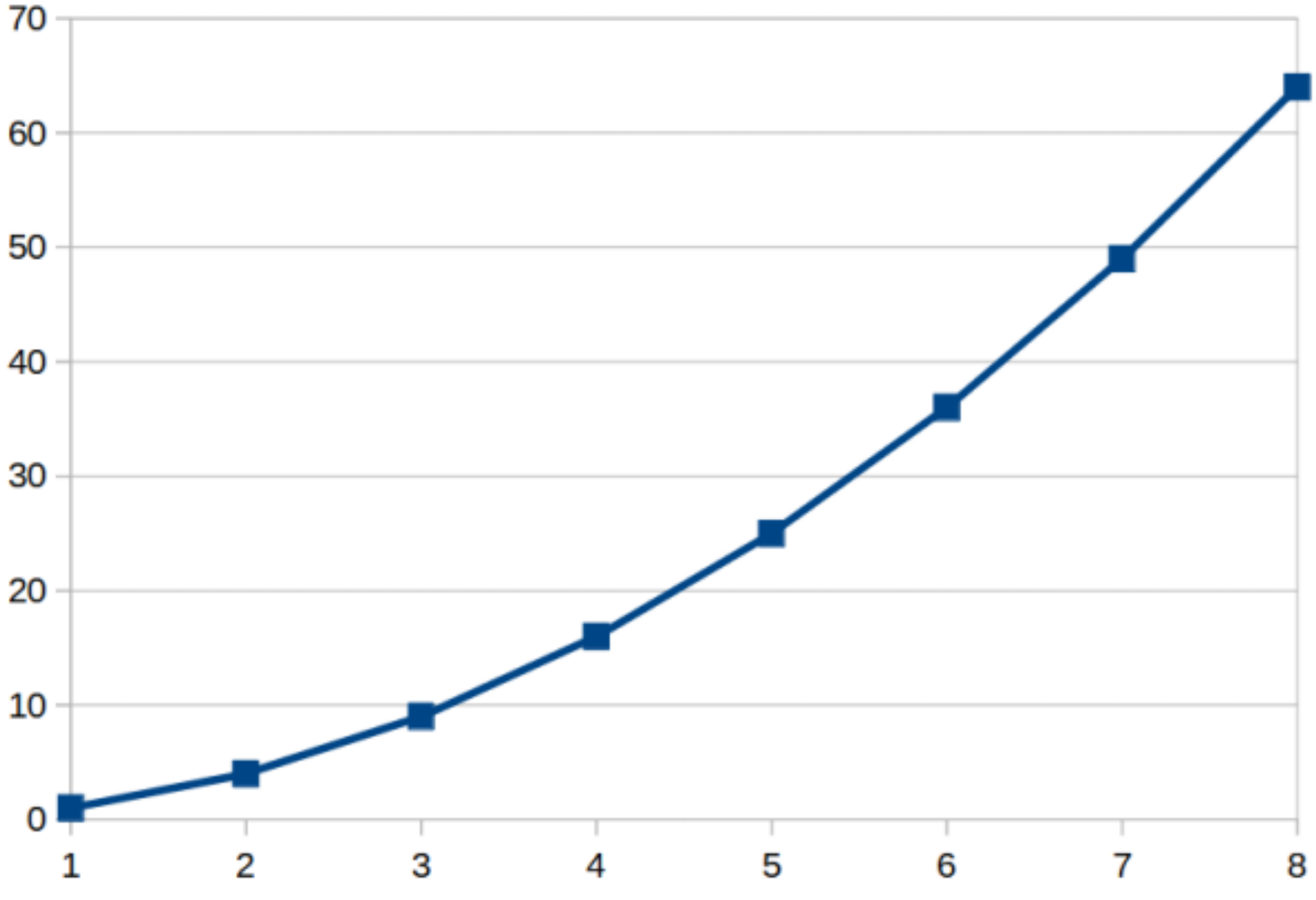
メソッドの中身が線形関数な場合がある(2/2)

```
function includes() {
  const items = [a,b,c,d,e];
  for (let i = 0; i < items.length; i++) {
    if (item == c) {
      return true; // => 2
    }
  }
  return false;
}
```

O(N^2)：二乗時間

```
// 重複を含む配列
const duplicated = [
  0,1,2,3,2,1,4,3,4,5,6,7,5,6,4,8,9,5,3,2
];
// 重複を取り除いた配列
const unique = [];
for (const duplicatedItem of duplicated) {
  let included = false;
  for (const uniqueItem of unique) {
    if (uniqueItem == duplicatedItem) {
      included = true;
      break;
    }
  }
  if (!included)
    unique.push(duplicatedItem);
}
console.log(unique); // => [0,1,2,3,4,5,6,7,8,9]
```

O(N^2)：二乗時間



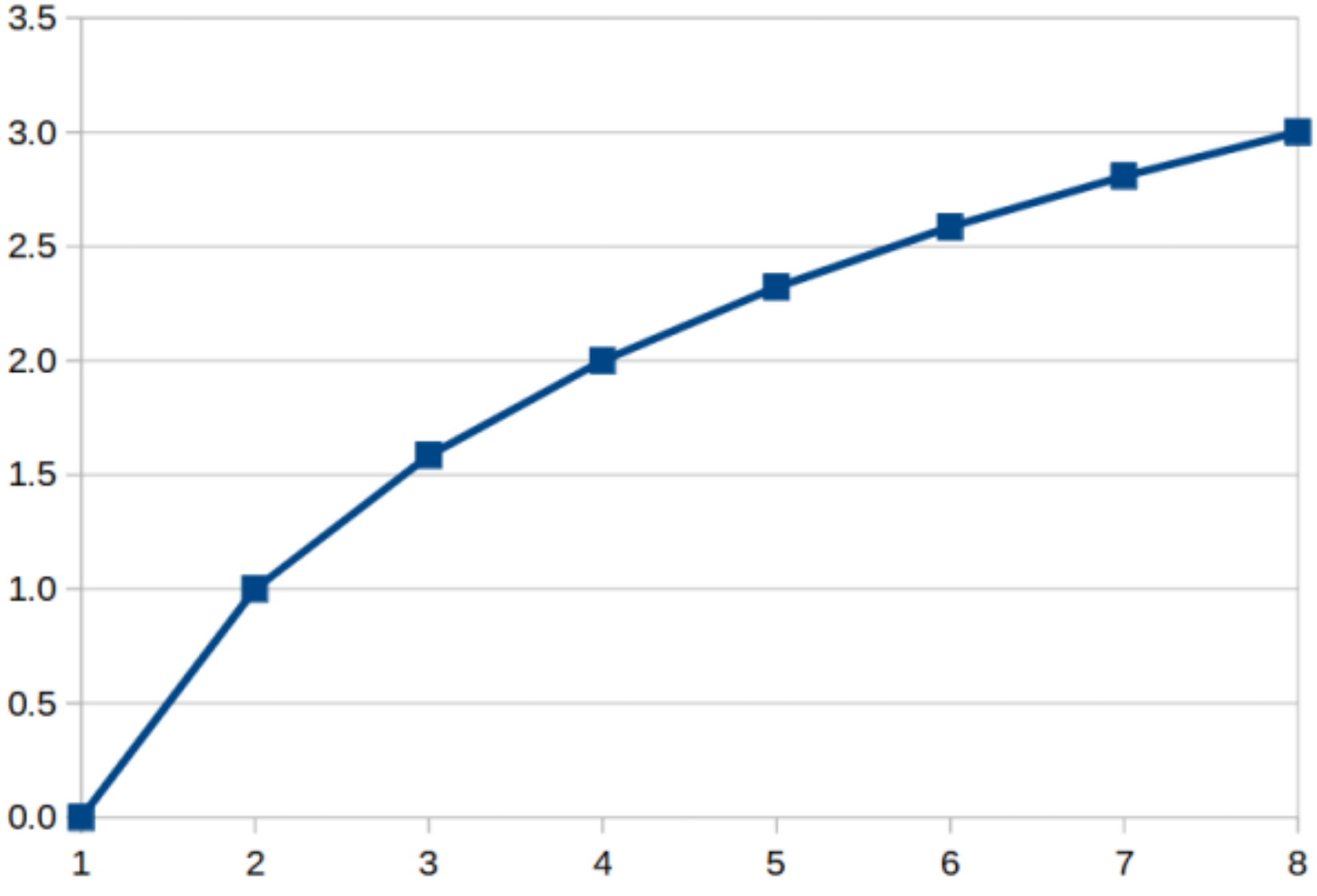
ループを使ってなくても安心できない

- ✓ メソッドの中身が（以下略）

O(logN)：対数

```
// 検索対象の配列（ソート済み）
const values = [0, 3, 6, 9, 12, 70, 102];
// 特定の値が何番目にあるかを二分探索で調べる
const target = 12;
let index = -1;
let minIndex = 0;
let maxIndex = values.length - 1;
while (minIndex <= maxIndex) {
  let middleIndex = Math.floor((minIndex + maxIndex) / 2);
  if (values[middleIndex] == target) {
      index = middleIndex;
      break;
  }
  else if (values[middleIndex] < target)
    minIndex = middleIndex + 1;
  else
    maxIndex = middleIndex - 1;
}
console.log(index); // => 4
```

O(logN)：対数



O(N!)：階乗関数(1/3)

巡回セールスマン問題

```
// ある都市から他の都市までの移動に要する時間のデータ
const cities = {
  tokyo:    { osaka: 2, hokkaido: 3, okinawa: 4,  kagawa: 5   },
  osaka:    { tokyo: 2, hokkaido: 5, okinawa: 3,  kagawa: 1   },
  hokkaido: { tokyo: 3, osaka: 5,    okinawa: 7,  kagawa: 6   },
  okinawa:  { tokyo: 4, osaka: 3,    hokkaido: 7, kagawa: 8   },
  kagawa:   { tokyo: 5, osaka: 1,    okinawa: 8,  hokkaido: 6 },
};
```

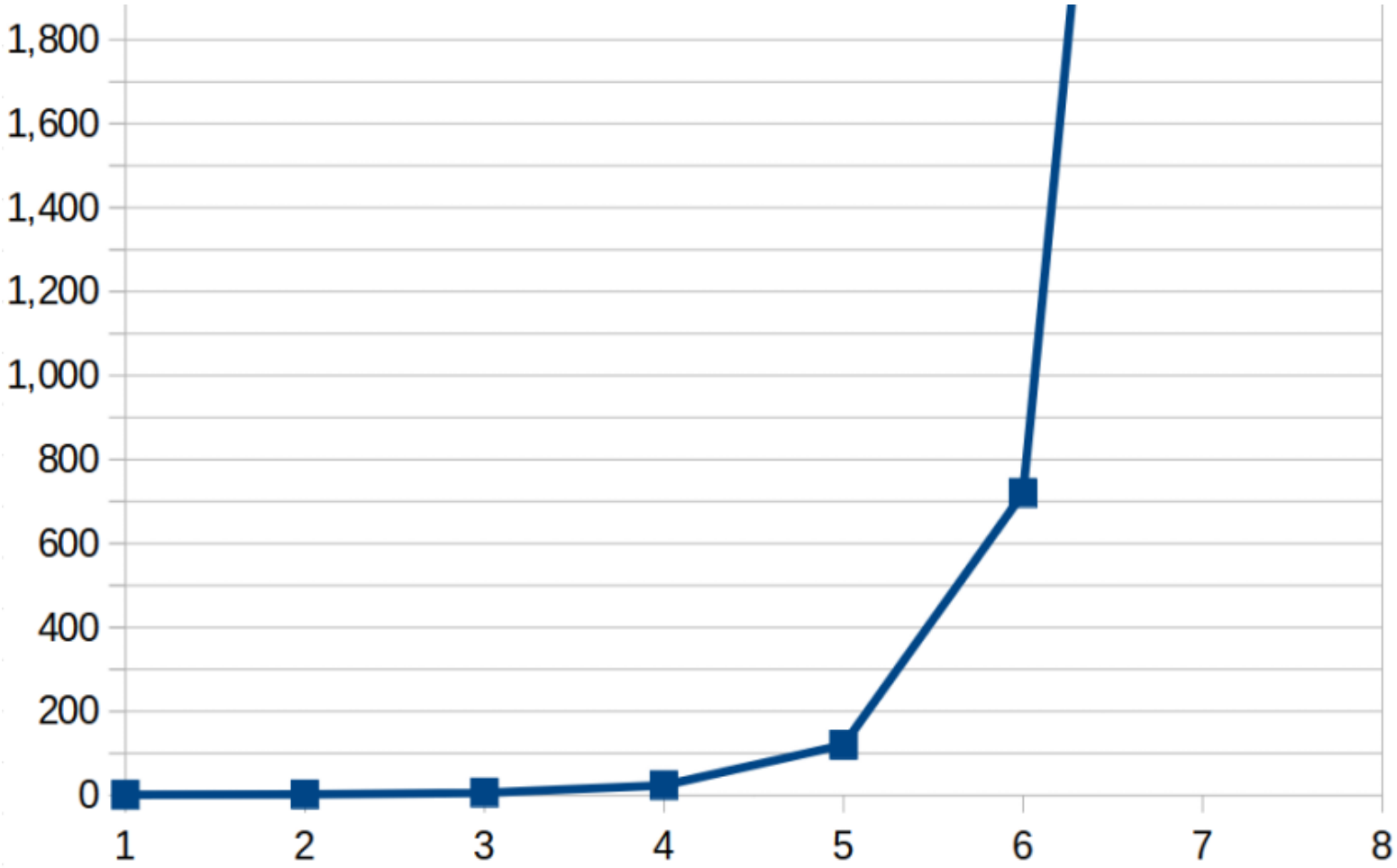
O(N!)：階乗関数(2/3)

```
// 配列から順列組み合わせを作る処理
function getPermutations(array) {
  const permutations = [];
  const nextPermutation = [];
  function permutate(array) {
    if (array.length === 0)
      permutations.push(nextPermutation.slice());
    for (let i = 0; i < array.length; i++) {
      array.push(array.shift());
      nextPermutation.push(array[0]);
      permutate(array.slice(1));
      nextPermutation.pop();
    }
  }
  permutate(array);
  return permutations;
}
```

O(N!)：階乗関数(3/3)

```
// 総当たりで移動時間を求めて、最短の移動パターンを見つける
const results = [];
for (const start of Object.keys(cities)) {
  const patterns = getPermutations(
    Object.keys(cities).filter(dest => dest != start)
  );
  for (const pattern of patterns) {
    let last;
    let total = 0;
    const route = [start, ...pattern, start];
    for (const current of route) {
      if (last)
        total += cities[last][current];
      last = current;
    }
    results.push({ route: route.join('-'), total });
  }
}
console.log(results.length);
// => 120
results.sort((a, b) => a.total - b.total);
console.log(results[0]);
// => { route: "tokyo-hokkaido-kagawa-osaka-okinawa-tokyo", total: 17 }
```

O(N!)：階乗関数



7件で5040回、8件で40320回

計算量を減らしたい！

計算量

- ✓ どういう結果を
- ✓ どういうデータから
- ✓ どのように求めるか

によって、計算量は大きく変わる

どうすれば計算量を減らせる？

計算量を減らすには(1/3)

- ✓ 計算方法を変える
  ＝アルゴリズムを工夫する

元のアルゴリズム：O(N^2)

```
// 重複を含む配列
const duplicated = [
  0,1,2,3,2,1,4,3,4,5,6,7,5,6,4,8,9,5,3,2
];
// 重複を取り除いた配列
const unique = [];
for (const duplicatedItem of duplicated) {
  let included = false;
  for (const uniqueItem of unique) {
    if (uniqueItem == duplicatedItem) {
      included = true;
      break;
    }
  }
  if (!included)
    unique.push(duplicatedItem);
}
console.log(unique); // => [0,1,2,3,4,5,6,7,8,9]
```

アルゴリズムを変えた：O(N)

```
// 重複を含む配列
const duplicated = [
  0,1,2,3,2,1,4,3,4,5,6,7,5,6,4,8,9,5,3,2
];
// 既に見つかった項目の情報
const found = {};
// 重複を取り除いた配列
const unique = [];
for (const item of duplicated) {
  if (found[item])
    continue;
  found[item] = true;
  unique.push(item);
}
console.log(unique); // => [0,1,2,3,4,5,6,7,8,9]
```

ループの実行回数を減らす！

- ✓ ループせず済ませるようにする（O(1)に寄せる）
- ✓ 直近の実行結果を使い回す（キャッシュ）

アルゴリズムは色々ある

例：並べ替え（ソート）

- クイックソート：O(NlogN)～O(N^2)
  - 結果は不安定
- バブルソート：O(N^2)
  - 結果は安定
- マージソート：O(NlogN)
  - 結果は安定

それぞれアルゴリズムには得手・不得手がある

例：

- ✓ ある程度揃ったデータだと速い
- ✓ データ件数が少ないと速い

など

先達の知見を活かそう

計算量を減らすには(2/3)

- ✓ データの構造の方を変える
  ＝**前加工する**

データ構造による制約：O(N)

```
// 色の名前とカラーコードのデータ（配列構造）
const COLOR_CODES = [
  { name: 'blue',   code: 0x0000FF },
  { name: 'green',  code: 0x00FF00 },
  { name: 'red',    code: 0xFF0000 },
  { name: 'yellow', code: 0xFFFF00 },
];
// 色の名前でカラーコードを特定する
let color;
for (const item of COLOR_CODES) {
  if (item.name == 'red') {
    color = item.code;
    break;
  }
}
console.log(color); // => 0xFF0000
```

データ構造による制約：O(1)

```
// 色の名前とカラーコードのデータ（ハッシュ構造）
const COLOR_CODE_BY_NAME = {
  blue:   0x0000FF,
  green:  0x00FF00,
  red:    0xFF0000,
  yellow: 0xFFFF00,
};
// 色の名前でカラーコードを特定する
let color = COLOR_CODE_BY_NAME['red'];
console.log(color); // => 0xFF0000
```

データ構造を変える(1/2)

```
// 色の名前とカラーコードのデータ（配列構造）
const COLOR_CODES = [
  { name: 'blue',   code: 0x0000FF },
  { name: 'green',  code: 0x00FF00 },
  { name: 'red',    code: 0xFF0000 },
  { name: 'yellow', code: 0xFFFF00 },
];
            ↓
// 色名をキーにしたハッシュ構造
const COLOR_CODE_BY_NAME = {
  blue:   0x0000FF,
  green:  0x00FF00,
  red:    0xFF0000,
  yellow: 0xFFFF00,
};
```

データ構造を変える(2/2)

```
const COLOR_CODES = [
  { name: 'blue',   code: 0x0000FF },
  { name: 'green',  code: 0x00FF00 },
  { name: 'red',    code: 0xFF0000 },
  { name: 'yellow', code: 0xFFFF00 },
];

const COLOR_CODE_BY_NAME = {};
for (const item of COLOR_CODES) {
  COLOR_CODE_BY_NAME[item.name] = item.code;
}
```

変換のコスト

- ✓ 何回も使うなら割に合う
- ✓ 1回しか使わなくても、
  応答性を挙げたいときにも
  （初期化に時間をかけてよければ）

計算量を小さくしやすいデータ構造にしよう

- ✓ 配列の全走査は基本的に遅い
- ✓ 配列は**よく検索するキー**でハッシュテーブルにしておく（インデックス）
- ✓ 読み込んだデータはとりあえずハッシュテーブルにしがち

計算量を減らすには(3/3)

「求めたい結果」を変えてしまう

- ✓ 厳密な結果でなく
  近似値でいいことにする
- ✓ **制約条件**を設ける

巡回セールスマン問題

- ✓ 各町を1回ずつ訪問したい
- ✓ すべての町を訪問したい
- ✓ なるべく短い時間で済ませたい

時間が最も短く済むのは
どのような巡回ルートか？

巡回セールスマン問題なら

- ✓ 最適解を求めることを諦めて、**現実的な計算時間の範囲で**ほどほどにいい結果を求める
- ✓ 巡回できる町の**最大数を制限する**

練習してみよう

- ✓ https://github.com
  /clear-code
  /readable-code-workshop
  /tree/master/20220804
  /0performance
  /try.docx または try.pdf