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(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
```

$O(N^2)$：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 duplicatedElem of duplicated) {
  let included = false;
  for (const uniqueElem of unique) {
    if (uniqueElem == duplicatedElem) {
      included = true;
      break;
    }
  }
  if (!included)
    unique.push(duplicatedElem);
}
console.log(unique); // => [0,1,2,3,4,5,6,7,8,9]
```

O(logN)：対数

```
1  // 検索対象の配列（ソート済み）
2  const values = [0, 3, 6, 9, 12, 70, 102];
3  // 特定の値が何番目にあるかを二分探索で調べる
4  const target = 12;
5  let index = -1;
6  let minIndex = 0;
7  let maxIndex = values.length - 1;
8  while (minIndex <= maxIndex) {
9     let middleIndex = Math.floor((minIndex + maxIndex) / 2);
10    if (values[middleIndex] == target) {
11          index = middleIndex;
12          break;
13    }
14    else if (values[middleIndex] < target)
15       minIndex = middleIndex + 1;
16    else
17       maxIndex = middleIndex - 1;
18 }
19 console.log(index); // => 4
```

O(N!)：階乗関数

```
// 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 },
};

// 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;
}

// 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(N²)から O(N)に最適化

```
1   // 重複を含む配列
2   const duplicated = [
3     0,1,2,3,2,1,4,3,4,5,6,7,5,6,4,8,9,5,3,2
4   ];
5   // 既に見つかった項目の情報
6   const found = {};
7   // 重複を取り除いた配列
8   const unique = [];
9   for (const item of duplicated) {
10    if (found[item])
11      continue;
12    found[item] = true;
13    unique.push(item);
14  }
15  console.log(unique); // => [0,1,2,3,4,5,6,7,8,9]
```