# O(1)：定数時間

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // 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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | // 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(N2)：二乗時間

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | // 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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | // 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(N!)：階乗関数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | // 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  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | // 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] |