Monaca

「情報Ⅰ」に向けたプログラミング研修会

～アルゴリズム編～

アシアル株式会社
アシアル情報教育研究所
岡本 雄樹

はじめに

自己紹介

- 名前
  - 岡本雄樹(アシアル情報教育研究所 所長)
- 著書
  - イラストでよくわかるPHP
  - WordPressプロフェッショナル養成読本
  - Monacaで学ぶはじめてのプログラミング
- メッセージ
  - 「コンピューター」「インターネット」「プログラミング」私は高校生の時にそれらと出会うことで人生が拓けました。先生方とMonacaによるアプリ開発を通じて、情報技術の活用方法や作品作りの楽しさを広めてまいります。

■ 教材サイトのご案内

- あんこエデュケーション
  - https://anko.education/
- 教科情報研修資料
  - https://anko.education/joho

■ プロジェクトのインポート

- 教材サイトに各種モデルのサンプルプロジェクトがあります
- Monacaでインポートして進めます
- 予めMonacaのログインと空きプロジェクト数の確保をお願いします

■ 学習15 アルゴリズムの比較

- はじめに
- サーチ(探索)
  - リニアサーチ(線形探索)
  - バイナリサーチ(2分探索)
- ソート(並び替え)
  - セレクションソート（挿入ソート）
  - クイックソート
  - 再帰
  - 配列の入れ替え
  - マージ
  - マージソート

はじめに

はじめに

■ はじめに

└ アルゴリズムとは

└ 問題を処理するための具体的な手段

└ 特定の問題を解くためのアルゴリズムは1つとは限りません

└ 今回学ぶアルゴリズム

└ 配列の値から、特定の値が存在するかを探索(サーチ)したり、順番に並び替えたり（ソート）するアルゴリズムを学習します。

└ 前提知識

└ 分岐・繰り返し・関数・配列などを駆使するため、事前にこれらを学習する必要があります。

└ 追加知識

└ 配列の要素を入れ替えたり統合する処理、また関数が処理の中で自分を再度呼び出す「再帰」処理が登場します。

■ アルゴリズムの比較ポイント

- 計算コスト
  - 値の比較や入れ替えには計算コストが掛かります。
  - 少ない回数で処理できるに越したことはありません。
- 記憶コスト
  - 計算した値などを一時的に別の場所に記憶するには追加の記憶領域が必要です。
  - 記憶領域を節約できるに越したことはありません。
  - しかし、記憶領域を代償に計算コストを下げられる場合もあります。

サーチ

■ 単純なサーチにかかる計算コストは幾つ？

- 7個の値から特定の値を探すコストは1～7の範囲
  - 1個目で見つかる場合もあります
  - 7個全部確認して最後に見つかる場合もあります
  - コスト上限は7
  - オーダー記法では0(n)と表します

■ 以下のようなサーチアルゴリズムをプログラムで書いてみましょう

1. 特定の数値を探す
2. 一番小さい数字を探す
3. 一番大きい数字を探す

■ 配列(サンプル)

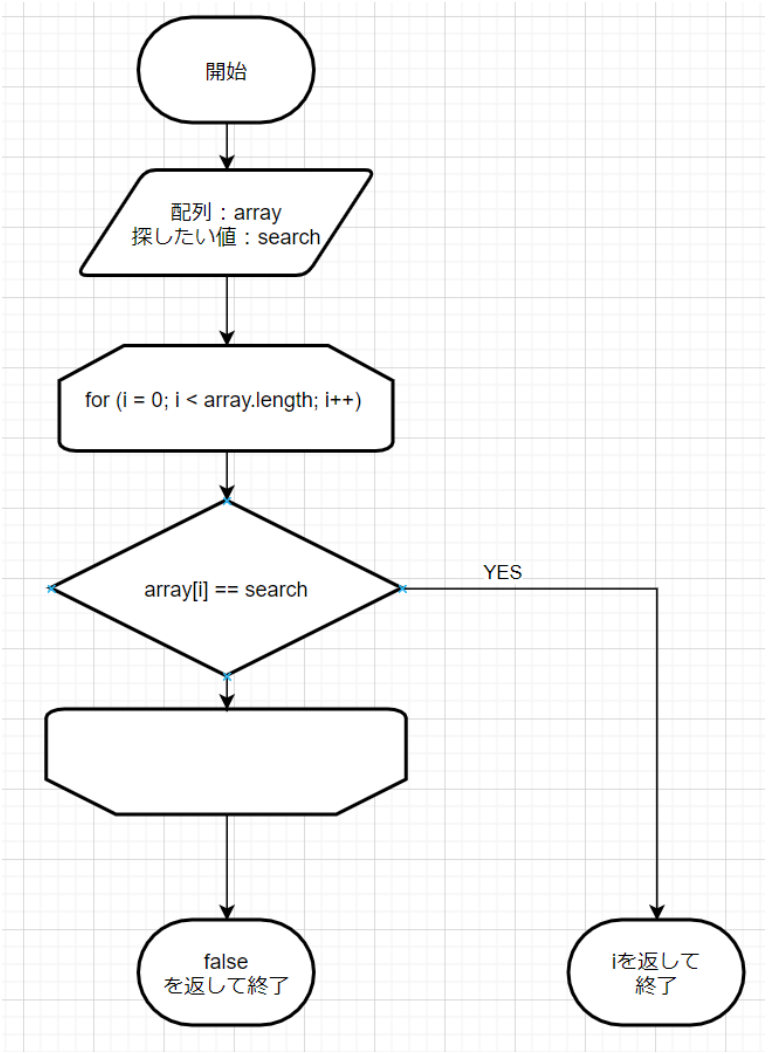
キー	0	1	2	3	4	5	6
バリュー	61	15	82	77	21	32	53

■ コピペ用の配列値

```
array = [61, 15, 82, 77, 21, 32, 53];
```

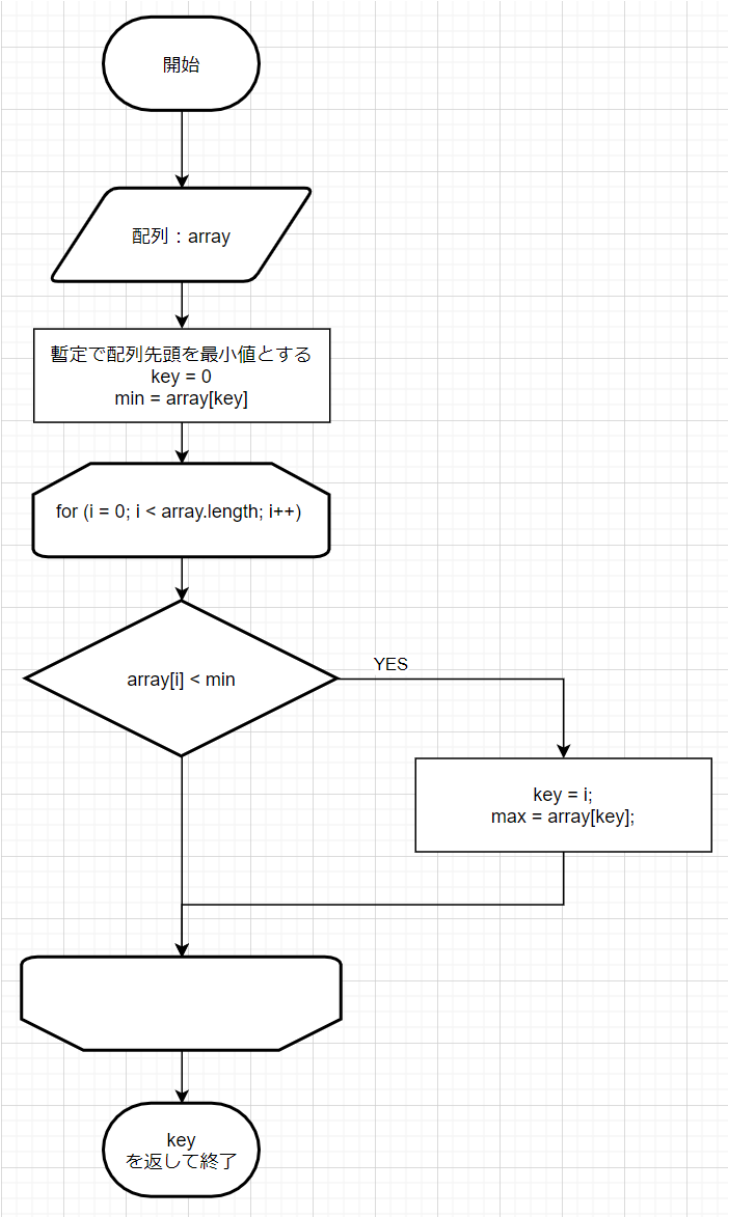
① 特定の数値を探す（今回は「82」）

```
function linerSearch(array, search) {
    for (let i = 0; i < array.length; i++) {
        if  (array[i] == search) {
            return i;
        }
    }
    return false;
}

let array = [61,15,82,77,21,32,53];
let search = 82;
let result = linerSearch(array, search);
```



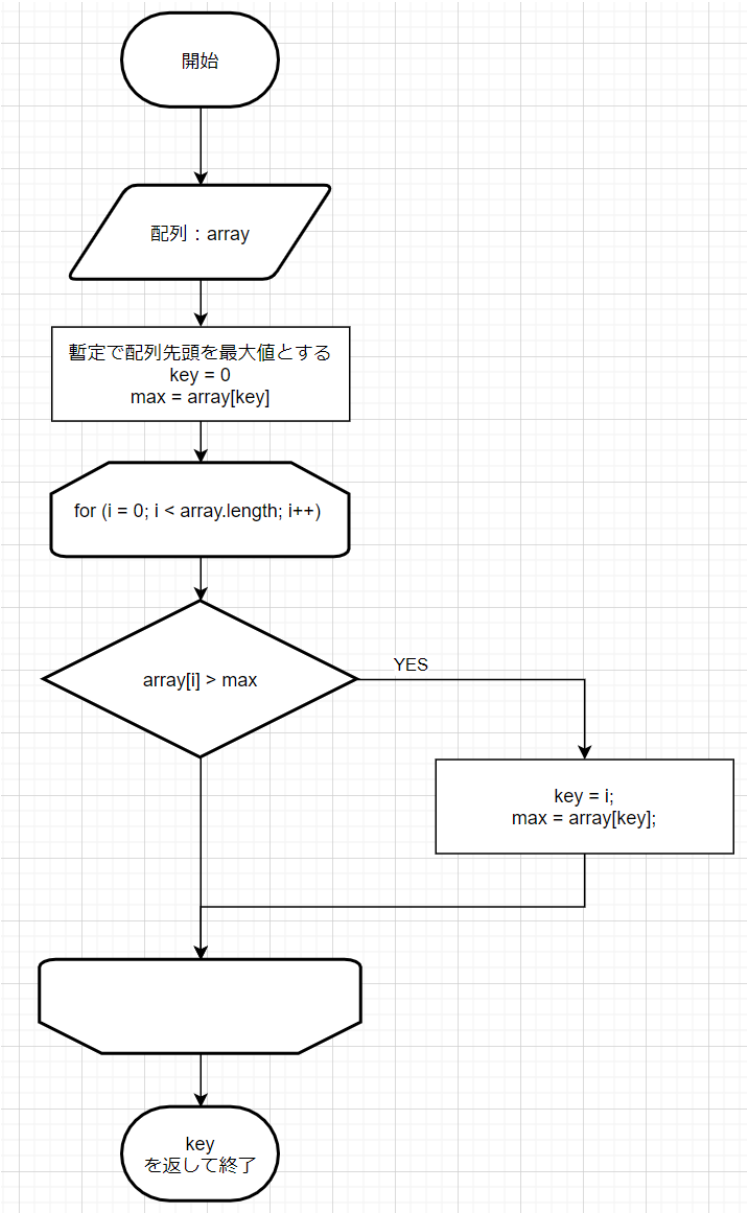
② 一番小さい数字のキーを探す

```
function findMin(array) {
    let key = 0;
    let min = array[key];

    for (let i = 0; i < array.length; i++) {
        if  (array[i] < min) {
            min = array[i];
            key = i;
        }
    }
    return key;
}
```



③ 一番大きい数字のキーを探す

```
function findMax(array) {
    let key = 0;
    let max = array[key];

    for (let i = 0; i < array.length; i++) {
        if  (array[i] > max) {
            max = array[i];
            key = i;
        }
    }
    return key;
}
```



■ アルゴリズムの比較

- 先頭から順番みていく探し方を「リニアサーチ（線形探索）」と呼びます
  - n個のデータから一つの値を探索するために最大n回の比較が必要です
  - オーダー記法では0(n)と表します
  - データが1,000個ある場合、最大1,000回の比較が必要です
  - 探したいデータが100個あるとしたら1,000 x 100個で100,000回必要です
    - データの数や探索回数が多い場合は大変なので、もっとコストの低い方法を検討したい

オーダー記法	意味	例
O(n)	計算コスト上限はnに比例	リニアサーチなど
O(log n)	log n < n nよりも小さい値に比例	バイナリサーチなど

■ もっと少ないコストで探索する方法

└ 配列をソート済みにすればもっと少ないコストで探索可能です

■ 配列(サンプル)

キー	0	1	2	3	4	5	6
バリュー	25	33	43	51	66	71	88

■ 探索例「43」を探す

- とりあえず中央の配列要素[3]と比較

キー	0	1	2	3	4	5	6
バリュー	25	33	43	51	66	71	88

- 「51」は「43」より大きい、故に43が存在するのは[0]～[2]の何処かに絞られる
- 次に[0]～[2]の中央の配列要素[1]と比較

キー	0	1	2	3	4	5	6
バリュー	25	33	43	51	66	71	88

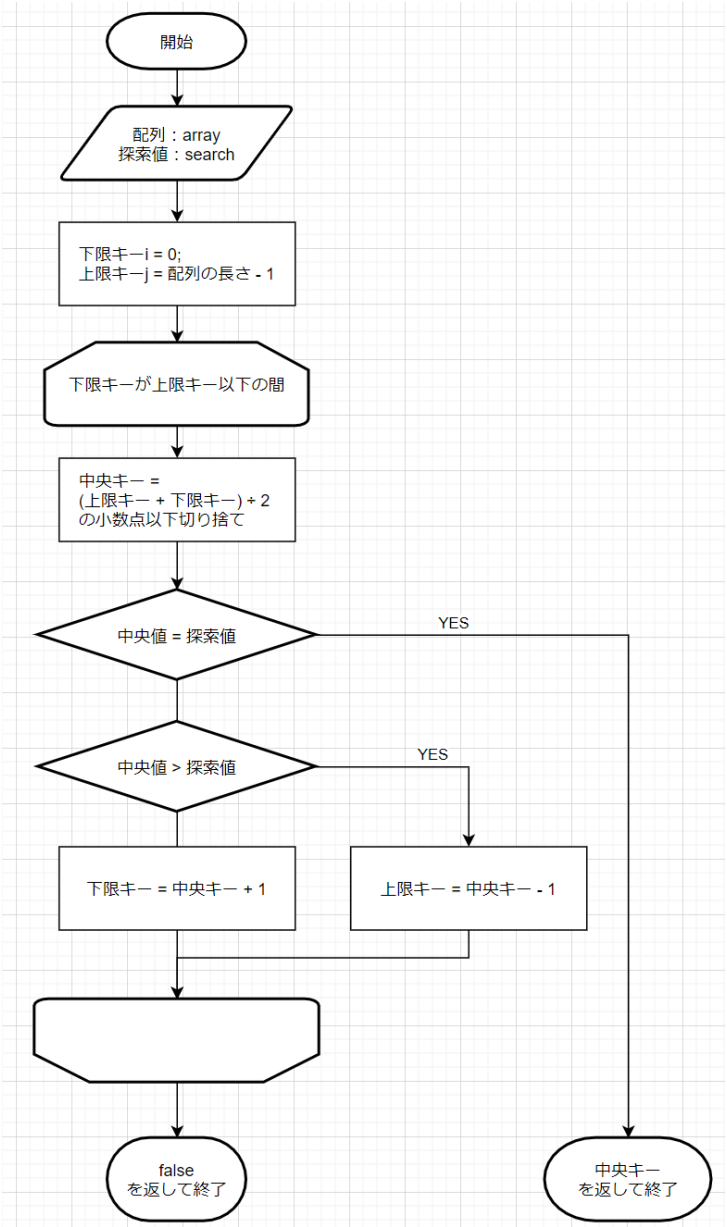
- 「33」は「43」より小さい、故に43が存在するのは[2]に絞られる

キー	0	1	2	3	4	5	6
バリュー	25	33	43	51	66	71	88

- もし、何処にも値が無い場合は？？？

バイナリサーチ

```
function binarySearch(array,value) {
    let i = 0;
    let j = array.length - 1;
    let m;
    while (i <= j) {
        m = Math.floor((i + j) / 2);
        if (array[m] == value) {
            alert(m + "番目にありました！");
            return m;
        } else {
            if (array[m] > value) {
                j = m - 1;
            } else {
                i = m + 1;
            }
        }
    }
    alert("ありませんでした");
    return false;
}
let sortedArray = [23,33,43,51,66,71,88];
let value = 43;
let result = binarySearch(sortedArray,value);
```



■ サーチアルゴリズムのまとめ

- リニアサーチ
  - 配列を順番に探す方法
  - コストはO(n)
- バイナリサーチ
  - ソート済みの配列を分割して探す方法
  - n個のデータから一つの値を探索するためのコストがnより小さい
  - オーダー記法ではO(log n)と表します
  - ただしソートのコストが別途必要です
    - 何度も探索したい場合、ソートのコストは最初の一回だけ払えば良いので有利です

オーダー記法	意味	例
O(n)	計算コスト上限はnに比例	リニアサーチなど
O(log n)	log n < n nよりも小さい値に比例	バイナリサーチなど

ソートアルゴリズム

■ 配列から小さい値を探してソートする（選択ソート）

1. 配列から一番小さい数字を探す
2. 探した値を、「別の配列に詰め直す」か「左側の値と入れ替える」

■ 配列(サンプル)

キー	0	1	2	3	4	5	6
バリュー	61	15	82	77	21	32	53

■ 別の配列に詰め直すパターンの流れ

└ 直感的ですが、配列を2つ用意する必要があります(メモリが無駄)

0	1	2	3	4	5	6
61	15	82	77	21	32	53

⇨

0
15

0	1	2	3	4	5
61	82	77	21	32	53

⇨

0	1
15	21

0	1	2	3	4
61	82	77	32	53

⇨

0	1	2
15	21	32

0	1	2	3
61	82	77	53

⇨

0	1	2	3
15	21	32	53

■ 配列の中で並び替えるパターンの流れ

- ちょっとわかりにくいですが、配列は1つ、かつ、要素の数も増減しません

0	1	2	3	4	5	6
61	15	82	77	21	32	53

キー[0]から[6]の範囲で
一番小さい値「15」を
左端(キー[0])と入れ替える

0	1	2	3	4	5	6
15	61	82	77	21	32	53

キー[1]から[6]の範囲で
一番小さい値「21」を
キー[1]と入れ替える

0	1	2	3	4	5	6
15	21	82	77	61	32	53

キー[2]から[6]の範囲で
一番小さい値「32」を
キー[2]と入れ替える

0	1	2	3	4	5	6
15	21	32	77	61	82	53

キー[3]から[6]の範囲で
一番小さい値「53」を
キー[3]と入れ替える

■ 配列の交換

- ソートを効率的に行うためには配列の値を入れ替える処理が必要です
- 言語の命令として用意されている場合もあります

配列の交換関数の例

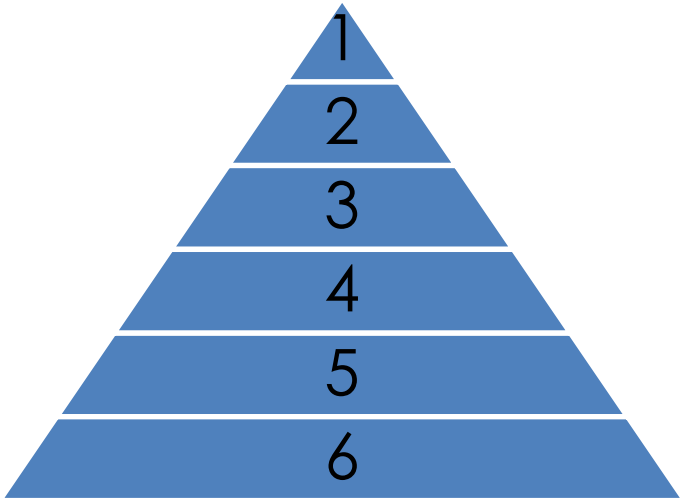
```
// 配列の値を入れ替える
function swap(array, key1, key2) {
    temp = array[key1];
    array[key1] = array[key2];
    array[key2] = temp;

    console.log("swap:", array[key1], "⇔" , array[key2] ,"array:", array);
}
```

配列の交換例

```
// 配列の値を入れ替える
function swap(array, key1, key2) {
    temp = array[key1];
    array[key1] = array[key2];
    array[key2] = temp;

    console.log("swap:", array[key1], "⇔" , array[key2] ,"array:", array);
}
let array = [61,15,45,82];

swap(array, 0, 1);
swap(array, 1, 2);

console.log(array);
```

■ 配列を選択ソートするコスト

- 値が7個なら、最小値を求めるための探索でまず6回比較が発生します
- 未ソートの値が6個に減るので次の探索は5回発生します
- 4回、3回、2回と未ソートの配列が無くなるまで繰り返します
- 合計21回の比較が必要です（1 + 2 + 3 + 4 + 5 + 6）
  - コストは値の数の二乗に比例、オーダー式で表すと$O(n^2)$
  - つまりコストが大きい、値の数が増えるととても大変



■ アルゴリズムの比較

- 先頭から最小値を探索して順番に選択して取り出すやり方を「セレクションソート（選択ソート）」と呼びます
  - n個のデータをソートするためにn + (n - 1) + (n - 2)... の比較が必要です
    - nの三角数と同じ回数
  - オーダー記法では$O(n^2)$と表します
    - nの三角数回はn^2より小さいけど、オーダー記法上ではn^2と表します

オーダー記法	意味	例
O(n)	計算コスト上限はnに比例	リニアサーチなど
O(log n)	log n < n nよりも小さい値に比例	バイナリサーチなど
$O(n^2)$	計算コスト上限はnの2乗に比例	セレクションソートなど
O(n log n)	n < (n log n) < n^2 nの2乗より小さい値に比例	クイックソート マージソートなど

セレクションソート

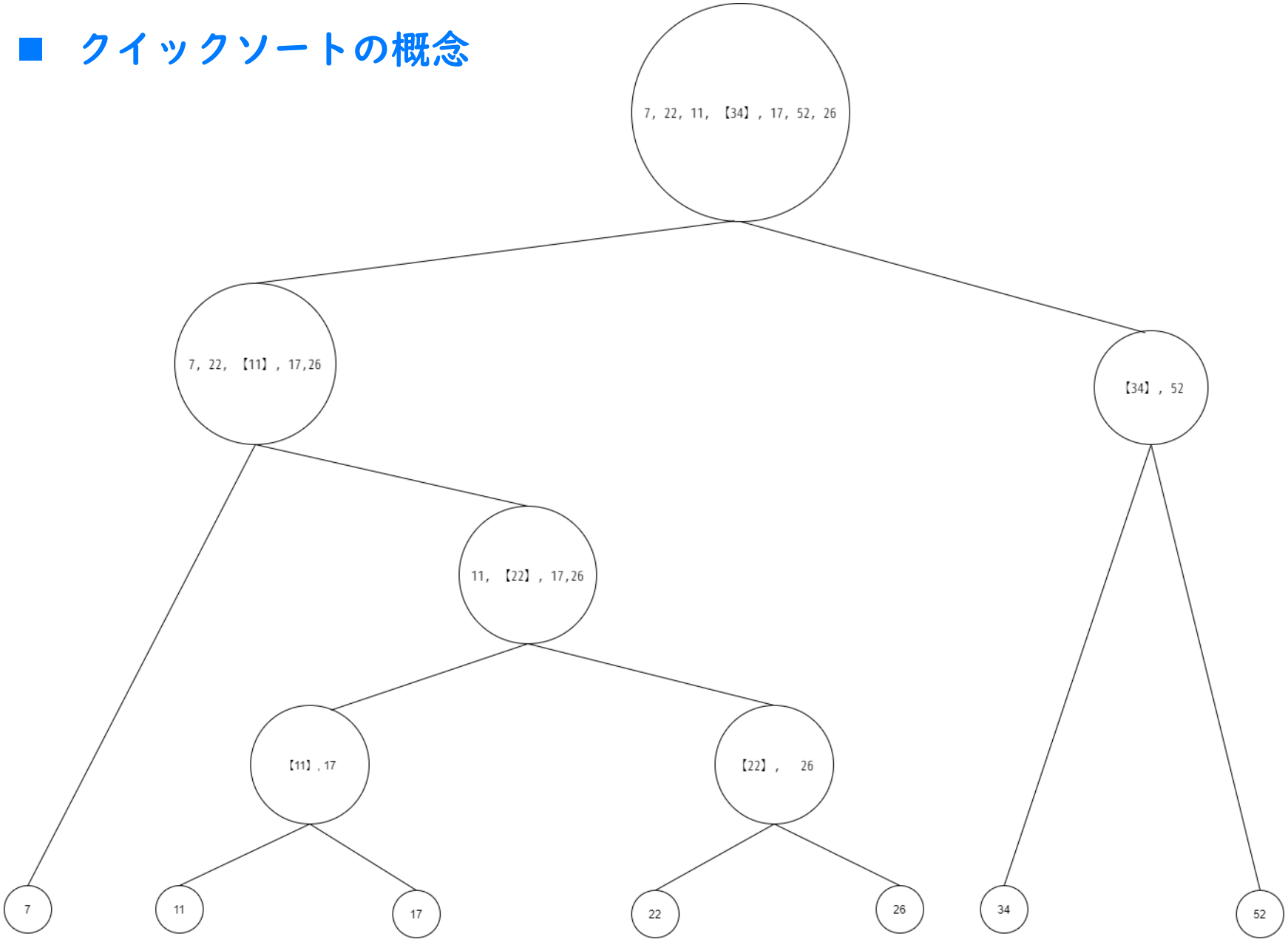
```
function swap(array, key1, key2) {
    temp = array[key1];
    array[key1] = array[key2];
    array[key2] = temp;
    console.log("swap:", array[key1], "⇔" , array[key2] ,"array:", array);
}

function selectionSort(array) {
    for (let i = 0; i < array.length; i++) {
        for (let j = 0; j < i + 1; j++) {
            if (array[j] > array[i]) {
                swap(array, i, j);
            }
        }
    }
}
let array = [61,15,82,77,21,32,53];
selectionSort(array);
console.log(array);
```

クイックソート

■ クイックソートの紹介

- クイックソートは最も高速なソートと言われています
- 実装には「配列操作」に加え「繰り返し」や「繰り返しの中断」、「分割統治法」や「再帰」などを駆使します。
  - そのため、難易度は高めです
    - 先に他の技術を学びましょう

■ クイックソートの概念



■ クイックソートによるswapの様子

swap	0	1	2	3	4	5	6
①	7	22	11	34	17	52	26
②	7	22	11	26	17	52	34
③	7	11	22	17	26	52	34
④	7	11	22	26	17	52	34
⑤	7	11	17	22	26	52	34
	7	11	17	22	26	34	52

クイックソートを補助する関数群

```
// 配列の値を入れ替える
function swap(array, key1, key2) {
    temp = array[key1];
    array[key1] = array[key2];
    array[key2] = temp;

    console.log("swap:", array[key1], "⇔" , array[key2] ,"array:", array);
}
// 配列からピボッド以上の値のキ□を探す
function leftSearch(array, m, i) {
    while (array[i] < array[m]) {
        i++;
    }
    return i;
}

// 配列からピボッド以下の値のキ□を探す
function rightSearch(array, m, j) {
    while (array[j] > array[m]) {
        j--;
    }
    return j;
}
```

クイックソート関数

```
function quicksort(array, start, end) {
    m = Math.floor((start + end) / 2);  // 真ん中のキー(ピボッド)を求める
    i = start;                          // iは左側のキー
    j = end;                            // jは右側のキー

    console.log("quicksort:","m:",m,"i:",i,"j:",j, "array:",array);

    while (i < j) { // 左側のキーが小さければループし続ける
        i =  leftSearch(array, m, i); // 配列の左側からピボッド以上の値のキーを探してiに代入
        j = rightSearch(array, m, j); // 配列の右側からピボッド以下の値のキーを探してjに代入

        if (i >= j) { // 左右のキーが同じか左の方が大きかったらループ終了
            break;
        }

        swap(array, i, j); // 左右のキーを元に値を交換

        if (i == m) { // もし左のキーとピボッドが同じなら、ピボッドを右のキーにする
            m = j;
        } else if (j == m) { // もし右のキーとピボッドが同じなら、ピボッドを左のキーにする
            m = i;
        }
        i++; // 左のキーを一つ右に進める
        j--; // 右のキーを一つ左に進める
    }

    // startが左のキーを一つ減らした数値より小さければ左を半分に分割して再帰的にクイックソート
    if (start < i - 1) {
        quicksort(array, start, m - 1);
    }
    // endが右のキーを一つ足した数値より大きければ右を半分に分割して再帰的にクイックソート
    if (end > j + 1) {
        quicksort(array, m + 1, end);
    }
}
```

クイックソート関数の呼び出し例

```
array = [7,22,11,34,17,52,26];

console.log("start",array);

quicksort(array, 0, array.length-1)

console.log("end");
```

マージソート

■ マージソートの紹介

- クイックソートよりも少し易しいアルゴリズムとして、マージソートを紹介します
  - コストはO(n log n)です
  - クイックソートよりも前に発明されたソートで、現在でも使われているソート技術です
- マージとは
  - ソート済みの配列同士を、ソート順を維持しながら統合する方法
  - 配列同士の先頭の値を比較しながら統合していく
- マージソートとは
  - マージを応用してソートするアルゴリズム
- 学習のポイント
  - 解説では配列を２分割しますが、実際にももっと分割します
  - 分割の処理では「再帰」処理を活用します
  - 再帰はクイックソートでも使用しているため、これを先に理解する必要があります

■ 分割統治法の検討

- 配列の要素数が大きいとセレクションソートのコストはo(n²)で膨れ上がる
- 配列を2つに分けてソートしたらコストは減る？
  - 2つの配列をそれぞれソート
  - ソート後の2つの配列を後で統合（マージ）する

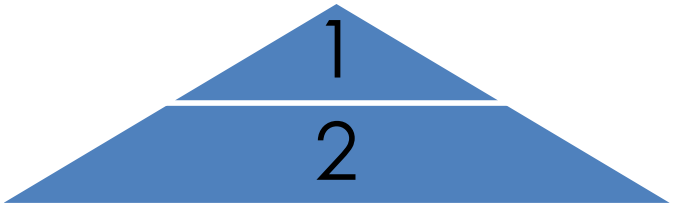
■ 配列(サンプル)

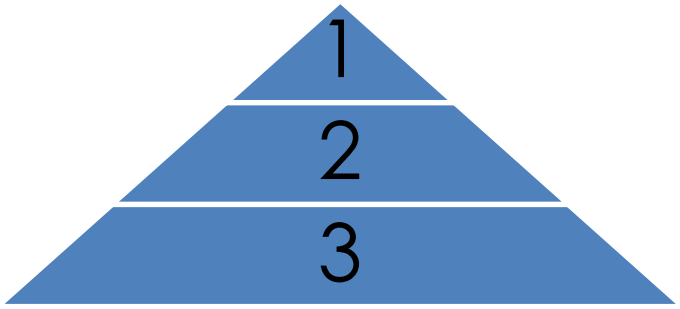
0	1	2
61	15	82

3	4	5	6
77	21	32	53

■ 分割した配列のソートコスト

- ここまでは、合計9回の比較で済む





■ ソート済みの配列をマージ(統合)する

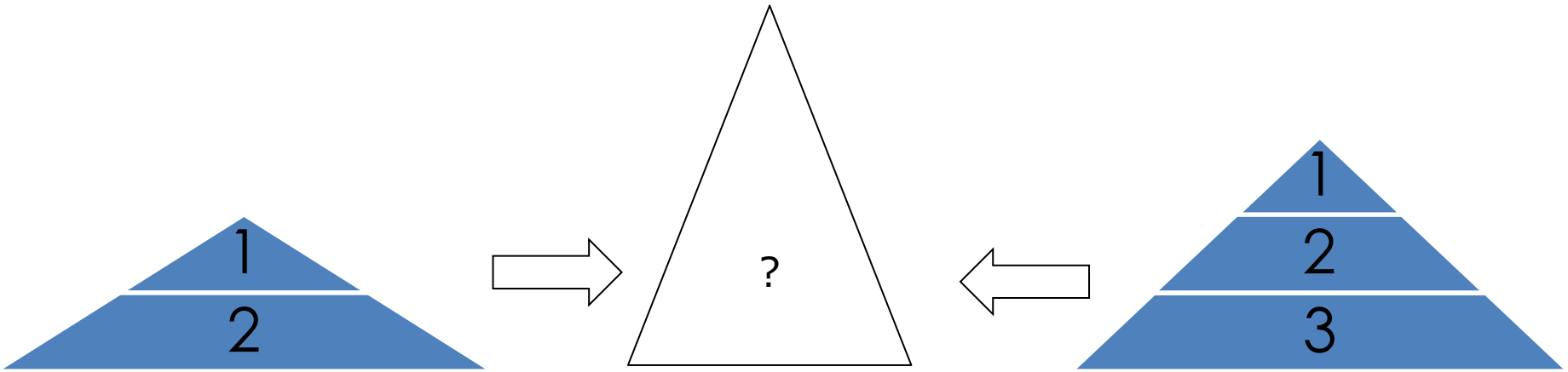
└ ソート済みの配列を統合するコストが、追加で幾ら必要か？

■ ソート済みの配列

0	1	2
15	61	82

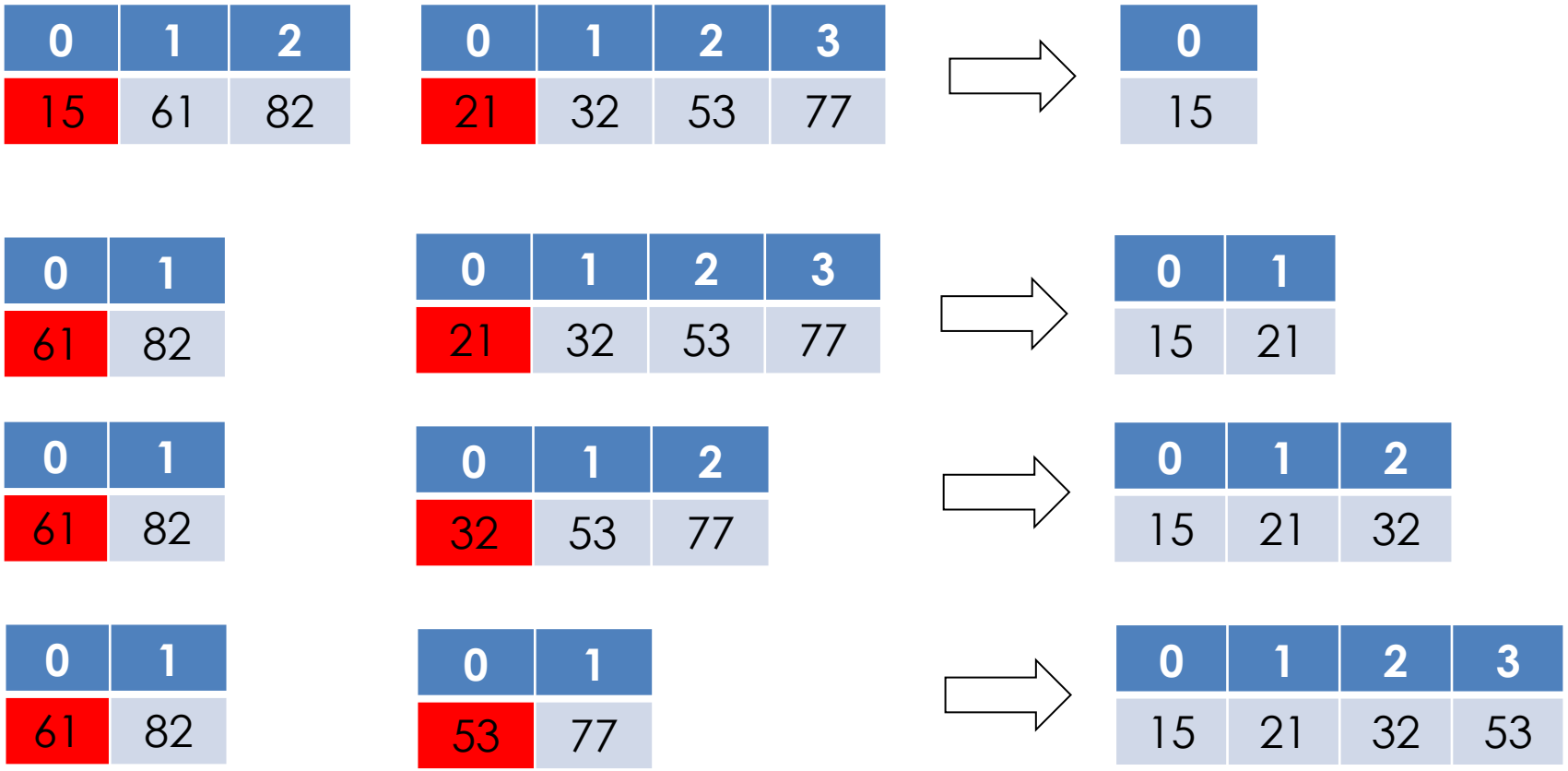
0	1	2	3
21	32	53	77

■ 統合のイメージ



■ ソート済みの配列をマージ(統合)する流れ

- 2つの配列の先頭同士を比較すればOK
  - コストはo(n)、つまり、マージのコストは低い



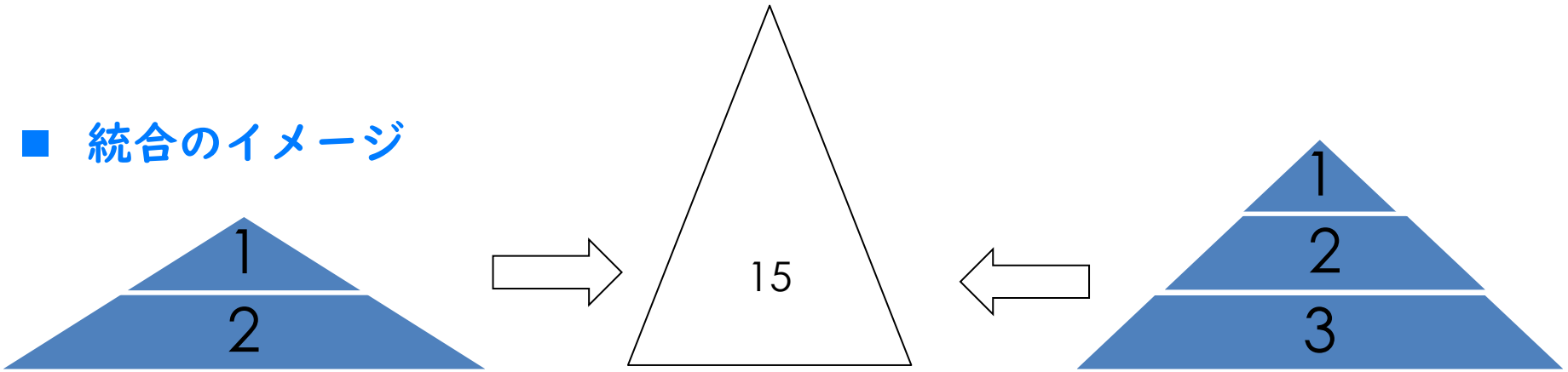
■ マージ(統合)のコスト

- 要素3個のソート済み配列と要素4個のソート済み配列は6回の比較でマージ可能

■ 全体のコストを集計

- 要素3個の配列をソート：3回
- 要素4個の配列をソート：6回
- 上記配列をマージソート：6回
- 合計15回の比較でソートが完了
- 単純に選択ソートした場合(21回)よりもコストが低い

■ 統合のイメージ



マージ関数

```
// 2つのソート済み配列を統合する関数
function merge(left, right) {
    // 統合結果を格納する配列を初期化
    let array = [];
    // 各配列の要素数を元に比較回数を決定
    let count = left.length + right.length;
    //各配列の末尾に巨大な数値を追加。この値が統合配列に混入することはない

    left.push(Number.MAX_VALUE);
    right.push(Number.MAX_VALUE);
    // 各配列の値を小さい順に統合
    for (let i = 0; i < count; i++) {
        if (left[0] < right[0]) {
            array.push(left.shift());
        } else {
            array.push(right.shift());
        }
    }
    // 統合配列を返す。
    return array;
}
```

マージソート関数

```
// 配列を再帰的に分割してmerge関数に引き渡す関数
function mergeSort(array) {
    // 配列要素が1以下で呼ばれたらreturnして終了
    if (array.length <= 1) {
        return array;
    }
    // 配列を中央で分割して二つに分ける
    let middle = Math.floor(array.length / 2);
    let left   = array.slice(0, middle);
    let right  = array.slice(middle);
    // 分けた配列を更にmergeSort関数に渡す(再帰的処理)
    let array1 = mergeSort(left);
    let array2 = mergeSort(right);
    // 戻ってきた結果をmerge処理してreturnして終了
    console.log("array:", array1, array2);
    return merge(array1, array2);
}
```

マージソート呼び出し部分関数

```
let array = [61,15,82,77,21,32,53];
mergeSort(array);
```

再帰

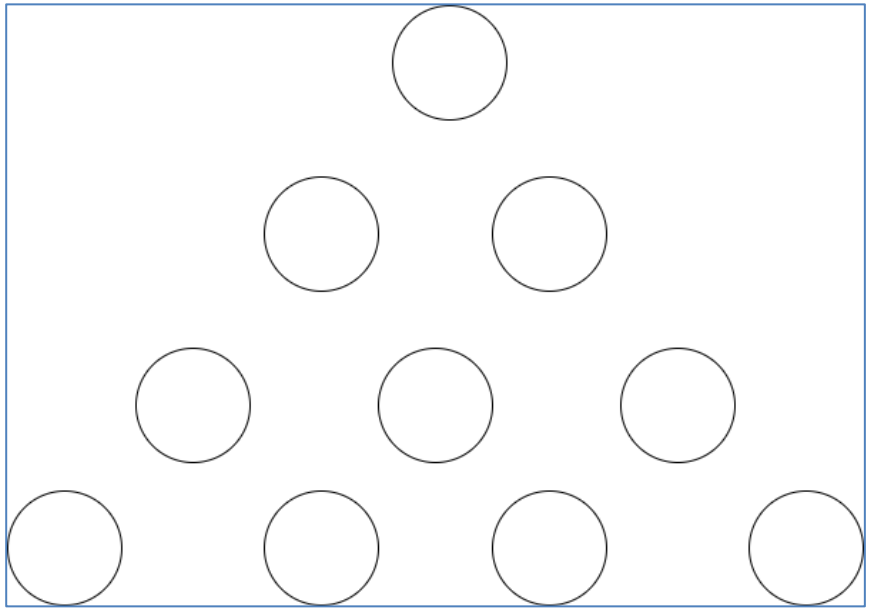
■ 再帰とは

- 関数の処理の中で自分を呼び出す処理
  - つまりループ処理です
  - 何も考えずに実行すると無限ループになります
  - if文などを使って条件付きで呼び出すなどの工夫が必要です
- 特定の問題を解決する際に、for文のループより上手く記述できます
  - たとえばクイックソート
  - あるいはマージソート

■ 再帰のお題

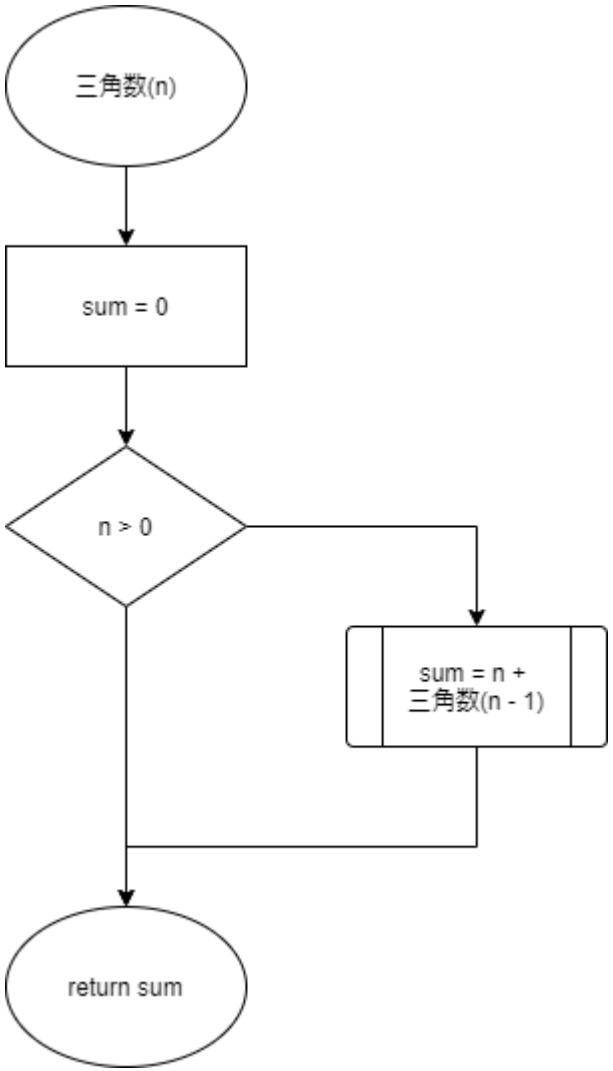
- 1 + 2 + 3 + 4 のような三角数の計算を行う関数を作りたい
  - 再帰バージョンとfor文バージョンを作成してください

■ 三角数のイメージ



三角数

```
function sankakusuu(n) {
    let sum = 0;

    if (n > 0) {
        sum = n + sankakusuu(n - 1);
    }
    console.log(n, sum);
    return sum;
}

sankakusuu(4)
```



三角数

```
function sankakusuuLoop(n){
    let sum = 0;

    for (let i = 1; i <= n; i++) {
        sum += i;
        console.log(i, sum);
    }
    return sum;
}

sankakusuu(4)
```