

アルゴリズムの授業

1. 線形探索 (Linear Search)

1.1 概要

配列（またはリスト）の先頭から順番に目標値と比較し、一致したらその位置を返す。見つからなければ最後まで調べ終えて「未発見」を返す。

1.2 計算量

- 最良ケース： $O(1)$ … 先頭にあった場合
- 平均ケース： $O(n/2) \approx O(n)$
- 最悪ケース： $O(n)$ … 存在しない場合や末尾にあった場合
- 空間計算量： $O(1)$

1.3 特徴・利点・欠点

- 利点
 - ソート不要。要素の順序を問わない
 - 実装が非常にシンプル
- 欠点
 - 要素数が大きいと比較回数も増大し、性能悪化
 - 頻繁に検索する場合は不向き

1.4 実装例 (C++)

```
// 配列 arr[n] の中から key を線形探索し、見つければそのインデックスを、  
// 見つからなければ -1 を返す  
int linearSearch(const std::vector<int>& arr, int key) {  
    for (size_t i = 0; i < arr.size(); ++i) {  
        if (arr[i] == key) {  
            return static_cast<int>(i);  
        }  
    }  
    return -1;  
}
```

1.5 適用場面

- データ数が少ない、あるいは検索回数が少ない場合
- 配列がソートされていない、かつソートコストをかけたくない場合

2. 二分探索 (Binary Search)

2.1 概要

ソート済み配列の中央要素と比較し、目標値が小さければ左半分、大きければ右半分を再帰的（またはループ）に探索。範囲を半分ずつ絞っていく。

2.2 計算量

- 最良・平均・最悪ケース：いずれも $O(\log n)$
- 空間計算量： $O(1)$ （ループ版）／ $O(\log n)$ （再帰版のスタック分）

2.3 特徴・利点・欠点

- 利点
 - 大量データでも高速
- 欠点
 - 事前にソートが必要 (ソートコスト $O(n \log n)$)
 - 配列へのランダムアクセスが前提
 - 実装ミス (境界条件) でバグを生みやすい

2.4 実装例 (C++)

```
// ソート済み vector の二分探索
int binarySearch(const std::vector<int>& arr, int key) {
    int left = 0;
    int right = static_cast<int>(arr.size()) - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == key) {
            return mid;
        } else if (arr[mid] < key) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}
```

2.5 適用場面

- データが事前にソートされている
- 検索回数が多く、毎回高速に探したい場合

3. ハッシュ探索 (Hash Search)

3.1 概要

ハッシュ関数でキーからテーブルの領域 (バケット) を計算し、直接アクセス。バケット内はリストや連想配列で衝突解消 (チェイニング、オープンアドレスなど) する。

3.2 計算量

- 平均ケース: $O(1)$
- 最悪ケース: $O(n)$ … ただし適切なりサイズやハッシュ関数でほぼ起こらない
- 空間計算量: $O(n)$ (テーブルサイズに応じたオーバーヘッドあり)

3.3 特徴・利点・欠点

- 利点
 - 平均 $O(1)$ で非常に高速
 - キー → 値 の対応に最適
- 欠点
 - メモリ使用量が増える
 - ハッシュ関数の設計 / 衝突解消の実装が必要
 - 順序付き探索には向かない

3.4 実装例 (C++ 標準ライブラリ利用)

```
#include <unordered_map>

// key -> value のマップを作成し、検索
int hashSearch(const std::unordered_map<int, std::string>& mp, int key) {
    auto it = mp.find(key);
```

```

if (it != mp.end()) {
    // 見つかったら値を参照できる
    return it->second;
}
// 見つからない
throw std::out_of_range("キーが存在しません");
}

```

3.5 適用場面

- キーとデータをペアで管理したい場合
- 頻繁に挿入・削除・検索を行う動的なデータ構造

4. 比較まとめ

特徴	線形探索	二分探索	ハッシュ探索
前提条件	なし	ソート済み	なし
平均時間計算量	$O(n)$	$O(\log n)$	$O(1)$
最悪時間計算量	$O(n)$	$O(\log n)$	$O(n)$
空間オーバーヘッド	ほぼゼロ	ほぼゼロ	テーブル分あり
実装の易しさ	◎ (簡単)	○ (やや注意)	△ (ハッシュ設計)
用途	小規模or単発探索	大規模かつ静的	動的なキー/値管理

5. 使い分けのポイント

- データが小さく、一度だけ検索する → 線形探索
- データはソート可能/ソート済みで、繰り返し検索する → 二分探索
- キーと値の対応を高速に管理したい/挿入・削除も行う → ハッシュ探索

もしそれぞれの詳しい応用例や、別の言語での実装例、あるいは木構造探索 (B木・AVL木など) についてもご興味があれば、ぜひお知らせください。

✓ あなた:

ぎりぎりC言語しか勉強していない人がわかる疑似言語表記で、それぞれのアルゴリズムを説明してほしい。

ChatGPT:

了解です!

C言語に慣れていない人がすんなり読めるように、「C言語風の疑似コード (変数宣言あり、for・if中心、関数呼び出しも明示)」で、以下3つの探索アルゴリズムを順に説明します。

☒ 1. 線形探索 (Linear Search)

● 概要

先頭から順番に調べていって、見つかったらその位置 (インデックス) を返す。

● 疑似コード (C風)

```

int linear_search(int arr[], int size, int key) {
    int i;

    for (i = 0; i < size; i++) {
        if (arr[i] == key) {
            return i; // 見つかったのでその位置を返す
        }
    }
}

```

```
return -1; //最後まで見つからなかった
}
```

● 説明ポイント

- `arr` … 探す対象の配列
- `size` … 配列の要素数
- `key` … 探したい値
- 見つかると `i` を返す、見つからないと `-1`

○ 2. 二分探索 (Binary Search)

● 概要

中央を調べて、半分ずつ範囲を狭めていく方法 (※配列がソートされている必要あり)

● 疑似コード (C風)

```
int binary_search(int arr[], int size, int key) {
    int left = 0;
    int right = size - 1;

    while (left <= right) {
        int mid = (left + right) / 2;

        if (arr[mid] == key) {
            return mid; // 見つかった
        } else if (arr[mid] < key) {
            left = mid + 1; // 右半分を探す
        } else {
            right = mid - 1; // 左半分を探す
        }
    }

    return -1; // 見つからない
}
```

● 説明ポイント

- `left` と `right` で調べる範囲を持っておく
- 中央 (`mid`) と `key` を比較
- `key` が大きければ右、小さければ左に範囲を狭める

☒ 3. ハッシュ探索 (Hash Search)

● 概要

値 (`key`) を使って、直接「探す場所 (インデックス)」を計算する方法
→ 計算式: `index = hash_function(key)`

※ここでは衝突がない簡易な例として説明します

● 疑似コード (C風: 簡易ハッシュテーブル使用)

```
#define TABLE_SIZE 10
```

```

int hash_function(int key) {
    return key % TABLE_SIZE;
}

int hash_table[TABLE_SIZE]; // 値を保存するテーブル

void insert(int key) {
    int index = hash_function(key);
    hash_table[index] = key; // *衝突処理は省略
}

int hash_search(int key) {
    int index = hash_function(key);
    if (hash_table[index] == key) {
        return index; // 見つかった
    } else {
        return -1; // 見つからない or 衝突で上書きされた
    }
}

```

● 説明ポイント

- `key % TABLE_SIZE` で場所を決める
- `insert()` で格納、`hash_search()` で検索
- ※衝突（同じ場所になること）を処理するには工夫が必要（例：線形探索など）

☒ 比較まとめ（覚えやすく）

種類	ソート必要	速さ（目安）	簡単さ	特徴
線形探索	不要	遅い $O(n)$	◎ 簡単	一番素直な方法
二分探索	必要	速い $O(\log n)$	○ 普通	中央から半分に絞っていく
ハッシュ探索	不要	速い $O(1)$	△ やや難	直接位置を計算して探す

☺Revision #1

★Created 16 July 2025 00:04:12 by youe2

✎Updated 2 June 2026 18:59:18 by youe2