C++プログラミングI

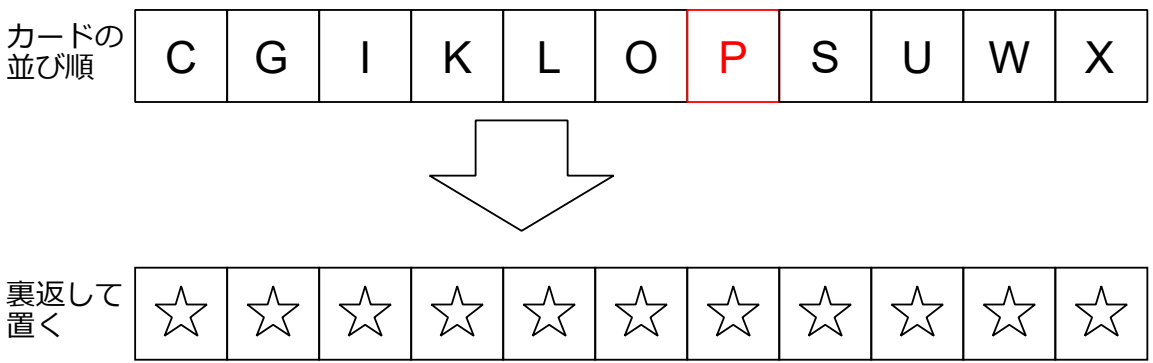
- 第12回：アルゴリズム
- 担当：二瓶芙巳雄

アルゴリズム

- アルゴリズムとは矛盾のない処理手順を示したもの。
- 細かなことより矛盾のないことが重要
- 情報処理の基本：
  1. 問題を分析してモデル化する(作る/当てはめる)
  2. モデルに合うアルゴリズムを探す（または考える)。
  3. プログラムを作る。
- 本日の内容
  - 逐次探索
  - 2分探索法
  - 選択ソート
  - 標準アルゴリズムの利用

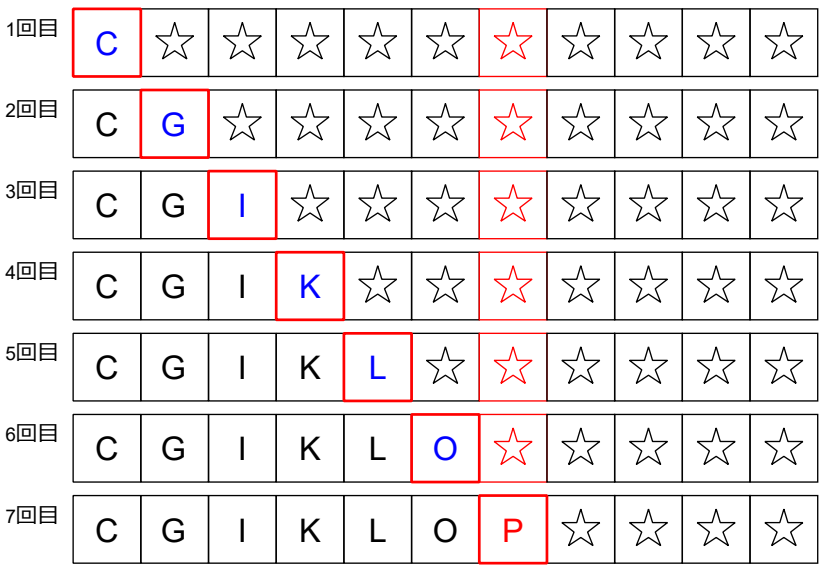
カード探索問題

- ある文字の書かれたカードを探したい
  - 例えば `P`
- 1回に1枚だけ調べることができる
- **ABC順に並んでいる**（※データはソート済み）



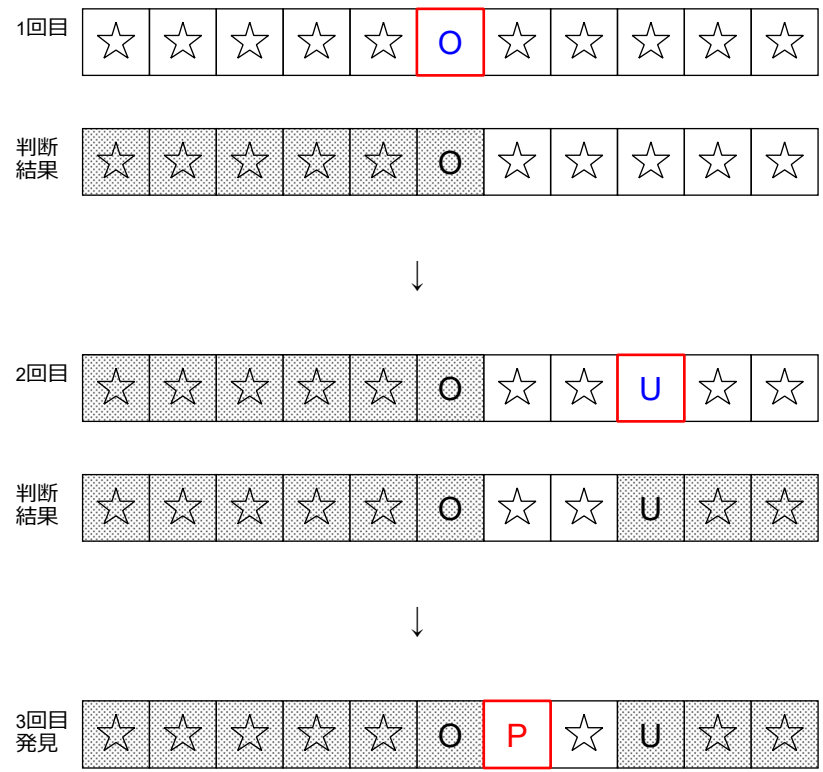
逐次探索

- 先頭から末尾に向かって一つ一つ順番にチェックする
  - n個のデータならば最悪n回調べることになる。



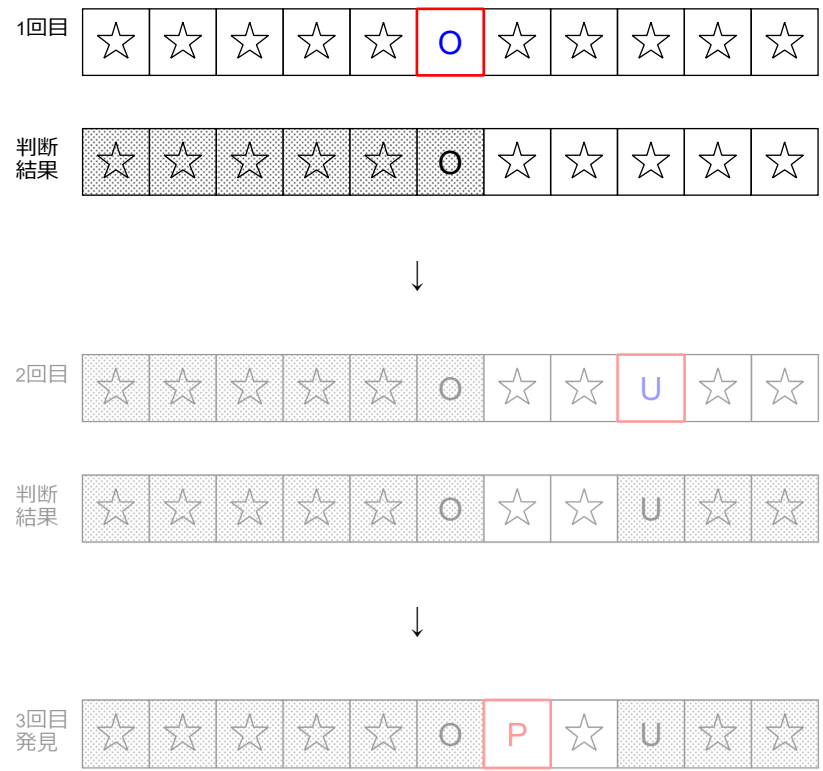
2分探索法

- データが**何らかの順番で並んでいることが分かっている場合に**効率良く探す方法
- 1回に1個のデータを調べる
- データがなければ探索範囲を半分にする
- 最良は逐次探索と同じ1回の調査
- 最悪でも $\log_2 n + 1$ 回だけ調べれば良い
  - ※最悪の場合の計算量が逐次探索より少ない
- 各ステップでの詳細は以降から
  - Pを探す場合の例



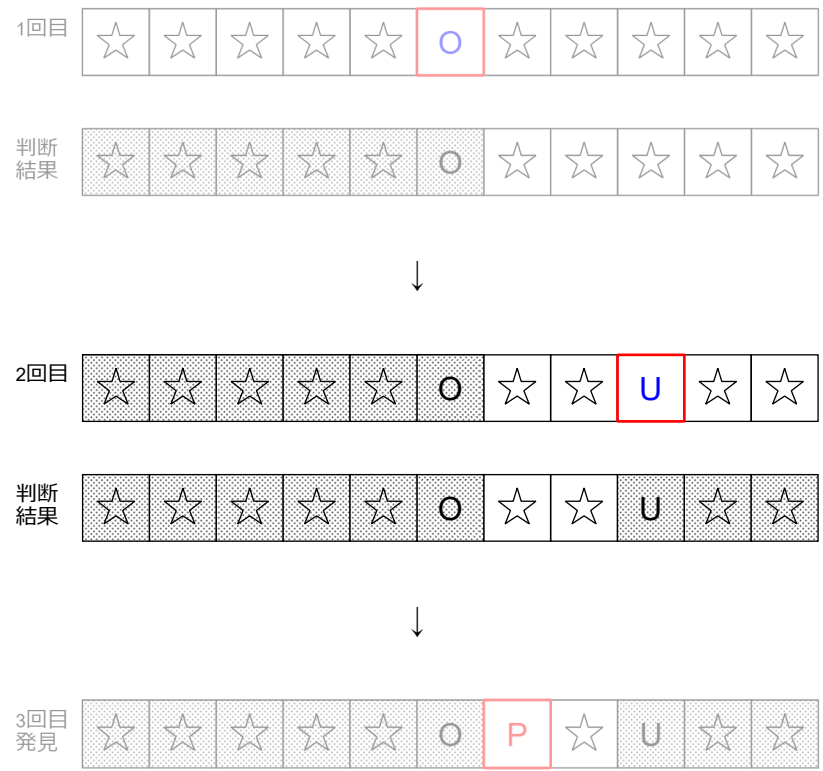
カードPを探す例: step 1

- 探索範囲 (1 … 11) の真ん中をチェックする
  - (1+11)/2 = 6番目
- 目的のカードではない
- カードOより左側には目的のPはない
  - カードはABC順に並んでいるため



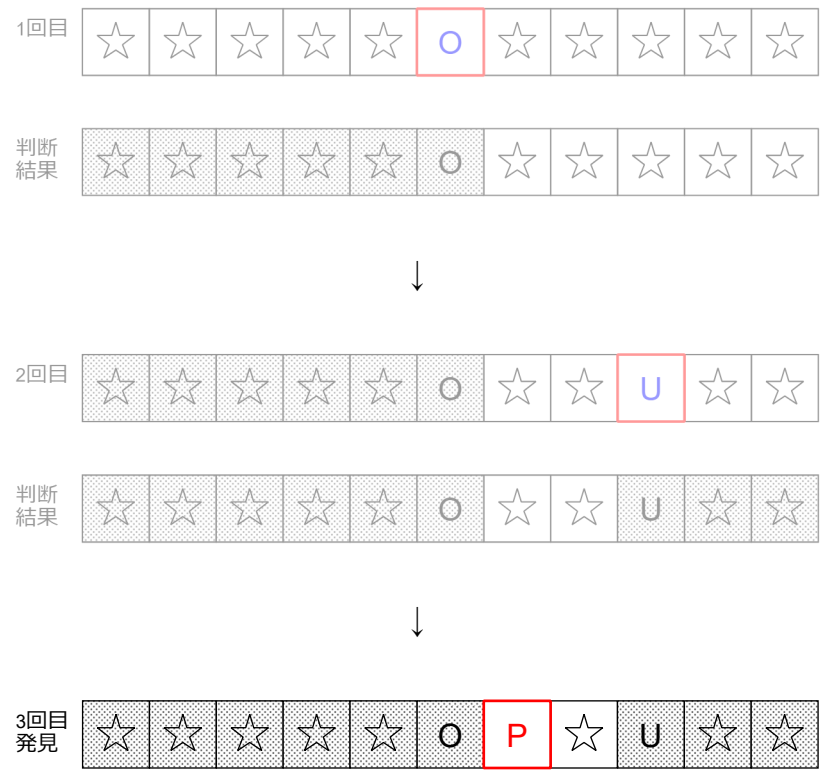
カードPを探す例: step 2

- 探索範囲 (7 … 11) の真ん中をチェックする
  - (7+11)/2 = 9番目
- 目的のカードではない
- カードUより右側には目的のPはない
  - カードはABC順に並んでいるため



カードPを探す例: step 3

- 探索範囲 (7 … 8) の真ん中をチェックする
  - 残り2枚でも同じ作業をする
  - 真ん中はないので (7+8)/2=7.5 から切り捨てて 7 とする
  - ※切り捨てでも切り上げでもどちらでもよいが，今回は切り捨て
- 目的のカードが見つかった。



ABCカードの2分探索法

- **入力**: ABC順に並べられた複数のカード、探したい文字
- **出力**: カード中の指定文字カードの有無
- **手順**:
  1. 探索範囲の先頭と末尾を指定するための順序数を $First$と$Last$とする。最初は、$First$は1であり、$Last$はカードの枚数の数。
  2. $First \leq Last$であるかぎり以下の手順を繰り返す。$First > Last$ となったら、対象のカードはない。
     1. $\lfloor \frac{First+Last}{2} \rfloor$を計算し$Middle$とする。
     2. 順序数が$Middle$であるカードを調べ、対象カードならば終了
     3. $First$または$Last$を次のように更新する。
        - $Middle$のカードが対象カードよりも ABC順で前ならば、$First$を$Middle + 1$とする
        - そうでなければ、$Last$を$Middle - 1$とする

上記のような問題を解決するための手順のまとめが**アルゴリズム**

注意点

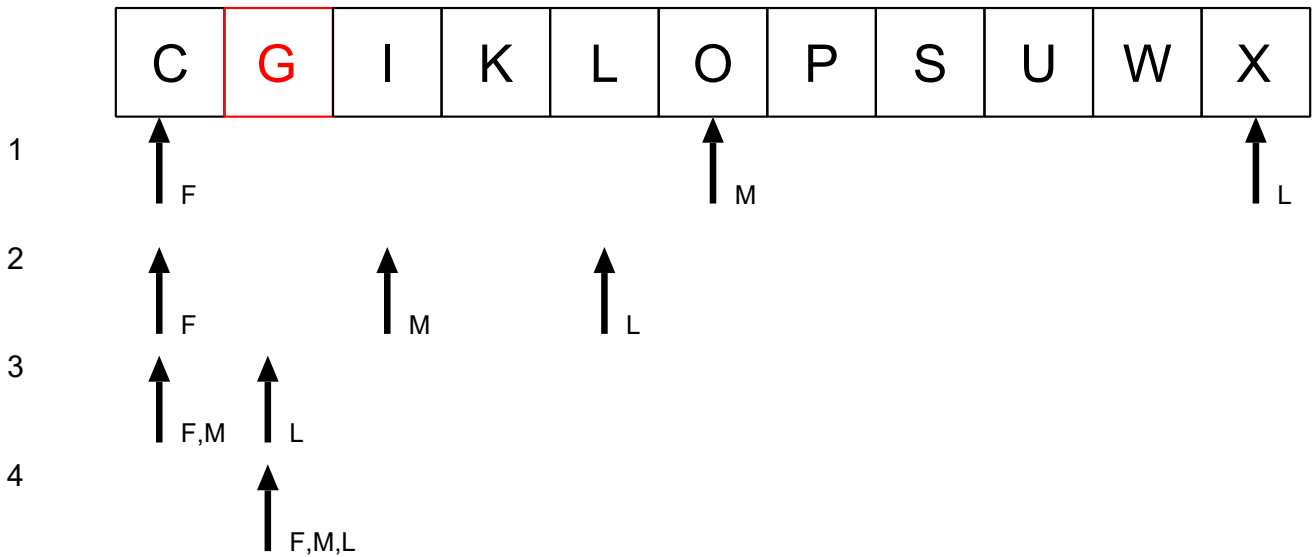
- $\lfloor x \rfloor$: 床関数:実数xに対してx以下の最大の整数
  - 小数点以下を切り捨てた値
- $\lceil x \rceil$:天井関数:実数xに対してx以上の最小の整数
- このアルゴリズムはABCの書かれたカード専用
- 何らかの順序があれば手順はカードに限らず適用可能

確認

- 例に挙げた11枚のカードを対象に, Gを探した場合, Kを探した場合, Wを探した場合をそれぞれ考え、例に習って各探索ステップを図示してみましょう。
- 例に挙げた11枚のカードを対象に, Aを探した場合, Jを探した場合, Qを探した場合, Zを探した場合をそれぞれ考え、例に習って各探索ステップを図示してみましょう。

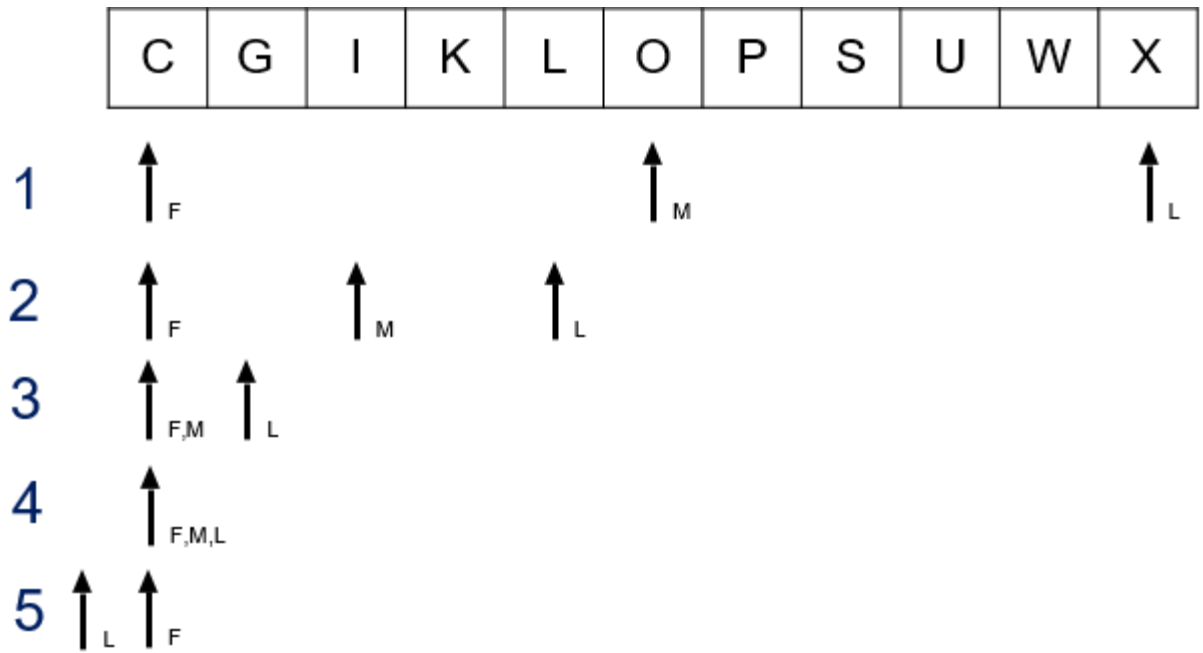
確認の例:Gを探す場合

- F, M, Lの変化を確認する
- 見つかる場合のパターン



確認の例:Aを探す場合

- 見つからない場合にどう終わるか?



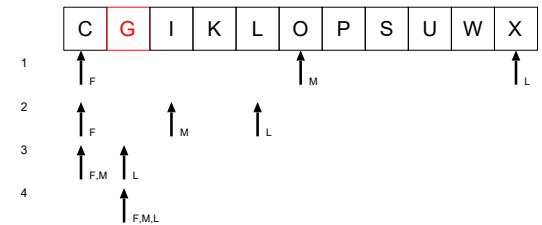
- **F > L となるため対象のカードはない**

C++による2分探索法

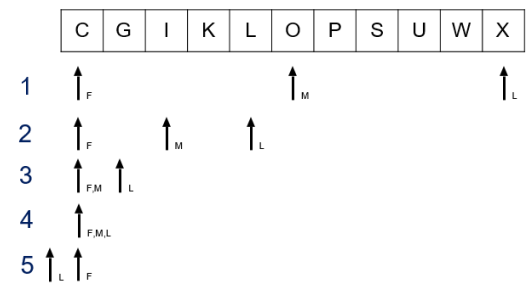
```cpp
bool binarySearch(const vector<string>& a, string x) {
  int first{0}, last{int(a.size() - 1)};

  while (first <= last) {
    int middle{(first + last) / 2};

    if (a[middle] == x) return true;
    if (a[middle] < x) first = middle + 1;
    else last = middle - 1;
  }
  return false;
}

int main() {
  vector<string> a { "C", "G", "I", "K", "L", "O", "P", "S", "U", "W", "X" };
  std::cout << "P is " << (binarySearch(a, "P") ? "" : "not ") << "found.\n";
  std::cout << "A is " << (binarySearch(a, "A") ? "" : "not ") << "found.\n";
}
```

```
% ./a.out
P is found.
A is not found.
```

- Gを探すとき



- Aを探すとき



C++による2分探索法：処理途中の値の確認

- 先ほどのプログラムの途中途中に `std::cout` を差し込む
- プログラミングが苦手な方は，処理途中の値を確認してみるとよいです

```
% ./a.out
P
f: 0, l: 10
f: 0, m: 5, l: 10, a[middle]: O
f: 6, m: 8, l: 10, a[middle]: U
f: 6, m: 6, l: 7, a[middle]: P
A
f: 0, l: 10
f: 0, m: 5, l: 10, a[middle]: O
f: 0, m: 2, l: 4, a[middle]: I
f: 0, m: 0, l: 1, a[middle]: C
```

```cpp
bool binarySearch(const vector<string>& a, string x) {
  int first{0}, last{int(a.size() - 1)};
  std::cout << "f: " << first << ", l: " << last << "\n";

  while (first <= last) {
    int middle{(first + last) / 2};
    std::cout << "f: " << first
              << ", m: " << middle
              << ", l: " << last
              << ", a[middle]: " << a[middle] << "\n";

    if (a[middle] == x) return true;
    if (a[middle] < x) first = middle + 1;
    else last = middle - 1;
  }
  return false;
}

int main() {
  vector<string> a { "C", "G", "I", "K", "L", "O", "P", "S", "U", "W", "X" };
  std::cout << "P\n" << binarySearch(a, "P") << "\n";
  std::cout << "A\n" << binarySearch(a, "A") << "\n";
}
```

C++コードの注意点

- C++では序数は0から
  - `first <-- 0, last <-- a.size()-1`
- `(first+last)/2` の計算
  - 整数の割り算は結果を切り捨て
- 初期化と型変換
  - `int first{0}, last {int(a.size()-1)};`
  - `int(x)` は `x` から `int` 型の値を指定する
  - `C++98` 形式なら指定不要だが変換が不明確
    - 後で問題が起きたときに探せない

```
1   int last = a.size() - 1; // 警告なし(C++98)
2   last = a.size() - 1;     // 警告なし（代入）
3   int last {a.size()-1};   // 警告あり(C++11)
```

配列の添字にはこれまで `size_t` 型を使う例が多いが、今回はint 型 → 負数を表現する必要があるため

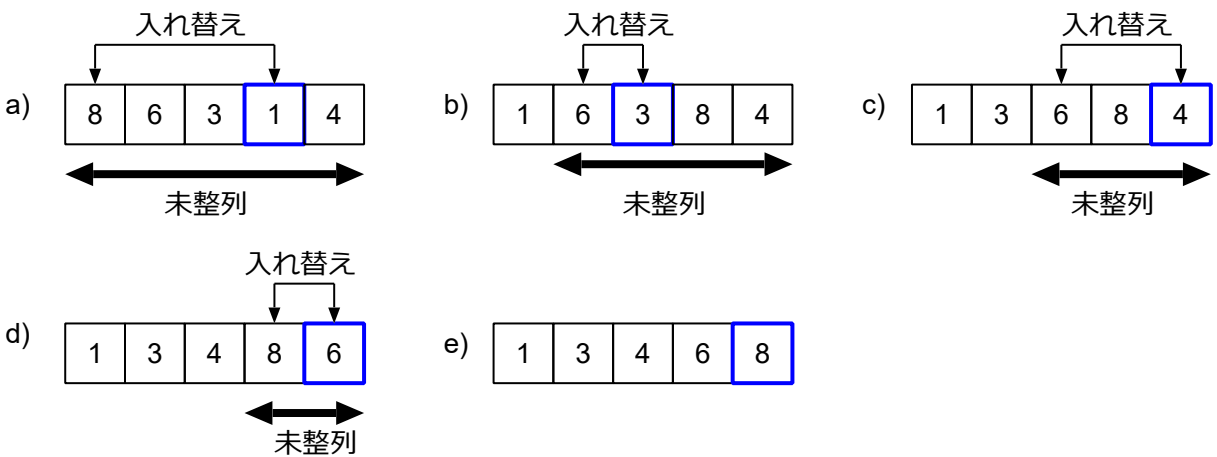
C++コードの注意点2

- `int`と`size_t`の違いで個数の制限
  - `int`型の正の最大値は$2^{31}-1$(21億)
  - `a.size()`の最大値は$2^{64}-1$(1845京)
- このコードは添字変数が`size_t`型だと動作しない
  - `first`が`0`ときは`last`が負数にならないと`while`文は終了しない
  - **符号なし整数`0`から1を引くと最大値になる**
- 以下が`size_t`を使う場合のコード
  - 配列の添字は`a.size()`未満
  - 無限ループを避けるため、条件を追加

```
1  size_t first{0}, last{a.size()-1};
2
3  while (first <= last && last < a.size()) {
4     auto middle {(first + last)/2};
5  }
```

選択ソートの概要

- ソートまたは整列：複数のデータをある基準に従って並べ替えること
- 選択ソートとは
  - 対象データ全体から1番目となるデータを選択して1番目と入れ替え、次に範囲をせばめて入れ替えることを繰り返す。

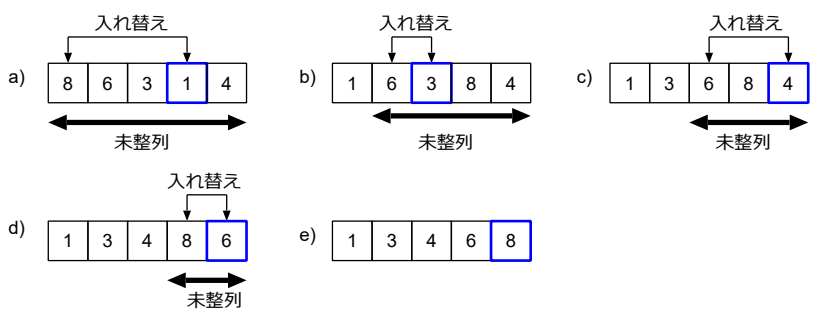


選択ソート

- **名前**: 数字の書かれたカードを昇順に並べ替える選択ソート
- **入力**: ランダムに並べられた数字の書かれたカード
- **出力**: 昇順に並べられた数字のカード
- **手順**:
  1. カードの置かれた場所を示す順序数を`i`とする。
  2. `i`を1からカードの合計枚数-1まで順に変化させて以下を繰り返す。
     1. `i`番目から最後のカードまで調べて最小値を持つカードを選択する。
     2. 選択したカードと現在`i`番目に置かれているカードを入れ換える。

C++による選択ソート

- 空配列を排除しないと無限ループとなる
- 添字変数 `i` を `0` から `a.size()-2` まで変化
- 内側の `for` 文のループ変数 `j` は `i+1`（`i` の右隣要素）から `a.size()-1`（末尾要素）に変化
- `<algorithm>` ヘッダファイルの `swap()` 関数




```cpp
void sort(std::vector<int>& a) {
  if (a.empty()) return;

  for (size_t i = 0; i < a.size()-1; i++) {
    size_t min { i }; // 最小値のindexを保存

    for (size_t j = i+1; j < a.size(); j++)
      if (a[j] < a[min]) min = j;

    std::swap(a[i], a[min]);
  }
}

int main() {
  std::vector data{8, 6, 3, 1, 4};
  sort(data);

  for (auto x : data) std::cout << x << " ";
}
```

```
% ./a.out
1 3 4 6 8
```

内側のループは最小値を探す

```
1   void sort(std::vector<int>& a) {
2     if (a.empty()) return;
3
4     for (size_t i = 0; i < a.size()-1; i++) {
5
6       size_t min { i }; // 最小値のindexを保存
7       for (size_t j = i+1; j < a.size(); j++)
8         if (a[j] < a[min]) min = j;
9
10      std::swap(a[i], a[min]);
11    }
12  }
```

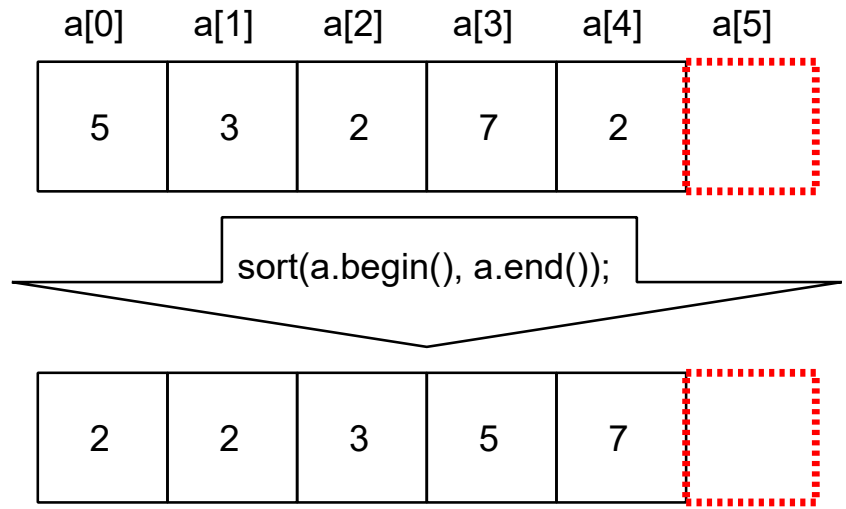
- 1p前のコード

```
1   size_t find_min(const vector<int>& a, size_t i)  {
2      size_t min { i };
3      for (size_t j = i+1; j < a.size(); j++)
4         if (a[j] < a[min]) min = j;
5      return min;
6   }
7
8   void sort(vector<int>& a)  {
9      if (a.empty()) return;
10
11     for (size_t i = 0; i < a.size()-1; i++) {
12        size_t min { find_min(a, i) };
13        std::swap(a[i], a[min]);
14     }
15  }
```

- 左のコードを書き換えたもの

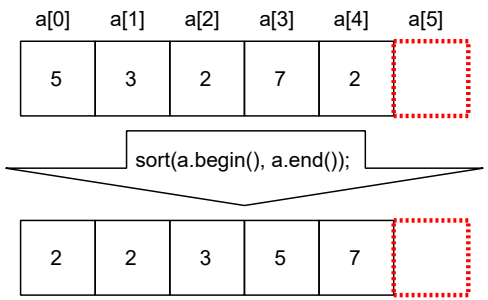
標準アルゴリズムの利用

- `<algorithm>` ヘッダファイルのライブラリ関数
- 大半が関数テンプレート
- 例： `sort()`, `binary_search()`, `find()`
- 配列の範囲指定
  - イテレータ：反復用の特殊な値（※授業範囲外）
  - `begin()`：「先頭の要素」
  - `end()`：「末尾要素の**次**」（図中の赤い部分）



標準アルゴリズムの `sort()`

- `sort()` の範囲指定: `begin()` と `end()`
- 範囲 `for` 文の範囲指定も `begin()` / `end()` が関係
- ※標準アルゴリズムの使い方は，お作法として覚えてください




```cpp
#include <algorithm> // std::sort()用
#include <iostream>
#include <vector>

int main() {
  std::vector a{5, 3, 2, 7, 2};
  std::sort(a.begin(), a.end());

  for (auto x : a) std::cout << x << ' ';
  std::cout << '\n';
  return 0;
}
```

```
% ./a.out
2 2 3 5 7
```

`sort()` と自作2分探索

- 自作2分探索は `begin()` / `end()` は指定しない
- このプログラムの `sort()` は関数テンプレートであるために、今回は `vector<string>` 型の配列を扱っている

```
% ./a.out
Z G W K Y T P S C I X
C G I K P S T W X Y Z
X
X is found.
Z
Z is found.
A
A is not found.
```

```cpp
// 前に作成したもの
bool binarySearch(const vector<string>& a, string x);

void print_vec( const vector<string>& v ) {
  for (auto x : v) std::cout << x << ' ';
  std::cout << '\n';
}

int main() {
  vector<string> a{
    "Z", "G", "W", "K", "Y",
    "T", "P", "S", "C", "I", "X"
  };
  print_vec( a );

  std::sort( a.begin(), a.end() );
  print_vec( a );

  for (string x; std::cin >> x;) {
    std::cout << x << " is "
              << (binarySearch(a, x) ? "" : "not ")
              << "found.\n";
  }
}
```

構造体配列のソート

- `<` 演算子があれば `sort()` が使用できる
  → その構造体用の `operator<()` を定義
- `<<` 演算子もあれば `main()` は組み込み型と変更なしで使いまわせる

```
% ./a.out
(Ari,0,0)
(Leo,1,0)
(Cnc,2,1)
(Gem,2,2)
(Vir,2,4)
(Sgr,3,0)
(Tau,3,1)
(Aqr,3,5)
(Sco,4,2)
(Psc,4,3)
(Cap,5,2)
(Lib,5,8)
```

```cpp
#include <algorithm>

struct Point { std::string name; int x, y; };

// 座標値で順序づけを行う ※後で詳細説明します
bool operator<(const Point& a, const Point& b) {
  return a.x < b.x || (a.x == b.x && a.y < b.y);
}

// 出力に対応させる
ostream& operator<<(ostream& out, const Point& a) {
  return out << "(" << a.name << "," << a.x << "," << a.y << ")";
}

int main() {
  vector<Point> vp{
    {"Ari", 0, 0}, {"Tau", 3, 1}, {"Gem", 2, 2}, {"Cnc", 2, 1},
    {"Leo", 1, 0}, {"Vir", 2, 4}, {"Lib", 5, 8}, {"Sco", 4, 2},
    {"Sgr", 3, 0}, {"Cap", 5, 2}, {"Aqr", 3, 5}, {"Psc", 4, 3}
  };

  std::sort(vp.begin(), vp.end());

  for (const auto& p : vp) std::cout << p << "\n";
}
```

`operator<()` の解釈

- 上：処理を1行に記述（ワンライナー）
  - 利点）簡潔，慣れている人にはわかりやすい
  - 欠点）読みにくい，メンテが困難
- 下：処理を複数行で記述
  - 利点）分岐が直感的（可読性が高い）
  - 欠点）行数がやや多い
- 昔の人の「短いコード＝良いコード」という信仰
- 最近は「可読性・保守性 ＞ 行数の少なさ」
  - チーム開発では可読性の高さが好まれる

```cpp
#include <algorithm>

struct Point { std::string name; int x, y; };

// // もともとの実装
// bool operator<(const Point& a, const Point& b) {
//   return a.x < b.x || (a.x == b.x && a.y < b.y);
// }

// 以下のように書き直しができる
bool operator<(const Point& a, const Point& b) {
  // x 座標で比較する
  if (a.x < b.x) return true;  // true -> (a<b) である
  if (a.x > b.x) return false; // false -> (a<b) でない

  // 以下が実行されるのは a.x == b.x のとき
  return a.y < b.y;
}

...以下略
```

標準アルゴリズムの `binary_search()`

- `std::binary_search()` は `bool` 値を返す
  - 第1引数と第2引数で，探索範囲を指定
  - 第3引数を探す
- `<` 演算子で比較を行う
  - `==` と `!(a<b) && !(b<a)` は等価
  - `operator<()` を定義する（既に定義されている）場合，どの型の配列でも使用できる

```cpp
#include <algorithm>

int main() {
  vector<string> a{
    "Z", "G", "W", "K", "Y",
    "T", "P", "S", "C", "I", "X"
  };
  // 標準アルゴリズムでソート
  std::sort( a.begin(), a.end() );

  for (string x; std::cin >> x;) {
    // 標準アルゴリズムで二分探索
    bool b{ std::binary_search(a.begin(), a.end(), x) };
    std::cout << x << " is "
              << (b ? "" : "not ")
              << "found.\n";
  }
}
```

```
% ./a.out
X
X is found.
Z
Z is found.
A
A is not found.
```

標準アルゴリズムの `find()`

- 逐次探索を行う
- 見つかると対応する配列の要素のイテレータを返す
  - ※イテレータについては授業範囲外なので，よくわからなくてOK
- 見つからない場合には第2引数の値を返す
- 変数が必要ならば `auto` を使って宣言

```cpp
#include <algorithm>

int main() {
  vector<string> a{
    "Z", "G", "W", "K", "Y",
    "T", "P", "S", "C", "I", "X"
  };

  for (string x; std::cin >> x;) {
    // 標準アルゴリズムで逐次探索
    auto it{ std::find(a.begin(), a.end(), x) };
    cout << x << " is "
         << (it != a.end() ? "" : "not ")
         << "found.\n";
  }
}
```

```
% ./a.out
X
X is found.
Z
Z is found.
A
A is not found.
```

`binary_search()` と `find()` の選択

- `binary_search()` は `find()` より探索時間が短い
- `binary_search()` はデータが整列が必要
  - 整列には時間がかかる
- アルゴリズム選択の基準
  - データを入力した時点でそれらが整列しているのか？
  - 探索を何度も行う必要があるか？
  - データを並びかえてしまって問題がないか？