大阪電気通信大学 情報通信工学部 光システム工学科 2年次配当科目

コンピュータアルゴリズム

整列アルゴリズムの復習

第6講: 平成20年11月14日 (金) 4限 E252教室

中村 嘉隆(なかむら よしたか)
奈良先端科学技術大学院大学 助教
y-nakamr@is.naist.jp
http://narayama.naist.jp/~y-nakamr/

第 4 講の復習

- 整列アルゴリズム
  - ソーティング，並べ替え
  - $O(n^2)$ のアルゴリズム
    - 選択ソート
      - 最小値を探して前から並べる
    - バブルソート
      - 隣の要素の大小関係で交換していく
    - 挿入法
      - 前から順番に入るべき位置に入れていく

第 5 講の復習

- 整列アルゴリズム
  - $O(n \log n)$ のアルゴリズム
    - マージソート
      - 2 つ，4 つ，8 つと整列する列を併合(マージ)していく
    - クイックソート
      - 基準値(ピボット)を選んで，それより小さい数値の列と大きい数値の列に分けていく
      - 分割統治法

今日の講義内容

- 整列アルゴリズムの演習
- オーダ記法の復習

整列(ソーティング)問題とは

- ソーティング: Sorting，整列，並べ替え
  - n 個のデータ列をある基準に従って順に並べ替える処理
- 昇順ソート(Ascending Sort)
  - 単調増加に整列(小さいもの順に整列)
  - 一般的にソートといえばこちらを指す
- 降順ソート(Descending Sort)
  - 単調減少に整列(大きいもの順に整列)
- 昇順と降順は比較に用いる不等号を逆にする
- ソーティングにおける時間計算量は比較の回数を基準として考える
  - if 文を用いた大小比較

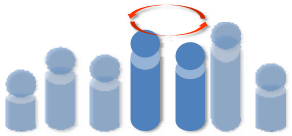
整列問題の分類

- 安定性
  - 安定ソート
    - ソートの際，同等なデータには元の並び順が保存されているソート法
      例) 元々学籍番号順に並んでいたデータをその成績順にソートしたとき，同じ点数の生徒は学籍番号順に並んでいるようなソート法
- 記憶領域
  - 外部ソート
    - ソートの際，定数個より多くの外部記憶領域を必要とするソート法
      例) 現在操作中の配列の他に，その長さに応じた別のデータ格納用の配列が必要なソート法
  - 内部ソート
    - ソートの際，定数個の記憶領域で十分なソート法
      例)現在操作中の配列の内部の入れ替えだけで十分なソート法

準備

- 入力は長さ n の数値の列
  - $\{a1, a2, a3, a4, \ldots, an\}$ で表す
- 数値の大小関係には推移律が成り立つ
  - $a < b$ で $b < c$ なら $a < c$
- Swap 手続き
  - 配列中の 2 つの要素の値を入れ替える手続き
  - 実際には以下のようにテンポラリ(一時的)の変数を準備して入れ替えする

```
swap (a, b) {
    temp ← a ;
    a ← b ;
    b ← temp ;
}
```

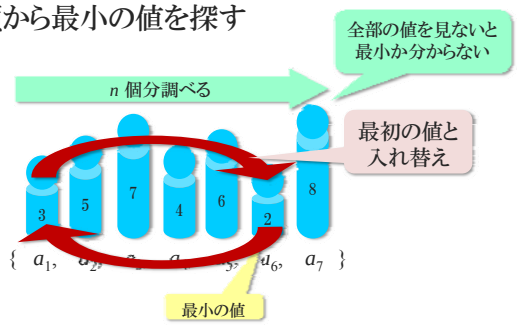


選択ソート: 概要

- 最小値選択法: Selection Sort
- 直接選択法: Straight Selection
- アルゴリズム
  1. 未整列部分から最小値を選択
  2. 未整列部分の先頭に置く
  3. 以上を未整列部分がなくなるまで繰り返す

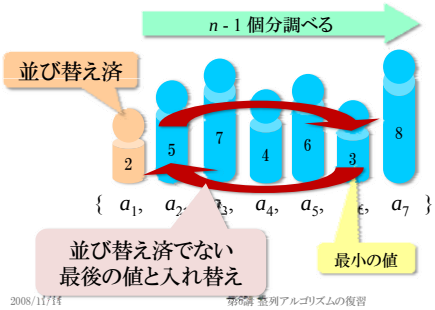
選択ソート: 概念図

- 先頭から最小の値を探す



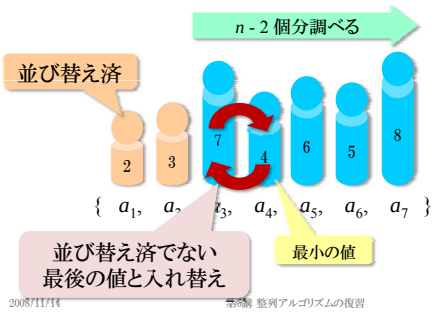
選択ソート: 概念図

- 並び替え済みでないもので繰り返す



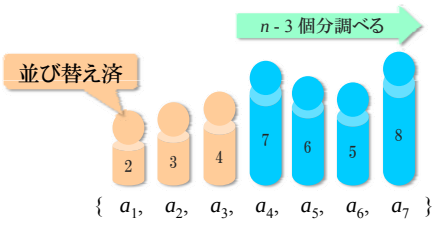
選択ソート: 概念図

- 並び替え済みでないもので繰り返す



選択ソート: 概念図

- 並び替え済みでないもので繰り返す……



選択ソートのまとめ

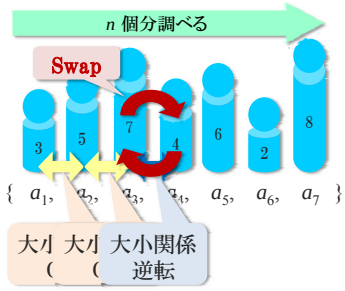
- 計算量 $O(n^2)$
  - 最良も最悪も $O(n^2)$
- 安定ソートではない
  - 最小値と先頭値を交換するので元の順番が崩れる
- 内部ソート
  - 最初の配列以外の記憶領域を使わない
- 最悪計算時間が $O(n^2)$ と遅いが，アルゴリズムが単純で実装が容易なため，しばしば用いられる

バブルソート：概要

- バブルソート：Bubble Sort
- アルゴリズム
  1. 隣接する 2 要素を比較
  2. 右側（要素番号の大きい側）の値が大きくなるように，比較結果によって値を交換
  3. 以上の操作をデータ列の左側（要素番号の小さい側）から右側へ向けて繰り返す

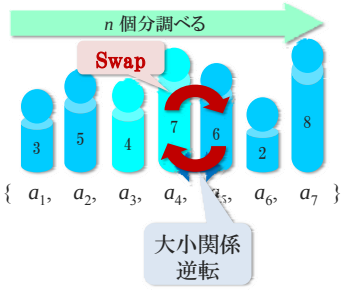
バブルソート：概念図

- 先頭から隣り合う 2 つずつを比べていく



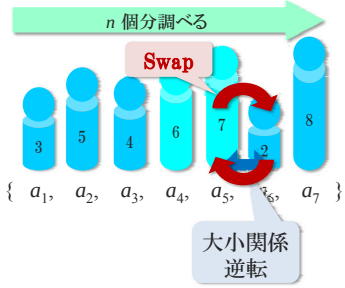
バブルソート：概念図

- 先頭から隣り合う 2 つずつを比べていく



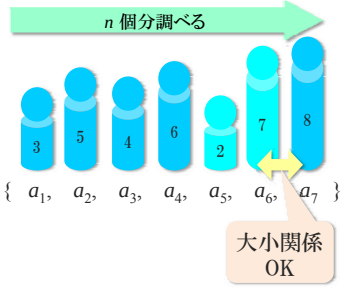
バブルソート：概念図

- 先頭から隣り合う 2 つずつを比べていく



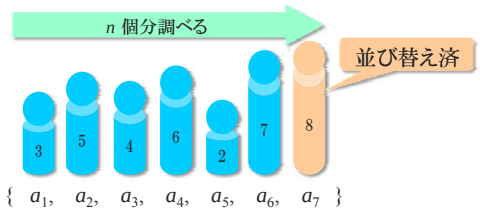
バブルソート：概念図

- 先頭から隣り合う 2 つずつを比べていく



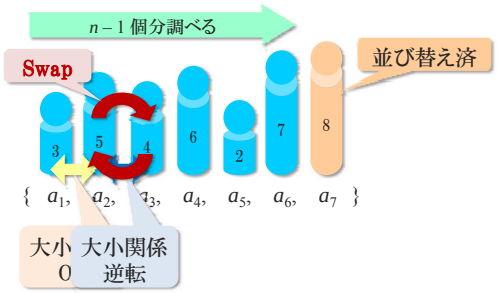
バブルソート：概念図

- 先頭から隣り合う 2 つずつを比べていく



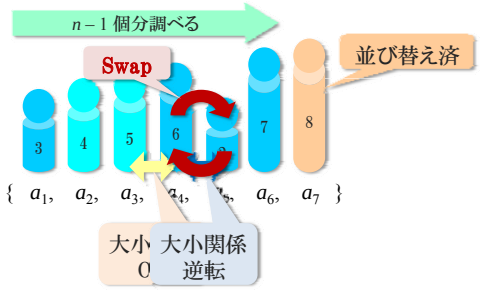
バブルソート：概念図

- 並べ替え済みでない部分で繰り返す



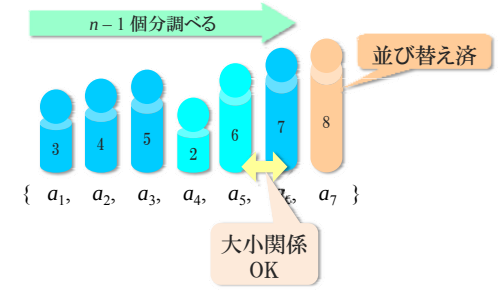
バブルソート：概念図

- 並べ替え済みでない部分で繰り返す



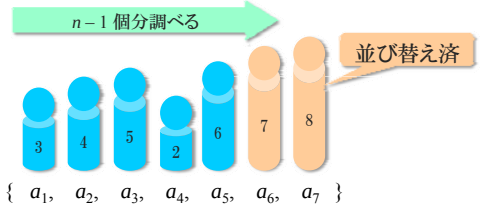
バブルソート：概念図

- 並べ替え済みでない部分で繰り返す



バブルソート：概念図

- 並べ替え済みでない部分で繰り返す

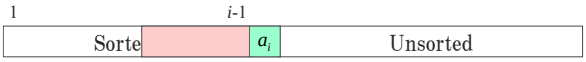


バブルソートのまとめ

- 計算量 $O(n^2)$
  - 最良は入力列がソートされているときで，1 回目の繰り返しで打ち切れば $O(n)$
  - 平均・最悪はもちろん $O(n^2)$
- 安定ソート
  - データの値が同じ場合，元の順番が保存される
- 内部ソート
  - 最初の配列以外の記憶領域を使わない
- 選択ソートと同じく最悪計算時間が $O(n^2)$ と遅いが，アルゴリズムが単純で実装が容易な上，安定ソートであるのでしばしば用いられる

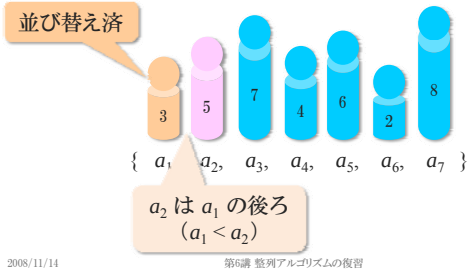
挿入法：概要

- 挿入法：Insertion Sort
- アルゴリズム
  1. a_1 から a_{i-1} までがソート済みと仮定
  2. a_i をしかるべき位置に挿入
     - 先頭から探す
  3. 以上を i を増加させつつ繰り返す



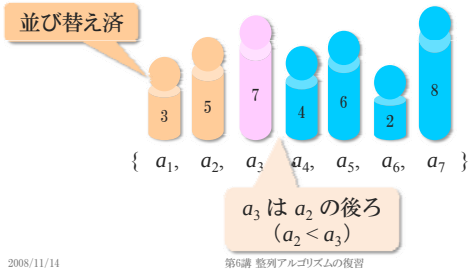
挿入法：概念図

- a_1 はソート済みとする（当たり前）
- a_2 が a_1 のどこに入るか調べる



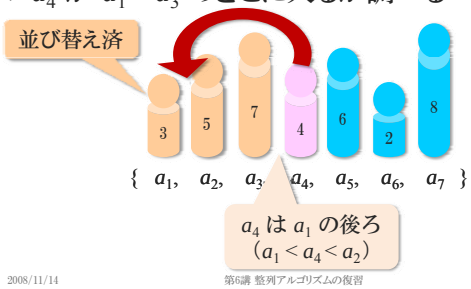
挿入法：概念図

- $a_1 \sim a_2$ はソート済み
- a_3 が $a_1 \sim a_2$ のどこに入るか調べる



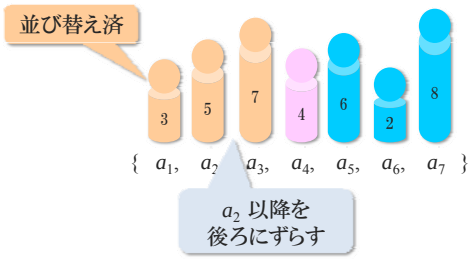
挿入法：概念図

- $a_1 \sim a_3$ はソート済み
- a_4 が $a_1 \sim a_3$ のどこに入るか調べる



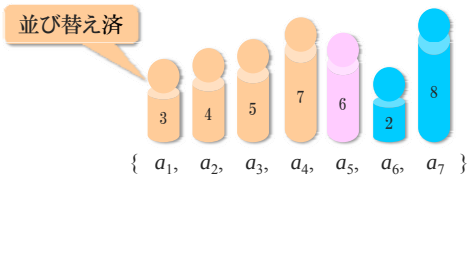
挿入法：概念図

- $a_1 \sim a_3$ はソート済み
- a_4 が $a_1 \sim a_3$ のどこに入るか調べる



挿入法：概念図

- $a_1 \sim a_4$ はソート済み
- a_5 が $a_1 \sim a_4$ のどこに入るか調べる……



挿入法のまとめ

- 計算量 $O(n^2)$
  - 最良は入力列がソートされているときで $O(n)$
  - 最悪は入力例が逆順にソートされているときで $O(n^2)$
  - 平均も $O(n^2)$
- 安定ソート
  - データの値が同じ場合，元の順番が保存される
- 内部ソート
  - 最初の配列以外の記憶領域を使わない

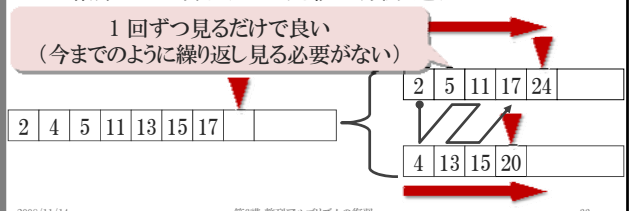
- 前述のソートと同じく最悪計算時間が $O(n^2)$ と遅いが，アルゴリズムが単純で実装が容易な上，安定ソートであるのでしばしば用いられる

マージソート：概要

- 単純マージソート：Straight Merge Sort
- 手順
  1. 初期状態の配列の要素を長さ 1 の列とする
  2. これらの n 本の列から 2 本ずつ組み合わせてマージし，長さ 2 の $n/2$ 本の列を得る
  3. 以下同様に長さを 2, 4, … と倍に増やし，全データが 1 本の列になれば終了

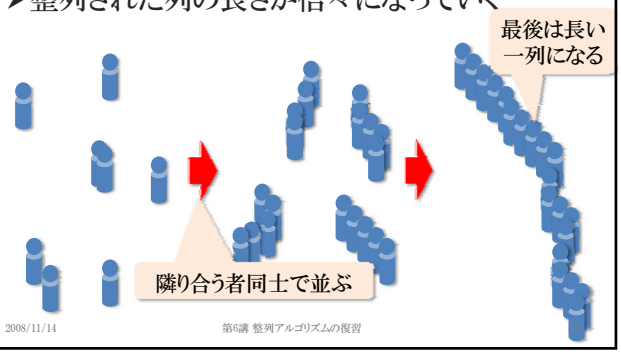
マージソート：概念

- マージとは
  - 整列された 2 本(3 本以上も可)を合わせて 1 本にする操作
  - 結果として得られる列も値の順序通りに並ぶ



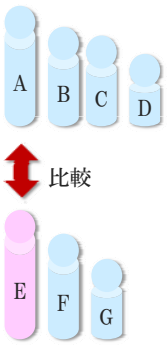
マージソート：概念図

- 整列された列の長さが倍々になっていく



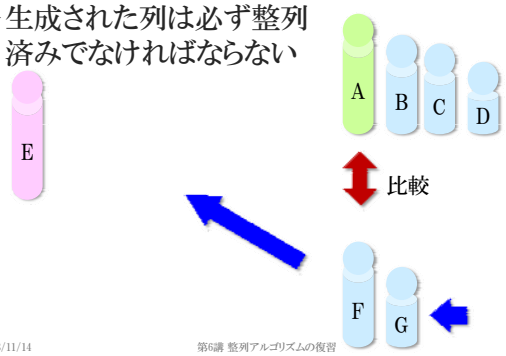
マージソート：列の生成

- 生成された列は必ず整列済みでなければならない



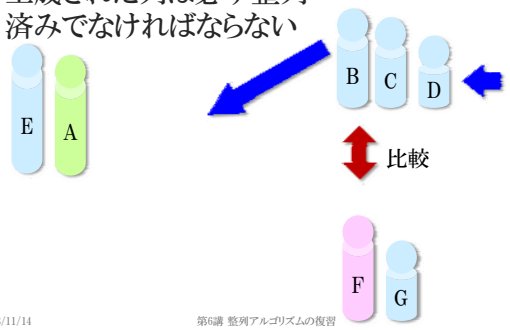
マージソート：列の生成

- 生成された列は必ず整列済みでなければならない



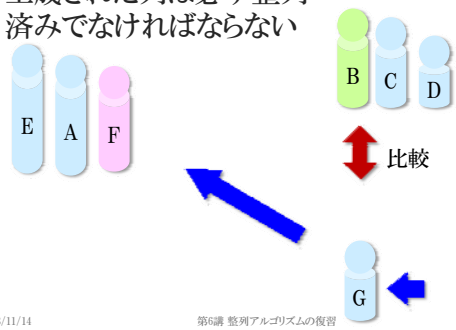
マージソート：列の生成

- 生成された列は必ず整列済みでなければならない



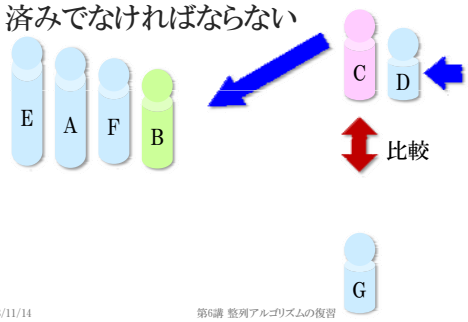
マージソート：列の生成

- 生成された列は必ず整列済みでなければならない



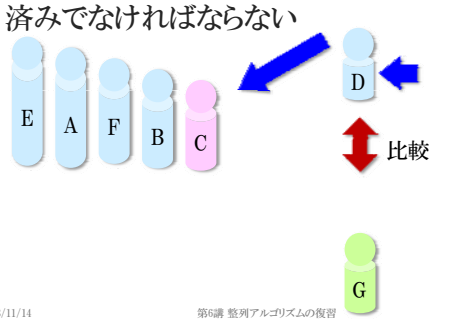
マージソート：列の生成

- 生成された列は必ず整列済みでなければならない



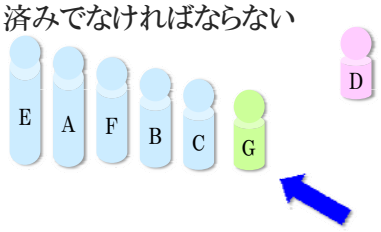
マージソート：列の生成

- 生成された列は必ず整列済みでなければならない



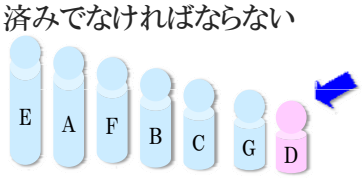
マージソート：列の生成

- 生成された列は必ず整列済みでなければならない



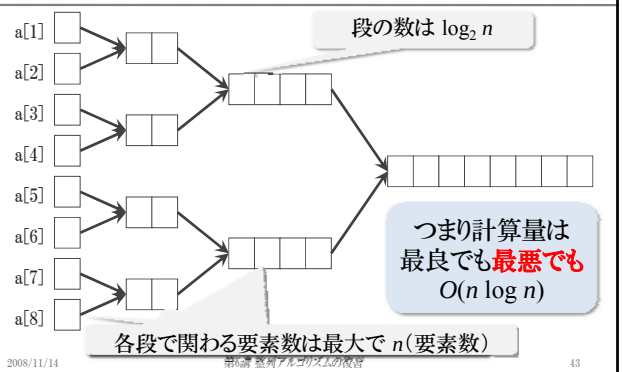
マージソート：列の生成

- 生成された列は必ず整列済みでなければならない



生成完了！

マージソート：列の生成



マージソートのまとめ

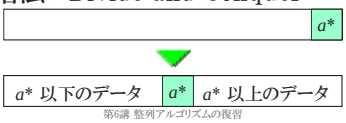
- 計算量 $O(n \log n)$
  - 最悪でも $O(n \log n)$ になる
- 安定ソート
  - データの値が同じ場合，元の順番が保存される
- 外部ソート（内部ソートではない）
  - 外部記憶に最初の配列と同じ長さ（$O(n)$）の記憶領域が必要
- 最悪でも計算時間が $O(n \log n)$ と早いが，外部ソートであるため，実際はあまり使われることがない
- 最悪時の計算量の上限を保証したいときは使う

クイックソート

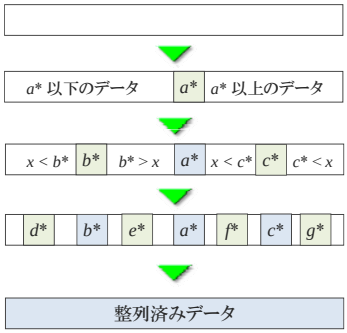
- クイックソート：Quick Sort
- Charles A. R. Hoare が考案
- 内部ソートでは最も速いアルゴリズム
- 平均計算量：$O(n \log n)$

クイックソート：概要

- アルゴリズム
  1. 基準値 a^* を選ぶ
  2. 基準値 a^* 以下のデータを左側，残りを右側に分割
  3. 分割された 2 つの部分に同様の操作を，分割部分の要素数が 1 になるまで繰り返す
- 再帰アルゴリズムで実装（しない方法もある）
- 分割統治法：Divide and Conquer



クイックソート：概念図

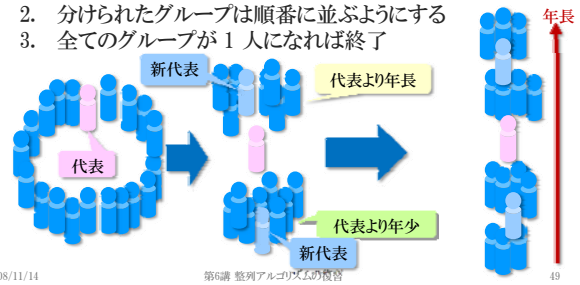


例題：子供の並べ替え

- 子供を生まれた日の順に並べ替えたい
  - 前提：同じ誕生日の子はいないとする
  - 全体法（選択ソートのような方法）
    - 全員で輪になって誕生日を言い，一番年長の子から順に抜けていく
    - 残ったメンバーでまた誕生日を言い，続けていく
    - そうすると最終的には誕生日の順に並ぶ
  - クイック法（ここで提案したい方法）
    - 代表の 1 人が自分の誕生日を言い，それより先に生まれた子とあとに生まれた子にグループ分けする
    - グループで大きい順に並ぶ
    - グループの人数が 1 人になるまで繰り返すと，最終的に誕生日の順に並ぶ

例題：クイック法

- グループの 1 人が誕生日を言い，それより早く生まれた子と遅く生まれた子のグループに分かれる
  1. 最初のグループは全員
  2. 分けられたグループは順番に並ぶようにする
  3. 全てのグループが 1 人になれば終了



クイックソート：プログラム

- アルゴリズム概略

```
quick_sort( left , right ) {
        if ( left < right ) {
            a* ← a[right] ;
            i ← partition( a* , left ,
    right );
            quick_sort( left , i - 1 ) ;
            quick_sort( i + 1 , right ) ;
        }
    }
```

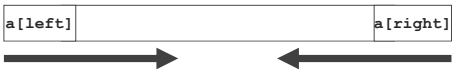
ここでは基準値を a[right] とする

- partition (a*, left , right)
  - 基準値(ピボット; pivot) a* によってデータを 2 つの部分に分割し，基準値の挿入されるべき位置 i を返す関数

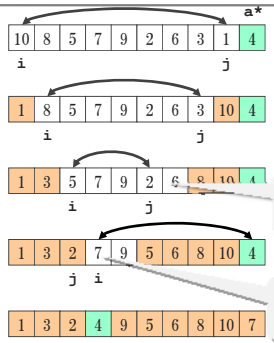
クイックソート：partition 手続き

- 基準値(ピボット)によってデータを 2 つの部分に分割
- partition(a* , left , right)

- 手順
  - 基準値を a[right] とする(簡単のため)
  - 左から走査し基準値より大きい要素を探索：位置 i
  - 右から走査し基準値より小さい要素を探索：位置 j
  - 両者の位置関係が i < j ならば交換
  - i ≧ j となるまで繰り返す



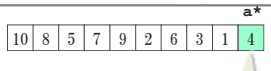
クイックソート：partition 手続き



a* < 6 なので飛ばす

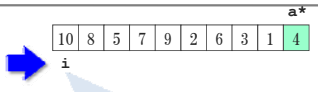
どんどん飛ばして最後は a* と交換

クイックソート：partition 手続き



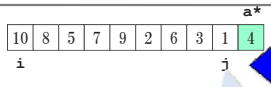
右端の値 a[right] を a* として採用

クイックソート：partition 手続き



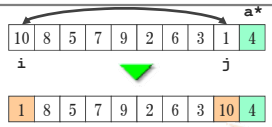
a[i] > a* なる i を探索
たまたま 1 つ目で当たる

クイックソート: partition 手続き



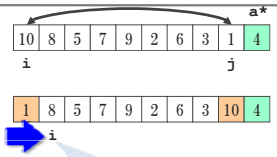
a[j] < a* なる j を探索
またもや 1 つ目で当たる

クイックソート: partition 手続き



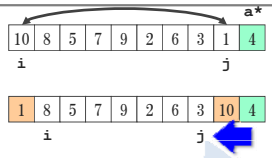
というわけで交換

クイックソート: partition 手続き



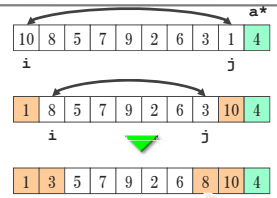
a[i] > a* なる i を探索
次から探していることに注意

クイックソート: partition 手続き



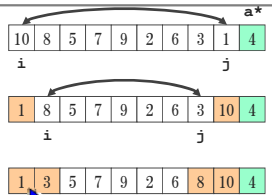
a[j] < a* なる j を探索
こちらも同様

クイックソート: partition 手続き



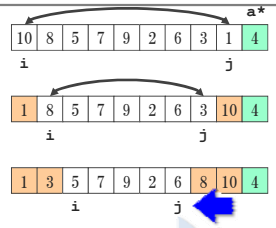
見つかったので交換

クイックソート: partition 手続き



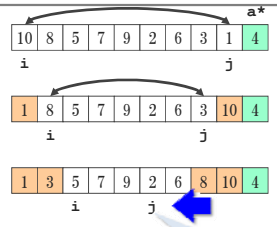
a[i] > a* なる i を探索

クイックソート: partition 手続き



a[j] < a* なる j を探索
これは条件に当てはまらない

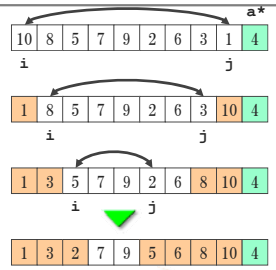
クイックソート: partition 手続き



ただし終了条件に注意

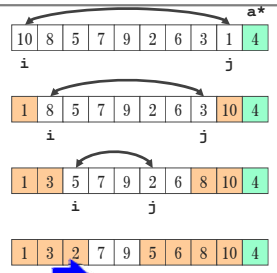
見つかるまでずらす
ここで見つかった

クイックソート: partition 手続き



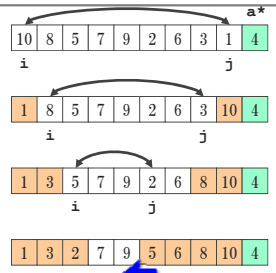
交換

クイックソート: partition 手続き



a[i] > a* なる i を探索

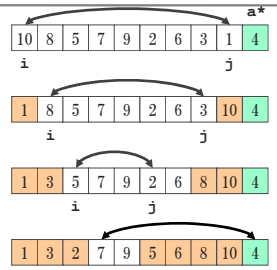
クイックソート: partition 手続き



最後まで見つから
ないこともある！

a[j] < a* なる j を探索
見つかったけれど j < i

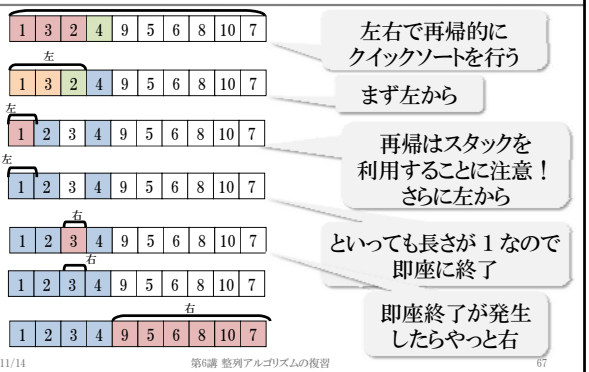
クイックソート: partition 手続き



分割完了！

最後なので
a* と交換

クイックソート



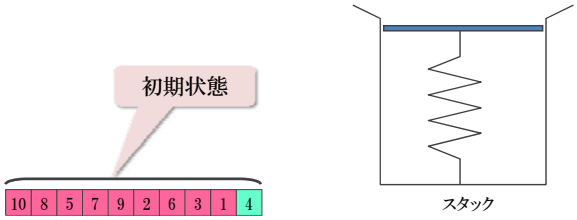
クイックソート

- こんな順番になる理由： 再帰アルゴリズム
  - 呼び出し元のデータを LIFO（Last In First Out）であるスタックを利用して保存している
- プログラム

```
quick_sort( left , right ) {
    if ( left < right ) {
        a* ← a[right] ;
        i ← partition( a* , left , right );
        quick_sort( left , i - 1 ) ;
        quick_sort( i + 1 , right ) ;
    }
}
```

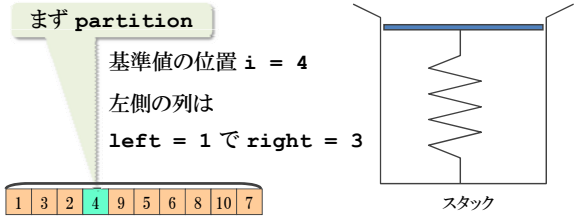

クイックソート：スタックの利用

- 例
  - quick_sort(1, 10)



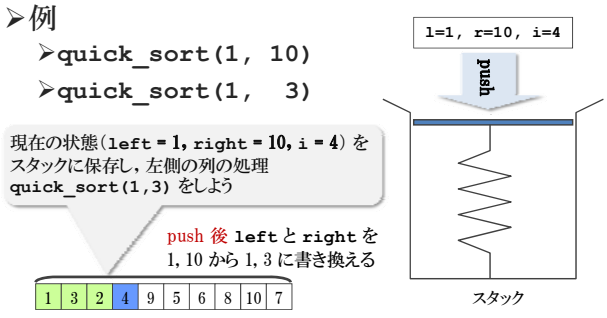
クイックソート：スタックの利用

- 例
  - quick_sort(1, 10)



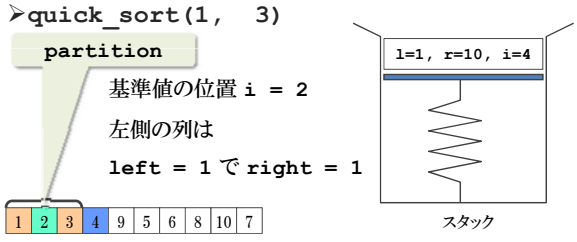
クイックソート：スタックの利用

- 例
  - quick_sort(1, 10)
  - quick_sort(1, 3)



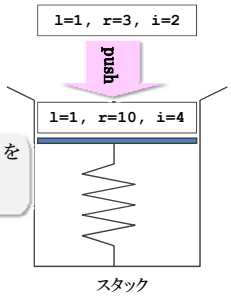
クイックソート：スタックの利用

- 例
  - quick_sort(1, 10)
  - quick_sort(1, 3)



クイックソート：スタックの利用

- 例
  - quick_sort(1, 10)
  - quick_sort(1, 3)
  - quick_sort(1, 1)



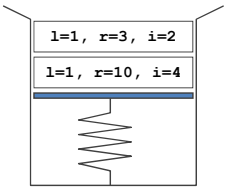
現在の状態（left = 1, right = 3, i = 2）をスタックに保存し，左側の列の処理 quick_sort(1,1) をしよう

push 後 left と right を 1, 3 から 1, 1 に書き換える

1	2	3	4	9	5	6	8	10	7

クイックソート：スタックの利用

- 例
  - quick_sort(1, 10)
  - quick_sort(1, 3)
  - quick_sort(1, 1)



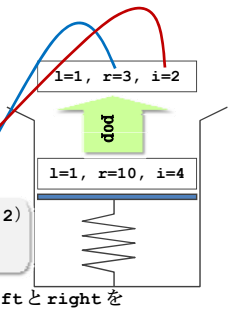
partition

値が 1 つなので，この部分の整列は終わり

1	2	3	4	9	5	6	8	10	7

クイックソート：スタックの利用

- 例
  - quick_sort(1, 10)
  - quick_sort(1, 3)
  - quick_sort(1, 1)
  - quick_sort(3, 3)



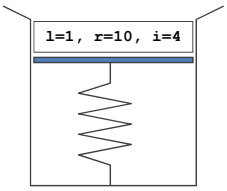
スタックの先頭（left = 1, right = 3, i = 2）から状態を読み出し，右側の列の処理 quick_sort(3,3) をしよう

pop 後 left と right を 1, 1 から 3, 3 に書き換える

1	2	3	4	9	5	6	8	10	7

クイックソート：スタックの利用

- 例
  - quick_sort(1, 10)
  - quick_sort(1, 3)
  - quick_sort(1, 1)
  - quick_sort(3, 3)



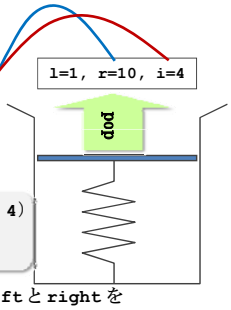
partition

値が 1 つなので，この部分の整列は終わり

1	2	3	4	9	5	6	8	10	7

クイックソート：スタックの利用

- 例
  - （省略）
  - quick_sort(3, 3)
  - quick_sort(5, 10)



スタックの先頭（left = 1, right = 10, i = 4）から状態を読み出し，右側の列の処理 quick_sort(5,10) をしよう

pop 後 left と right を 3, 3 から 5, 10 に書き換える

1	2	3	4	9	5	6	8	10	7

クイックソート：計算量

- 最良の場合
  - 基準値によって左側の列と右側の列が半分に別れていくとき
  - 再帰の繰り返しは $\log n$ 回
  - 全体は $O(n \log n)$
- 最悪の場合
  - 基準値が最大値，または最小値のとき
  - 列の大きさが 1 つずつしか減っていかない
  - 再帰の繰り返しは n 回
  - 全体は $O(n^2)$

クイックソート：高速化の知恵

- 基準値（ピボット）の選び方
  - 今までは右端の値（`a[right]`）を基準値にしたが，三数値を取って（`a[left]`, `a[(left+right)/2]`, `a[right]`），その真ん中の値を基準値 `a*` に採用
  - こうすると最悪の状況になる可能性が小さくなる
- 規模が小さい等，クイックサーチが不適であることがわかれば挿入法にスイッチ（そのほうが早い）

クイックソートのまとめ

- 平均計算量 $O(n \log n)$
- 最悪計算量 $O(n^2)$
- 安定ソートではない
  - 整列されていない列でのデータの入れ替えでは元の順番が保存されない
- 内部ソート
  - 外部記憶を必要としない
- 最悪計算量は悪いが，実際の使用状況では最速のソーティングアルゴリズム（マージソートより速い）
- さまざまなところで使用されている

アルゴリズムのオーダー

- アルゴリズムの時間計算量が $f(n)$ のオーダーである：$O(f(n))$ である
  - 入力データの大きさ n に対し，アルゴリズムの実行時間が関数 $f(n)$ に比例して増加する

係数は考えない

さきほどの例の場合：

最大ステップ数 $15 = 3 \times \mathrm{MAX}$

平均ステップ数 $9 = \left(\sum_{i=1}^{\mathrm{MAX}} 3i \right) / \mathrm{MAX} = \frac{3}{2}(\mathrm{MAX}+1)$

配列サイズ＝入力データサイズと考えると．．．
最大時間計算量，平均時間計算量とも $O(n)$ である

オーダーの見積もり

- 計算量のオーダー表現：
  - きわめて大雑把な評価尺度
  - 大雑把な見積もりで導出することができる

1. アルゴリズムを小さな操作単位に分割
2. 各操作単位のオーダーを評価
3. 操作単位のオーダーを合成して，全体のオーダーを得る

アルゴリズムの分割

```
search(key) /* 配列 perm の中から値 key の位置を探す */
int   key;
{
        int   i = 0;

        while (i < MAX) {
              if (perm[i] == key)
                    return(i);
              i++;
        }
        return(-1);
}
```

実行時間が入力サイズに依存しないステップ（基本ステップ）

ループ回数が入力サイズに依存するループ構造

オーダーの評価 (1)

- ルール 1：**基本ステップのオーダーは $O(1)$**
- 基本ステップ
  - 実行時間が入力サイズに依存しないステップ
    - 変数への代入
    - 数値の演算
    - ポインタ操作 etc.
- 一般に，以下は基本ステップでないことに注意
  - （入力サイズに依存した）配列のコピー
  - 関数呼び出し

オーダーの評価 (2)

- ルール 2: $O(f(n))$ の操作と $O(g(n))$ の操作を連続して行う場合，操作全体のオーダーは $O(\max(f(n), g(n)))$

ただし，関数の大小比較は増加率により行う

$1 < \log n < n < n \log n < n^2 < n^3 < \ldots 2^n$

オーダーの評価 (3)

- ルール 3: $O(f(n))$ 回だけまわるループの内部で $O(g(n))$ の操作を実行する場合，全体のオーダーは $O(f(n) \times g(n))$

- 係数は無視してよい
- 最高次の項以外は無視してよい

ポイント (1/3)

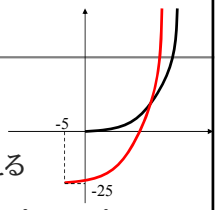
- 係数は無視する
  - $2n \to n$, $3n \to n$, $10n^2 \to n^2$, $5\log n \to \log n$
  - 足し算は足し算でなくなる
    - $O(n)+O(n) \to O(n+n) \to O(2n) \to O(n)$
    - $O(1)+O(1) \to O(1+1) \to O(2) \to O(1)$

仮に元の係数が 100 だとしても，定数 c はいくらでも大きく決められるので，c を 100 倍にすれば問題ない

係数と増加率は関係ない（係数を大きくしても関数の増加率は変わらない）

ポイント (2/3)

- 次数の大きい項だけ残す
  - $n^2+n \to n^2$, $n^3+100n^2 \to n^3$, $n+n\log n \to n \log n$
  - 足し算すると次数の低い項は消える
    - $O(n^2)+O(n) \to O(n^2+n) \to O(n^2)$
    - $O(n^2)+O(n^2)+O(n^3)+O(n^2) \to O(n^3+3n^2) \to O(n^3)$



n^2+10n というのは $n^2+10n+25-25=(n+5)^2-25$ であり，n^2 のグラフをただ x 軸方向に -5，y 軸方向に -25 ずらしただけ

つまり増加率は n^2 となんら変わらないので $10n$ は無視できる

無限に大きな n を考えてみよう
（例えば n = 10000000000000000000000000）
そうすれば -5 も -25 もほとんど増加率には無意味

ポイント (3/3)

- ループは掛け算
  - $O(n) \times O(n^2) \to O(n \times n^2) \to O(n^3)$
  - 括弧の中の計算は先にやっておく
    - $O(n) \times \{O(1)+O(n)\} \to O(n) \times \{O(n)\} \to O(n^2)$

内側から順番に計算していくとややこしくない

必ず 1 つの掛け算になる

$O(n) \times \{ O(n) \times \{\{O(1) + O(n)\} \times \{O(1) + (1)\}\}\}$

も括弧を 1 つずつ外していけばやさしい

オーダー評価の例

```
search(key) /* 配列 perm の中から値 key の位置を探す */
int   key;
{
        int   i = 0;                                   O(1)
        while (i < MAX) {                              O(n)
            if (perm[i] == key)                        O(1)
                return(i);                             O(1)
            i++;                                       O(1)
        }
        return(-1);                                    O(1)
}
```

O(1) + O(n) + O(n) + O(1)

ループの回数：平均時，最悪時とも $O(n)$

⇒ 平均時間計算量，最大時間計算量とも $O(n)$

オーダー評価：特殊ケース 1

- 条件分岐部の評価には要注意

```
if (x % 2 == 0)
    O(f(n)) の処理
else
    O(g(n)) の処理
```

➡ 計算量は $O(\max(f(n), g(n)))$

```
if (x % 2 == 3)
    O(f(n)) の処理
else
    O(g(n)) の処理
```

➡ 計算量は $O(g(n))$

表現上の構造にとらわれず，実質的な振舞いの把握が必要

オーダー記法に用いる関数

- n, $n\log n$, n^2, n^3 ：n の多項式
  - 多項式時間アルゴリズム
  - Polynomial Time Algorithm
  - 現実的
- 2^n, $n!$, n^n ：n の指数関数
  - 指数時間アルゴリズム
  - Exponential Time Algorithm
  - 非現実的

多項式オーダーと指数オーダー

- 計算速度向上の効果

表 1.3 計算速度向上の効果

	1秒で解ける最大の問題サイズ				
計算速度	$O(2^n)$	$O(n)$	$O(n^2)$	$O(n^5)$	$O(n^{10})$
1	100	100	100	100	100
2	101	200	141	115	107
10	103	1000	316	158	126
100	107	10000	1000	251	158
1000	110	100000	3162	398	200
10000	113	1000000	10000	631	251
100000	117	10000000	31623	1000	316
1000000	120	100000000	100000	1585	398

再帰アルゴリズム

- 処理手順が自身を用いて定義されているもの

```
recursive (n) {
    if (自明なケース) {
        自明なケースの処理 ; /* 終了条件 */
    } else {
        recursive (m) ;  /* m < n */
        (処理) ;
    }
}
```

  - 自身の引数より小さな引数で自身を呼び出す
  - 自明なケースの処理が存在
  - 表面的にループが出現しない

再帰プログラムの例: 階乗の計算

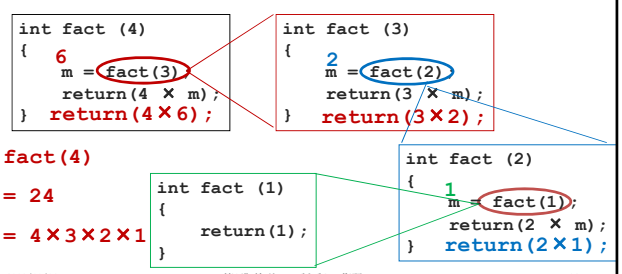
- 階乗
  - 例: $6! = 5\times4\times3\times2\times1$
- ヒント
  - $6! = 6\times5!$, $5! = 5\times4!$, ⋯, $2! = 2\times1!$, $1! = 1$
- プログラム

```
int fact (int n)
{
    int m;
    if(n = 1)
        return(1);
    else{
        m = fact(n-1);
        return(n × m);
    }
}
```

ちょっとフローチャートでは書けない

再帰プログラムの概念

- ちょっと分かりにくいので以下の図のように考えるとよい



ユークリッドの互除法を再帰で書く

- ヒント
  - `r = 0` でないなら，`m`, `n` の最大公約数の代わりに `n`, `r` の最大公約数を求める

```
int gcd (int m, int n)
{
    int r;
    r = m % n;
    if(r = 0)
        return(n);
    else
        return( gcd(n,r) );
}
```

`r=0` なら `n` が 最大公約数

`r=0` でないなら `n` と `r` の 最大公約数を求める

オーダー評価：特殊ケース 2

再帰プログラムのオーダー評価は，少し面倒

```
int recursive(int n)
{
  if (n <= 1)
    return(1);
  else
    return(recursive(n - 1) + recursive(n - 1));
}
```

入力が `n` のときの，この再帰プログラムの計算量を T_n とする

この場合のステップ数は，漸化式 $T_n = 2T_{n-1}$ で与えられる

⇒ 計算量は $O(2^n)$

（互除法は $T_n = T_{n-1}$ なので $O(n)$）

第 6 講のまとめ

- 整列アルゴリズムの復習
- オーダー記法の復習