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 (2)

第5講: 平成20年11月7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 1 講の復習

- アルゴリズムの定義
  - 入力と出力
  - 正当性, 決定性, 有限性, 停止性
- ユークリッドの互除法
- フローチャートの描き方
- 疑似言語の書き方

第 2 講の復習

- アルゴリズムの評価は時間計算量で行う
  - 領域計算量もある
- 計算量の評価にはオーダー記法を用いる
  - 並んでいる計算量は足し算
  - 繰り返しに含まれる計算量は掛け算
  - 係数は省略
- 多項式オーダーと指数オーダー
  - 指数オーダーのアルゴリズムは使い物にならない
- 再帰プログラム

第 3 講の復習

- アルゴリズムとデータ構造
  - アルゴリズムに適したデータ構造の選択が必要
- 基本的なデータ構造(抽象データ型)
  - 配列
    - 参照は $O(1)$, 挿入・削除は $O(n)$
  - リスト
    - 参照は $O(n)$, 挿入・削除は $O(1)$
  - キュー (待ち行列)
    - 先入れ先出し, ランダム参照不可, 追加・取出しは $O(1)$
  - スタック
    - 後入れ先出し, ランダム参照不可, 追加・取出しは $O(1)$
  - 木
    - 根(root), 親, 子, 葉(終端頂点), 非終端頂点, 高さ, 深さ

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

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

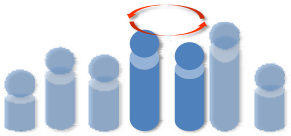
整列問題の分類

- 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 ;
}
```



今日の講義の内容

- 今日紹介する整列アルゴリズム
  - 選択ソート（先週）
  - バブルソート（先週）
  - 挿入法（先週）
  - マージソート
  - クイックソート

マージソート：概念

- merge
  - [名詞]マージ
  - [他動詞]（二つ以上のものを）一つにまとめる，～を合併させる
  - [自動詞] 合併する，同化吸収する，融合する
- アルゴリズム
  1. 最初に隣合う 2 つの要素で大小関係を比較し，長さ 2 の整列された列を作る
  2. 隣合う整列された 2 本の列を合わせて（マージ），整列された 1 本の列を作る
  3. 最終的に 1 本になるまで繰り返す

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
  - マージ法（ここで提案したい方法）
    - 隣合う 2 人ペアで誕生日を言い合う，年少の子は後ろへ並ぶ
    - 次に隣の 2 人組と並んでいる順に誕生日を言い合う，年少なら後ろに並ぶ
    - 4 人組で，8 人組で続けていく
    - 最終的には誕生日の順に並ぶ

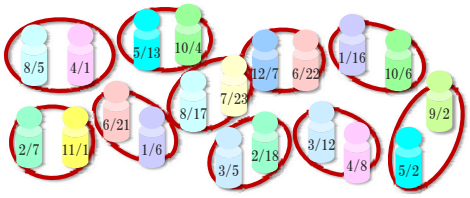
例題：全体法

- 全員で誕生日を言う
  1. 全員（n 人）の誕生日を知るには $O(n)$ かかる
  2. みんなは自分が最年長かどうか耳を澄ます
  3. 最年長の子が抜ける
  4. 1. へ戻る
  5. 1 人ずつしか抜けていかないので $O(n)$ 繰り返す
- 全体のオーダーは $O(n^2)$

最年長が抜ける

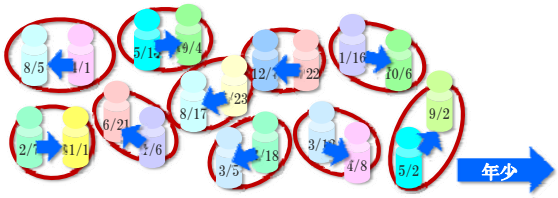
例題：マージ法

- 2^i 人ずつ誕生日の順に並んでいく
  1. 隣合う 2 人で誕生日を言い合い，年少者は後ろに並ぶ（O(2 人 × n/2 組) = $O(n)$）



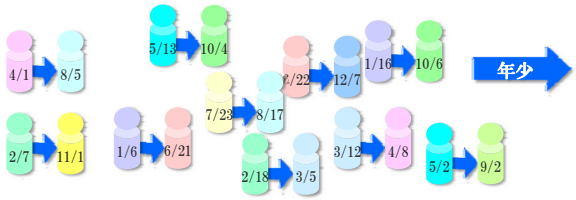
例題：マージ法

- 2^i 人ずつ誕生日の順に並んでいく
  1. 隣合う 2 人で誕生日を言い合い，年少者は後ろに並ぶ（O(2 人 × n/2 組) = $O(n)$）



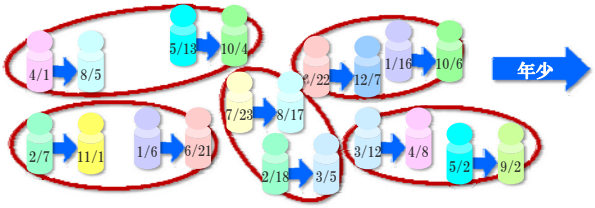
例題：マージ法

- 2^i 人ずつ誕生日の順に並んでいく
  1. 隣合う 2 人で誕生日を言い合い，年少者は後ろに並ぶ（O(2 人 × n/2 組) = $O(n)$）



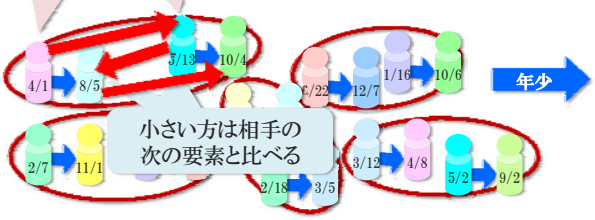
例題：マージ法

- 2^i 人ずつ誕生日の順に並んでいく
  2. 隣合う 2 人組 2 組で誕生日を言い合い，年少者は後ろに並ぶ（O(4 人 × n/4 組) = $O(n)$）



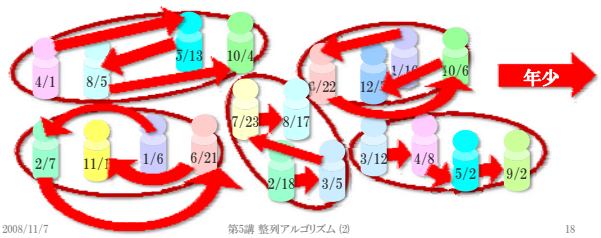
例題：マージ法

- 2^i 人ずつ誕生日の順に並んでいく
  2. 先頭同士を比べる 組で誕生日を言い合い，年少
     は後 並ぶ（O(4 人 × n/4 組) = $O(n)$）



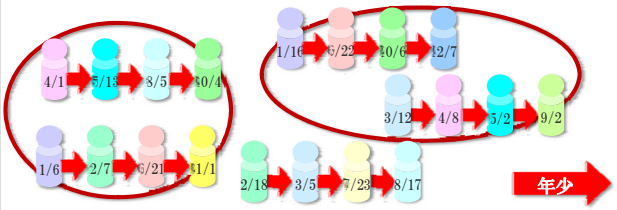
例題：マージ法

- 2^i 人ずつ誕生日の順に並んでいく
  2. 隣合う 2 人組 2 組で誕生日を言い合い，年少者は後ろに並ぶ（O(4 人 × n/4 組) = $O(n)$）



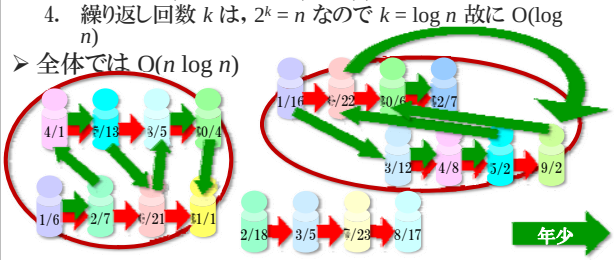
例題：マージ法

- 2^i 人ずつ誕生日の順に並んでいく
  2. 隣合う 4 人組 2 組で誕生日を言い合い，年少者は後ろに並ぶ（$O(8\text{ 人} \times n/8\text{ 組}) = O(n)$）



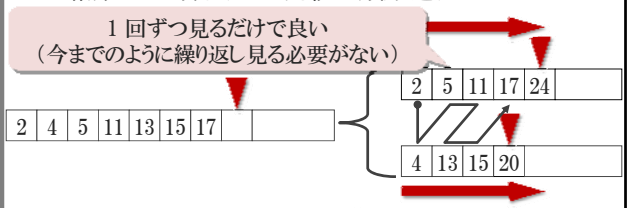
例題：マージ法

- 2^i 人ずつ誕生日の順に並んでいく
  3. 隣合う 4 人組 2 組で誕生日を言い合い，年少者は後ろに並ぶ（$O(8\text{ 人} \times n/8\text{ 組}) = O(n)$）
  4. 繰り返し回数 k は，$2^k = n$ なので $k = \log n$ 故に $O(\log n)$
- 全体では $O(n \log n)$



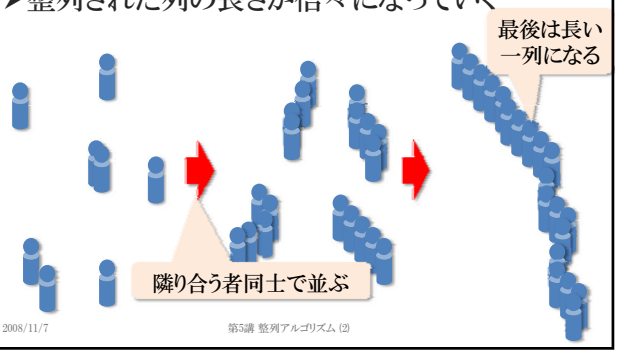
マージソート：概念

- 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



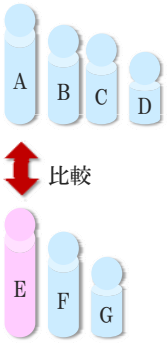
マージソート：概念図

- 整列された列の長さが倍々になっていく



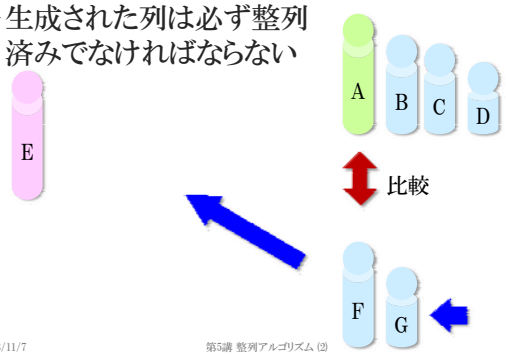
マージソート：列の生成

- 生成された列は必ず整列済みでなければならない



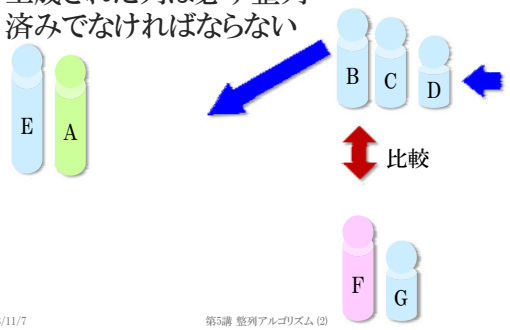
マージソート：列の生成

- 生成された列は必ず整列済みでなければならない



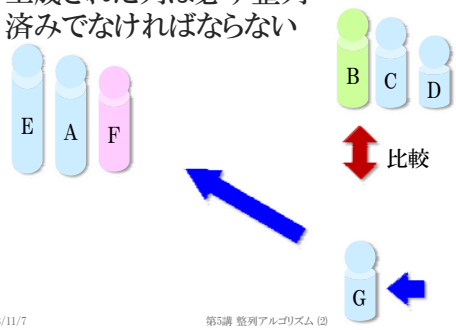
マージソート：列の生成

- 生成された列は必ず整列済みでなければならない



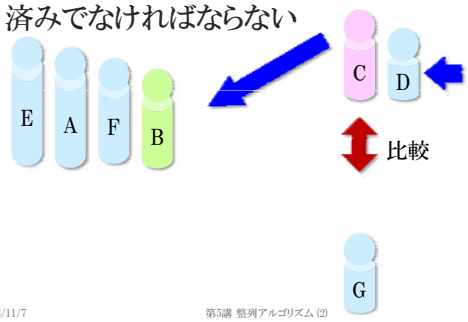
マージソート：列の生成

- 生成された列は必ず整列済みでなければならない



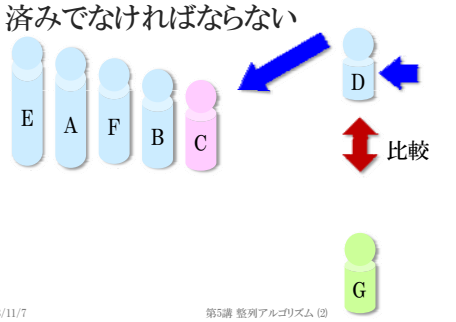
マージソート：列の生成

- 生成された列は必ず整列済みでなければならない



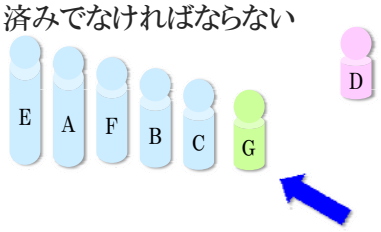
マージソート：列の生成

- 生成された列は必ず整列済みでなければならない



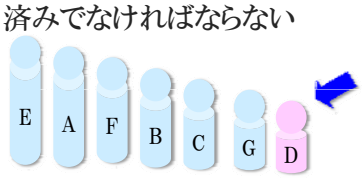
マージソート：列の生成

- 生成された列は必ず整列済みでなければならない



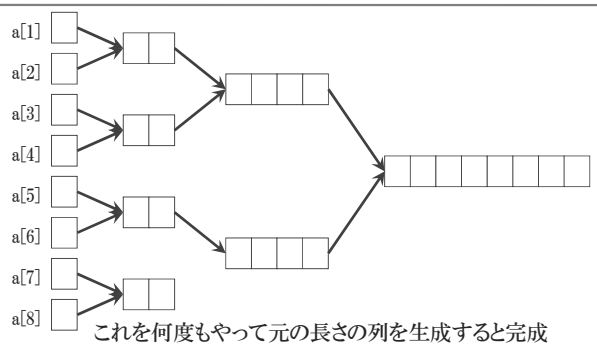
マージソート：列の生成

- 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！

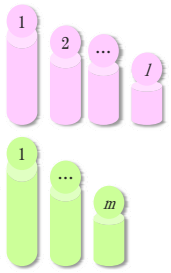
マージソート：列の生成



これを何度もやって元の長さの列を生成すると完成

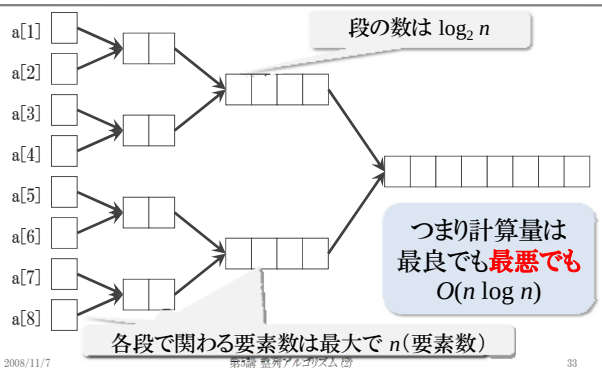
マージソート：計算量

- 計算量（比較の回数）は？
  - 列の長さをそれぞれ l, m ($l > m$) とすると
  - 最良の場合：m
    - 短い方が全部長い方より小さい
  - 最悪の場合：$l + m$
    - ジグザグに列を生成



つまり，最悪でも関わった要素数だけの比較回数しかかからない（1 回の繰り返し当たり $O(n)$）

マージソート：列の生成



マージソート：再び概要

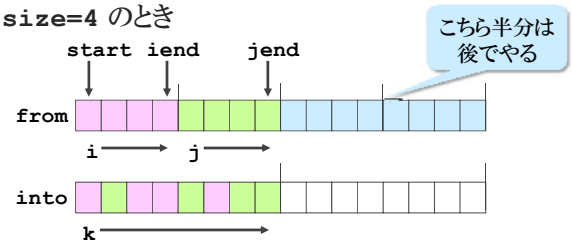
- 単純マージソート：Straight Merge Sort
- 手順
  1. 初期状態の配列の要素を長さ 1 の列とする
  2. これらの n 本の列から 2 本ずつ組み合わせてマージし，長さ 2 の $n/2$ 本の列を得る
  3. 以下同様に長さを 2, 4, … と倍に増やし，全データが 1 本の列になれば終了

マージソート：演習

8	14	6	12	16	2	5	15	7	11	9	1	13	4	10	3
~~8~~	~~14~~	~~6~~	~~12~~	2	16	5	15	7	11	1	9	4	13	3	10
6	8	12	14	2	5	15	16	1	7	9	11	3	4	10	13
2	5	6	8	12	14	15	16	1	3	4	7	9	10	11	13
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**

マージソート：性質

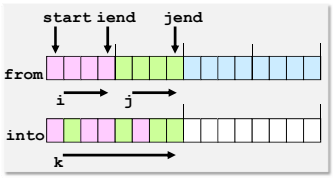
- `size=4` のとき



- 並べ替え結果保存用（`into`）として，同じだけの記憶領域が必要 → 内部ソートではない
- 記憶領域がたくさん要るので人気がない

マージソート: プログラム (1/2)

```
Merge_Seq(size, from, into) {
   start ← 1 ;
   while (start < n) {
      i ← start ; j ← start + size ; k ← start ;
      iend ← min (start+size-1, n) ;
      jend ← min (start+2*size-1, n) ;
      while (i <= iend) and (j <= jend) {
         if (from[i] <= from[j]) { Put(i); }
         else { Put(j) ; }
      }
      while (i <= iend){
         Put(i); }
      while (j <= jend) {
         Put(j); }
      start ← start+2*size;
   }
}
```



マージソート: プログラム (2/2)

➢ メインの部分

```
Straight_Merge_Sort {
     seqsize ← 1 ;
     while (seqsize < n) {
          Merge_Seq(seqsize, a, b) ;
          Merge_Seq(2*seqsize, b, a) ;
          seqsize ← 4 * seqsize ;
     }
}

Put (index) {
     into[k] ← from[index] ;
     index ← index + 1 ; k ← k + 1 ;
}
```

配列 a から b へ

配列 a から b へ

要素を 1 つ書き出す

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

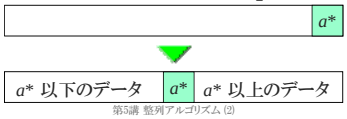
- 最悪でも計算時間が $O(n \log n)$ と早いが，外部ソートであるため，実際はあまり使われることがない
- 最悪時の計算量の上限を保証したいときは使う

クイックソート

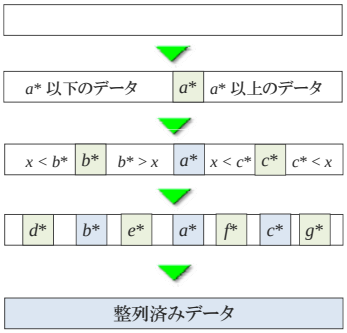
- クイックソート: Quick Sort
- Charles A. R. Hoare が考案
- 内部ソートでは最も速いアルゴリズム
- 平均計算量: $O(n \log n)$

クイックソート: 概要

- アルゴリズム
  - 基準値 a^* を選ぶ
  - 基準値 a^* 以下のデータを左側，残りを右側に分割
  - 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

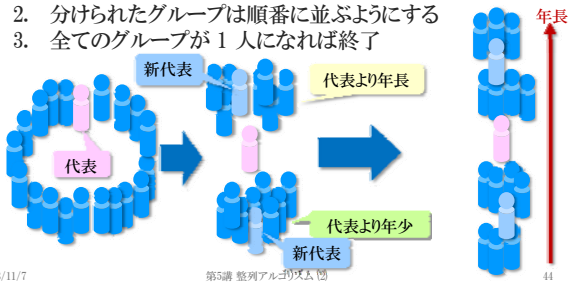


例題：子供の並べ替え

- 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

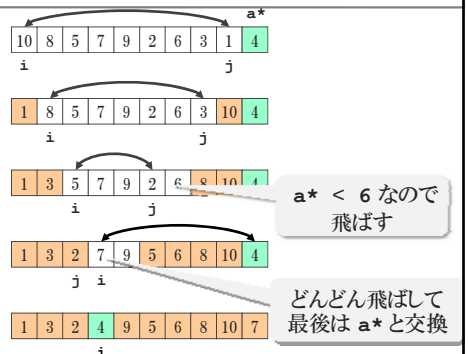
- partition (a*, left , right)
  - 基準値（ピボット; pivot）a* によってデータを 2 つの部分に分割し，基準値の挿入されるべき位置 i を返す関数

クイックソート：partition 手続き

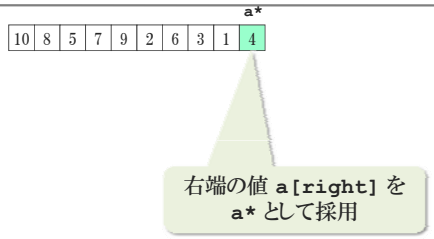
- 基準値（ピボット）によってデータを 2 つの部分に分割
- partition(a* , left , right)

- 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

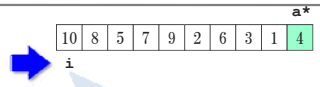
クイックソート：partition 手続き



クイックソート：partition 手続き

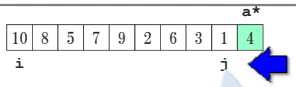


クイックソート: partition 手続き



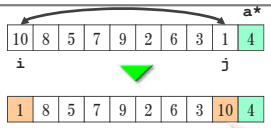
a[i] > a* なる i を探索
たまたま 1 つ目で当たる

クイックソート: partition 手続き



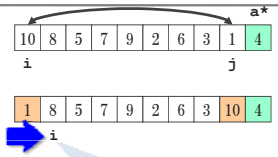
a[j] < a* なる j を探索
またもや 1 つ目で当たる

クイックソート: partition 手続き



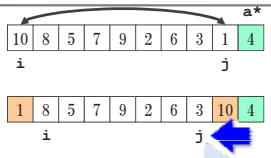
というわけで交換

クイックソート: partition 手続き



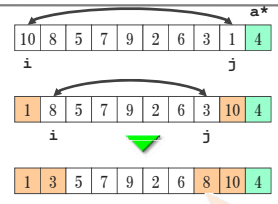
a[i] > a* なる i を探索
次から探していることに注意

クイックソート: partition 手続き



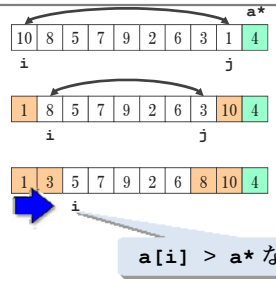
a[j] < a* なる j を探索
こちらも同様

クイックソート: partition 手続き

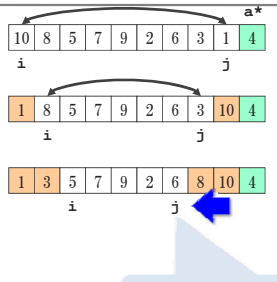


見つかったので交換

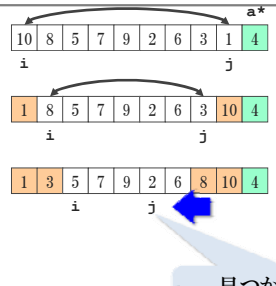
クイックソート: partition 手続き



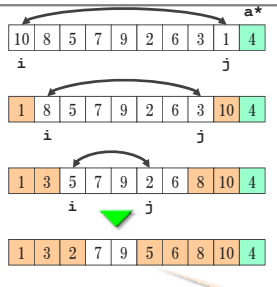
クイックソート: partition 手続き



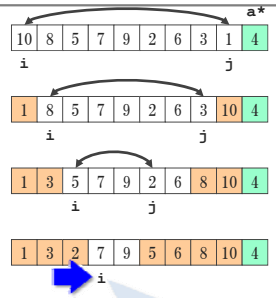
クイックソート: partition 手続き



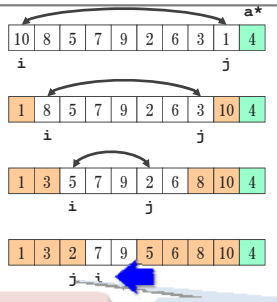
クイックソート: partition 手続き



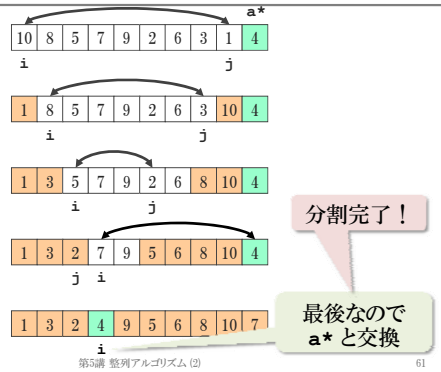
クイックソート: partition 手続き



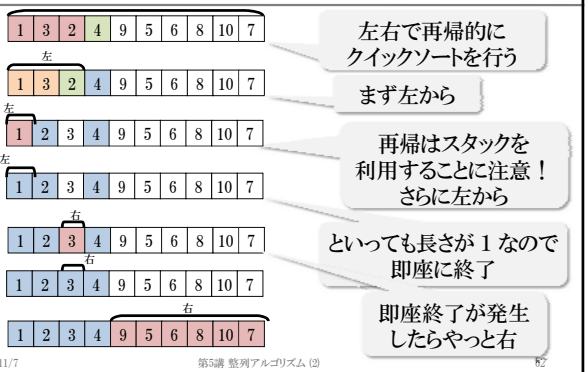
クイックソート: partition 手続き



クイックソート：partition 手続き



クイックソート



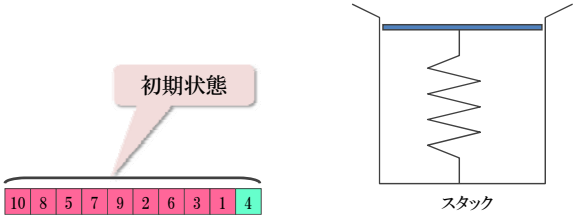
クイックソート

- 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 ) ;
    }
}
```

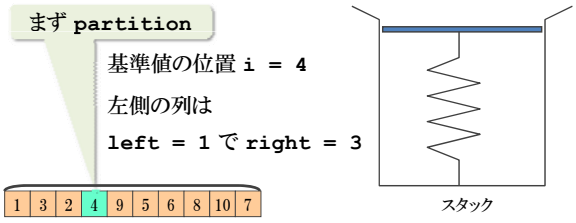

クイックソート：スタックの利用

- 例
  - quick_sort(1, 10)



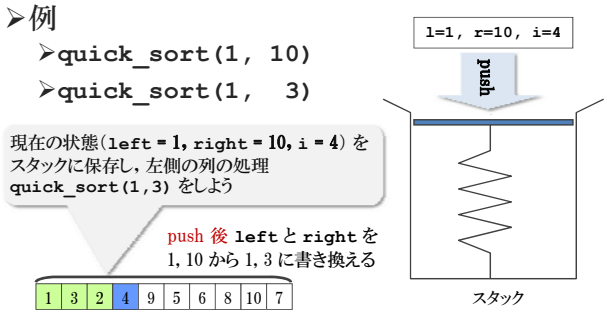
クイックソート：スタックの利用

- 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

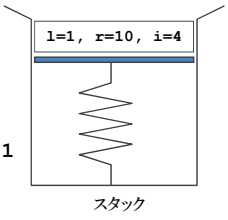
- 例
  - quick_sort(1, 10)
  - quick_sort(1, 3)

partition

基準値の位置 i = 2

左側の列は

left = 1 で right = 1



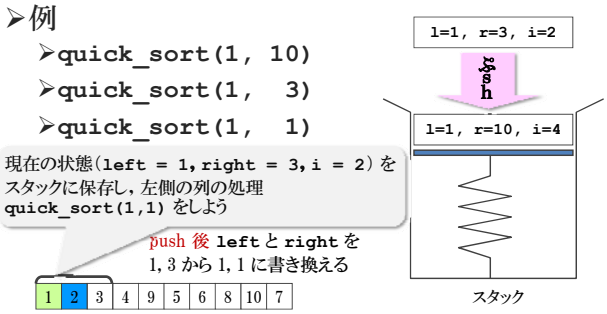
1	2	3	4	9	5	6	8	10	7

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)

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



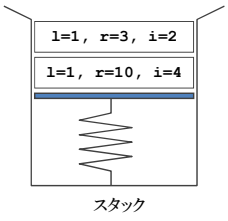
1	2	3	4	9	5	6	8	10	7

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



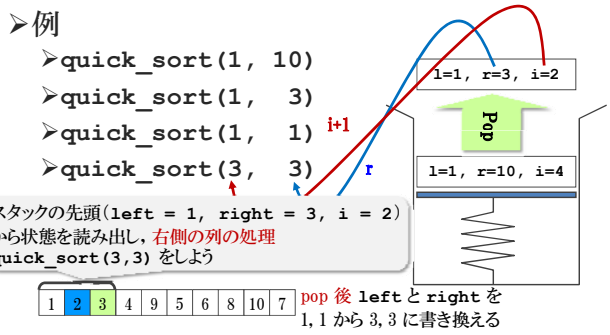
1	2	3	4	9	5	6	8	10	7

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



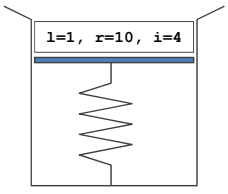
1	2	3	4	9	5	6	8	10	7

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



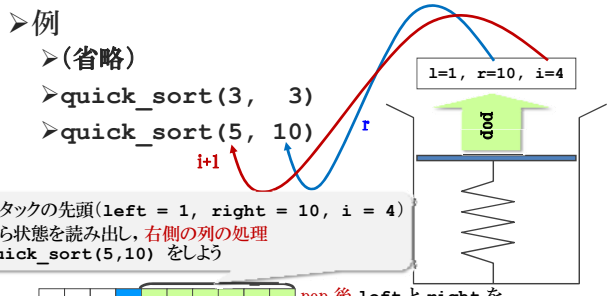
1	2	3	4	9	5	6	8	10	7

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: 演習

- 区切った列の右端を基準値(ピボット)にする

3	10	4	13	1	9	11	7	15	5	2	16	12	6	14	8
3	6	4	2	1	5	7	8	15	9	13	16	12	10	14	11
3	6	4	2	1	5	7	8	10	9	11	16	12	15	14	13
3	1	4	2	5	6	7	8	9	10	11	12	13	15	14	16
1	2	4	3	5	6	7	8	9	10	11	12	13	15	14	16
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

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

第 5 講のまとめ

- 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
    - クイックソート