アルゴリズムとデータ構造

第9回　整列アルゴリズム（1）

注意：
分かり易いように動作説明を
何枚かのスライドに分けて
少しずつ書いています
（枚数にビックリしないでね）

小技：
同じような絵が描いてある
ページは、PCのキー操作で
次前次前次前と往復すると
変化した場所が分かりやすい

第9回 整列アルゴリズム (1)

- 今日の内容
  - 整列（ソート, sorting）
  - 整列アルゴリズムの種類と特徴
  - $O(n^2)$ 時間の整列アルゴリズム
    - 選択ソート、挿入ソート、バブルソート
  - $O(n (\log n)^2)$ 時間の整列アルゴリズム
    - シェルソート
  - 平均時 $O(n \log n)$ 時間の整列アルゴリズム
    - クイックソート

第9回 整列アルゴリズム (1)

「整列」って何かは、
次のページから説明します

- ポイント
  - 整列（ソート, sorting）をするための、
    色んなやり方を知ろう

- Q: でも、何で色々勉強するの？
  Best のものを1つ知っていればいいのでは？

- A1: 素朴なアイデアがアルゴリズムとして
  形になるのを知ると、自信がつく
- A2: やり方が1つだけしか許されないのは窮屈
  自由にアイデアを出せる方が、楽しい
- A3: その上で、高速なアルゴリズムを知ると
  その凄さが実感できる

整列（ソート，sorting）

データを大きい順あるいは小さい順に**並べ替える**こと

龍太郎	恵三	喜朗	純一郎	康夫	太郎	由紀夫	直人	佳彦	晋三
7	6	3	8	2	1	0	4	5	9

数値の大きい順に並べると…

晋三	純一郎	龍太郎	恵三	佳彦	直人	喜朗	康夫	太郎	由紀夫
9	8	7	6	5	4	3	2	1	0

データ分析の基本中の基本

整列の形式的な定義

・整数の大小は、分かりやすい全順序関係
・同様に、辞書は、単語に全順番を付けて並べてある

（授業なので、「大小」って何？をマジメに書きます）

集合X上の**全順序**（total order, 線形順序（linear order））とは，
X上の要素間の**2項関係**「$\leq$」で，次の性質を持つものをいう

(1) $x \leq x$ for all $x \in X$ （**反射律** reflexivity）

(2) $x \leq y,\ y \leq z \ \Rightarrow\ x \leq z$ （**推移律** transitivity）

(3) $x \leq y,\ y \leq x \ \Rightarrow\ x = y$ （**反対称律** anti-symmetry）

(4) $x \leq y$ or $y \leq x$ for all $x, y \in X$ （**比較可能性** comparability）

全順序 $\leq$ が定義された集合Xの相異なる2つの要素 x と y に対して
$x \leq y$ が成り立つとき，***x* は *y* より小さい*** ということにする

整列 … 全順序が定義されている集合の要素がリストとして与えられたとき，
それを小さい順に並び替える処理

関係を逆にすれば，大きい順になる

以下では、リストは配列 A[0], A[1], …, A[n-1] で与えられるものとする.

整列アルゴリズムの種類と特徴

今日のアルゴリズム

アルゴリズム	最悪時間計算量の漸近的上界	コメント
選択ソート (selection sort) 挿入ソート (insertion sort) バブルソート (bubble sort)	$O(n^2)$	直感的に理解しやすい
シェルソート (shell sort)	$O(n(\log n)^2)$	実用性は高い．平均時間計算量で$O(n \log n)$であるかは未解決
クイックソート (quick sort)	$O(n^2)$	平均時間計算量は$O(n \log n)$ 実用上最も高速．分割統治法
マージソート (merge sort) ヒープソート (heap sort)	$O(n \log n)$	最悪時間計算量の漸近的上界が最小．マージソートは分割統治法
バケットソート (bucket sort) 基数ソート (radix sort)	$O(n)$ 注)	高速だが，ある範囲に限定された整数に対してのみ適用可能

注）バケット数と桁数を定数とみた場合

選択ソート（selection sort）

アルゴリズムを見ながら、
右図の動作を追いかけよう

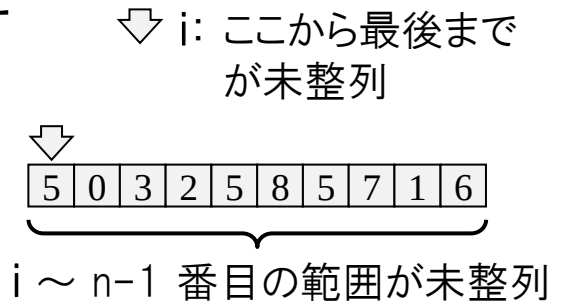
まだ整列していない中から最小のものを取り出す
という操作を繰り返して整列（ソート）する

［アルゴリズム］

step 1: i←0

step 2: i≧n-1ならば停止
そうでなければ j←arg min{A[j]:i≦j<n}を実行

step 3: A[i]とA[j]の値を交換
i←i+1としてstep 2へ



最悪/最良/平均時間計算量は$\Theta(n^2)$

n: 要素数

選択ソート（selection sort）

まだ整列していない中から最小のものを取り出すという操作を繰り返して整列（ソート）する

［アルゴリズム］

step 1: i←0

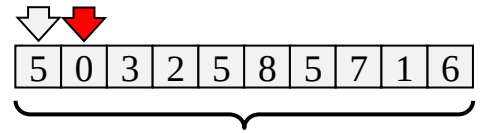
step 2: i≥n-1ならば停止
そうでなければ j←arg min{A[j]:i≤j<n}を実行

step 3: A[i]とA[j]の値を交換
i←i+1としてstep 2へ

A[j] が最小になるような j を求める

j：最小値のある場所

i：ここから最後までが未整列



i ～ n−1 番目の範囲が未整列

Step 2：j を求める
（最小値はどこ?）

最悪/最良/平均時間計算量は$\Theta(n^2)$

n: 要素数

選択ソート（selection sort）

まだ整列していない中から最小のものを取り出すという操作を繰り返して整列（ソート）する

［アルゴリズム］

step 1: i←0

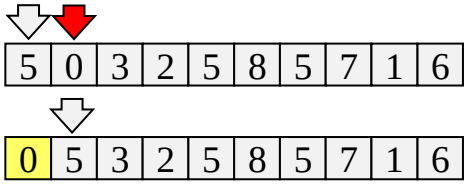
step 2: i≥n-1ならば停止
そうでなければ j←arg min{A[j]:i≤j<n}を実行

step 3: A[i]とA[j]の値を交換
i←i+1としてstep 2へ

まだ整列していない中で最小のものを A[i] へ

j：最小値のある場所

i：ここから最後までが未整列



Step 3：A[i] と A[j] の値を交換

i ← i + 1

最悪/最良/平均時間計算量は$\Theta(n^2)$

n: 要素数

選択ソート（selection sort）

まだ整列していない中から最小のものを取り出すという操作を繰り返して整列（ソート）する

［アルゴリズム］

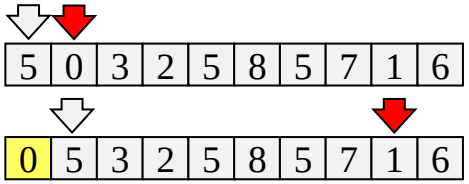
step 1: i←0

step 2: i≧n-1ならば停止
そうでなければ j←arg min{A[j]:i≦j<n}を実行

step 3: A[i]とA[j]の値を交換
i←i+1としてstep 2へ

j：最小値のある場所

i：ここから最後までが未整列



Step 2：j を求める
（最小値はどこ?）

最悪/最良/平均時間計算量は$\Theta(n^2)$

n: 要素数

選択ソート（selection sort）

まだ整列していない中から最小のものを取り出すという操作を繰り返して整列（ソート）する

［アルゴリズム］

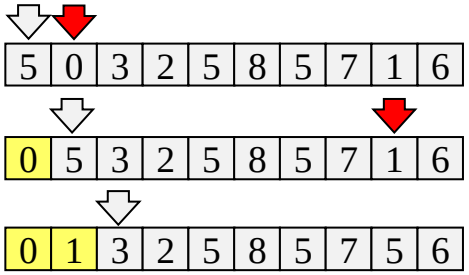
step 1: i←0

step 2: i≥n-1ならば停止
そうでなければ j←arg min{A[j]:i≤j<n}を実行

step 3: A[i]とA[j]の値を交換
i←i+1としてstep 2へ

j：最小値のある場所

i：ここから最後までが未整列



Step 3：A[i] と A[j] の値を交換

i ← i + 1

以下、同様に繰り返す

最悪/最良/平均時間計算量は$\Theta(n^2)$

n: 要素数

選択ソート（selection sort）

まだ整列していない中から最小のものを取り出すという操作を繰り返して整列（ソート）する

［アルゴリズム］

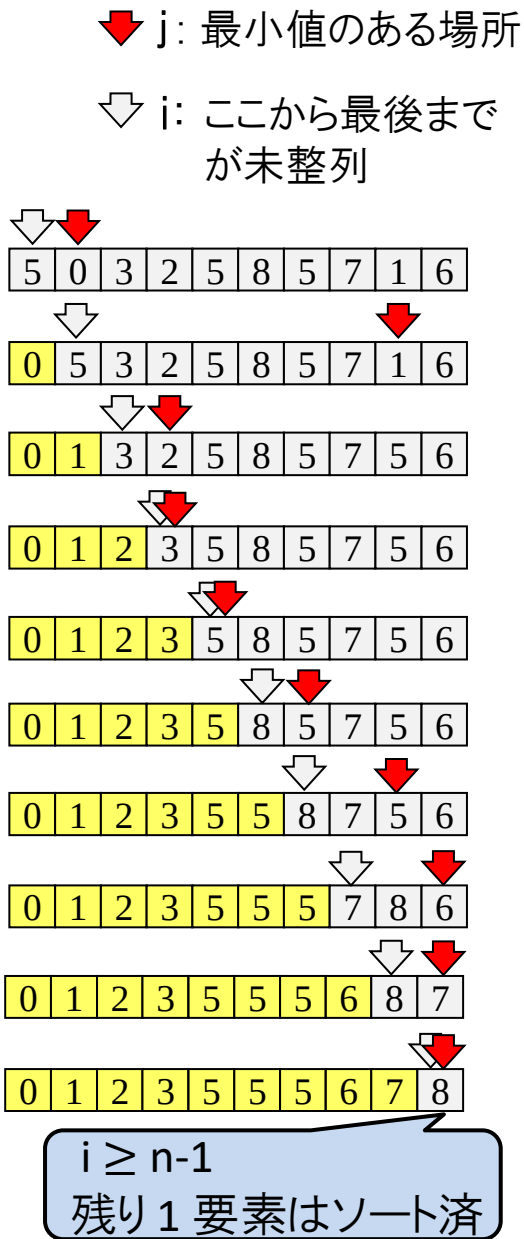
step 1: i←0

step 2: i≥n-1ならば停止
そうでなければ j←arg min{A[j]:i≤j<n}を実行

step 3: A[i]とA[j]の値を交換
i←i+1としてstep 2へ

最悪/最良/平均時間計算量は$\Theta(n^2)$

n: 要素数



選択ソート（selection sort）

まだ整列していない中から最小のものを取り出すという操作を繰り返して整列（ソート）する

［アルゴリズム］

step 1: i←0

step 2: i≧n-1ならば停止
そうでなければ j←arg min{A[j]:i≦j<n}を実行

step 3: A[i]とA[j]の値を交換
i←i+1としてstep 2へ

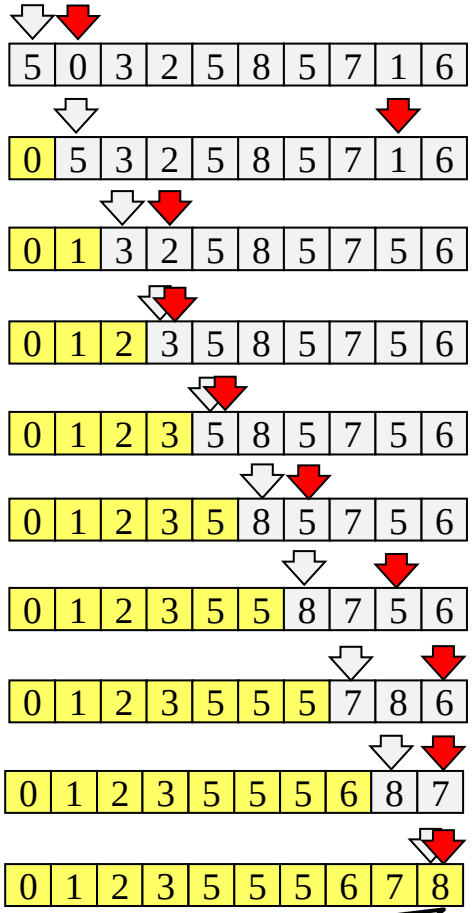
Step 2 は i ～ n-1 の n-i 個の要素をチェック
これを i = 0 ～ n-2 で繰り返す： $\Sigma_{i=0}^{n-2} (n-i) = O(n^2)$

最悪/最良/平均時間計算量は$\Theta(n^2)$

Ω(n^2) かつ O(n^2)、つまり、c n^2 以上で、c' n^2 以下である
まずは、時間計算量が O(n^2) と分かれば、ok

j：最小値のある場所

i：ここから最後までが未整列



i ≧ n-1
残り1要素はソート済

選択ソート（selection sort）

まだ整列していない中から最小のものを取り出すという操作を繰り返して整列（ソート）する

［アルゴリズム］

step 1: i←0

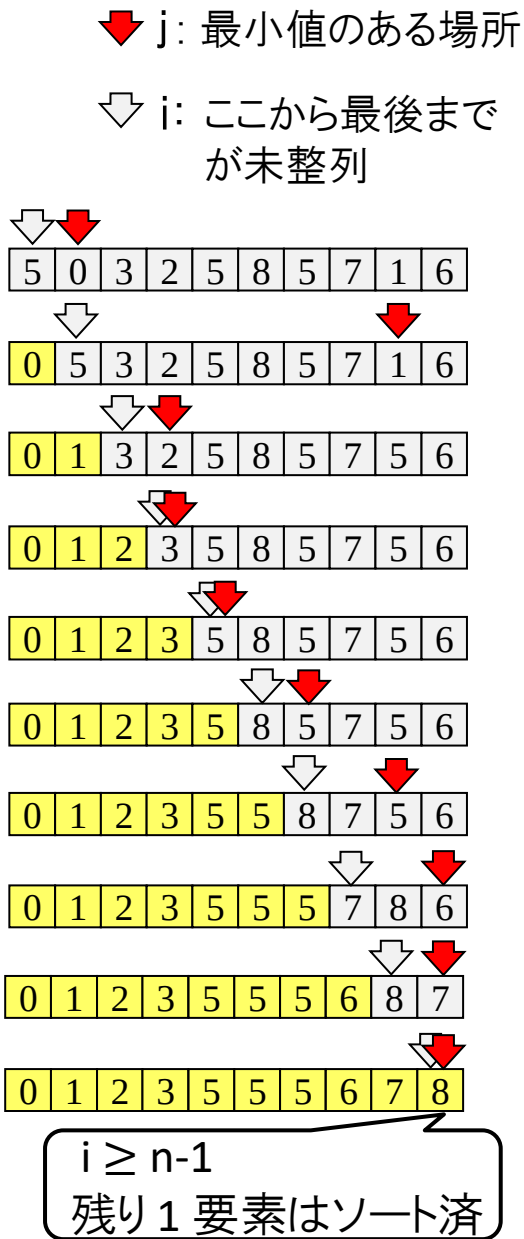
step 2: i≧n-1ならば停止
そうでなければ j←arg min{A[j]:i≦j<n}を実行

step 3: A[i]とA[j]の値を交換
i←i+1としてstep 2へ

Step 2 は i ～ n-1 の n-i 個の要素をチェック
これを i = 0 ～ n-2 で繰り返す： $\Sigma_{i=0}^{n-2} (n-i) = O(n^2)$

最悪/最良/平均時間計算量は$\Theta(n^2)$

もっとザックリ言うと、
Step 2 は O(n) 個の要素をチェック
これを O(n) 回だけ繰り返す → $O(n^2)$



挿入ソート（insertion sort）

アルゴリズムを見ながら、
右図の動作を追いかけよう

整列済みの配列に1つずつ要素を挿入する

［アルゴリズム］

step 1: i←1

step 2: i≥nならば停止
そうでなければ temp←A[i], j←i

step 3: j≥1かつA[j-1]>tempが成り立つ間,
A[j]←A[j-1], j←j-1を繰り返す

step 4: A[j]←temp. i←i+1としてstep 2へ

左方向へ入替ながら探索

最悪時間計算量$\Theta(n^2)$ 逆順にソートされた入力の時

最良時間計算量$\Theta(n)$ ソート済の入力の時

平均時間計算量$\Theta(n^2)$

n: 要素数

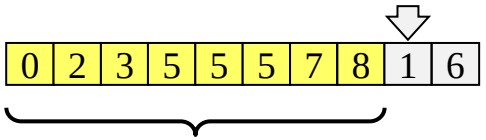
▼ j: ここに入れたい

▽ i: この要素を、整列済みの列に入れたい

イメージをつかむために、
まず、実行途中の様子

i = 8 で Step 2 から実行

▽の要素を、整列済みの列に入れたい
（どこに入れる？）



0 ～ i−1 番目の範囲が整列済

挿入ソート（insertion sort）

整列済みの配列に1つずつ要素を挿入する

j：ここに入れたい
i：この要素を、整列済みの列に入れたい

［アルゴリズム］

step 1: i←1

step 2: i≥nならば停止
そうでなければ temp←A[i], j←i

step 3: j≥1かつA[j-1]>tempが成り立つ間,
A[j]←A[j-1], j←j-1を繰り返す

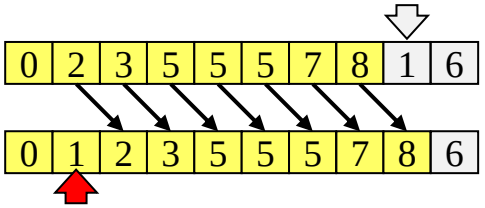
step 4: A[j]←temp．i←i+1としてstep 2へ

左方向へ入替ながら探索

イメージをつかむために、まず、実行途中の様子

i = 8 で Step 2 から実行

の要素を、整列済みの列に入れたい
（どこに入れる？）



ここに入れたい
どうやって？
1つずつ右にずらす

最悪時間計算量$\Theta(n^2)$ 逆順にソートされた入力の時
最良時間計算量$\Theta(n)$ ソート済の入力の時
平均時間計算量$\Theta(n^2)$

n: 要素数

挿入ソート（insertion sort）

整列済みの配列に1つずつ要素を挿入する

［アルゴリズム］

step 1: i←1

step 2: i≥nならば停止
そうでなければ temp←A[i], j←i

step 3: j≥1かつA[j-1]>tempが成り立つ間,
A[j]←A[j-1], j←j-1を繰り返す

step 4: A[j]←temp．i←i+1としてstep 2へ

左方向へ入替ながら探索

最悪時間計算量$\Theta(n^2)$ 逆順にソートされた入力の時

最良時間計算量$\Theta(n)$ ソート済の入力の時

平均時間計算量$\Theta(n^2)$

n: 要素数

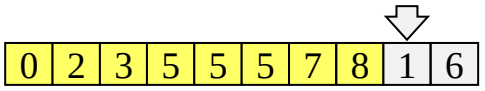
▼ j：ここに入れたい

▽ i：この要素を、整列済みの列に入れたい

イメージをつかむために、まず、実行途中の様子

i = 8 で Step 2 から実行

▽の要素を、整列済みの列に入れたい（どこに入れる？）



Step 2：temp←A[i]
（一旦▽の場所を空ける）
j ← i
（Step3で、A[j] に A[j-1]をずらしてくる）

挿入ソート（insertion sort）

整列済みの配列に1つずつ要素を挿入する

［アルゴリズム］

step 1: i←1

step 2: i≥nならば停止
そうでなければ temp←A[i], j←i

step 3: j≥1かつA[j-1]>tempが成り立つ間,
A[j]←A[j-1], j←j-1を繰り返す

step 4: A[j]←temp．i←i+1としてstep 2へ

左方向へ入替ながら探索

最悪時間計算量$\Theta(n^2)$ 逆順にソートされた入力の時

最良時間計算量$\Theta(n)$ ソート済の入力の時

平均時間計算量$\Theta(n^2)$

n: 要素数

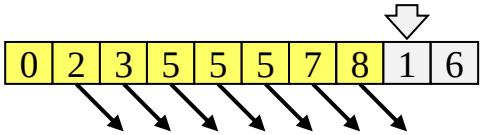
j：ここに入れたい

i：この要素を、整列済みの列に入れたい

イメージをつかむために、まず、実行途中の様子

i = 8 で Step 2 から実行

の要素を、整列済みの列に入れたい（どこに入れる？）



Step 3：
A[j-1] をA[j] にずらす
（j はどこまで？）
（tempを入れる場所は？）

挿入ソート（insertion sort）

整列済みの配列に1つずつ要素を挿入する

［アルゴリズム］

step 1: i←1

step 2: i≥nならば停止
そうでなければ temp←A[i], j←i

step 3: j≥1かつA[j-1]>tempが成り立つ間,
A[j]←A[j-1], j←j-1を繰り返す

step 4: A[j]←temp．i←i+1としてstep 2へ

左方向へ入替ながら探索

最悪時間計算量$\Theta(n^2)$ 逆順にソートされた入力の時

最良時間計算量$\Theta(n)$ ソート済の入力の時

平均時間計算量$\Theta(n^2)$

n: 要素数

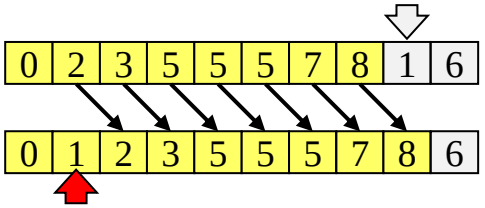
j：ここに入れたい

i：この要素を、整列済みの列に入れたい

イメージをつかむために、まず、実行途中の様子

i = 8 で Step 2 から実行

の要素を、整列済みの列に入れたい（どこに入れる？）



Step 4： j が、ここを指す

挿入ソート（insertion sort）

整列済みの配列に1つずつ要素を挿入する

j：ここに入れたい
i：この要素を、整列済みの列に入れたい

［アルゴリズム］

step 1: i←1

step 2: i≥nならば停止
そうでなければ temp←A[i], j←i

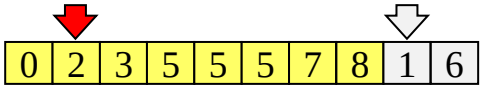
step 3: j≥1かつA[j-1]>tempが成り立つ間,
A[j]←A[j-1], j←j-1を繰り返す

step 4: A[j]←temp．i←i+1としてstep 2へ

左方向へ入替ながら探索

イメージをつかむために、まず、実行途中の様子

以上の操作を
（この要素を入れたい）と
（ここに入れたい）で
以下のように簡略化して描く



最悪時間計算量$\Theta(n^2)$　逆順にソートされた入力の時

最良時間計算量$\Theta(n)$　ソート済の入力の時

平均時間計算量$\Theta(n^2)$

n: 要素数

挿入ソート（insertion sort）

では、最初から

整列済みの配列に1つずつ要素を挿入する

［アルゴリズム］

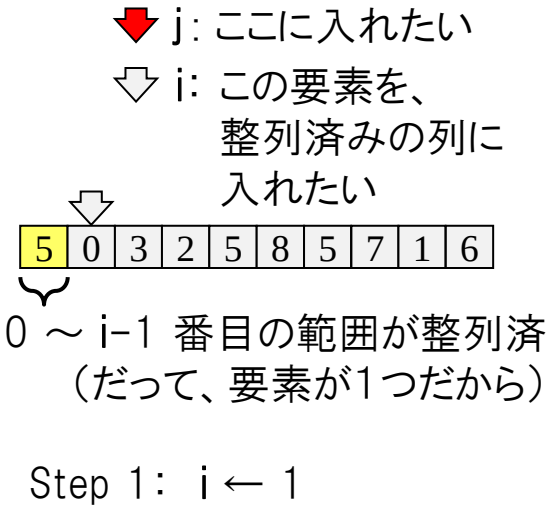
step 1: i←1

step 2: i≥nならば停止
そうでなければ temp←A[i], j←i

step 3: j≥1かつA[j-1]>tempが成り立つ間,
A[j]←A[j-1], j←j-1を繰り返す

step 4: A[j]←temp. i←i+1としてstep 2へ

左方向へ入替ながら探索



最悪時間計算量$\Theta(n^2)$ 逆順にソートされた入力の時

最良時間計算量$\Theta(n)$ ソート済の入力の時

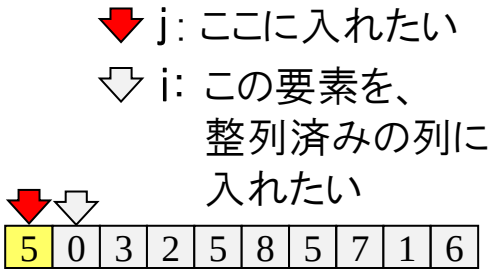
平均時間計算量$\Theta(n^2)$

n: 要素数

挿入ソート（insertion sort）

では、最初から

整列済みの配列に1つずつ要素を挿入する



［アルゴリズム］

step 1: i←1

step 2: i≥nならば停止
そうでなければ temp←A[i], j←i

step 3: j≥1かつA[j-1]>tempが成り立つ間,
A[j]←A[j-1], j←j-1を繰り返す

step 4: A[j]←temp．i←i+1としてstep 2へ

左方向へ入替ながら探索

Step 2～4 で、
ここに入れたい

最悪時間計算量$\Theta(n^2)$ 逆順にソートされた入力の時

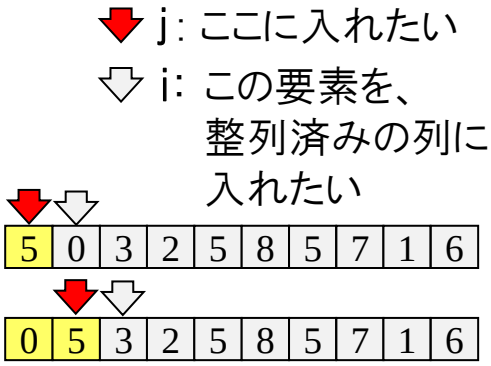
最良時間計算量$\Theta(n)$ ソート済の入力の時

平均時間計算量$\Theta(n^2)$

n: 要素数

挿入ソート（insertion sort）

整列済みの配列に1つずつ要素を挿入する



［アルゴリズム］

step 1: i←1

step 2: i≥nならば停止
そうでなければ temp←A[i], j←i

step 3: j≥1かつA[j-1]>tempが成り立つ間,
A[j]←A[j-1], j←j-1を繰り返す

step 4: A[j]←temp．i←i+1としてstep 2へ

左方向へ入替ながら探索

最悪時間計算量$\Theta(n^2)$ 逆順にソートされた入力の時

最良時間計算量$\Theta(n)$ ソート済の入力の時

平均時間計算量$\Theta(n^2)$

n: 要素数

挿入ソート（insertion sort）

整列済みの配列に1つずつ要素を挿入する

［アルゴリズム］

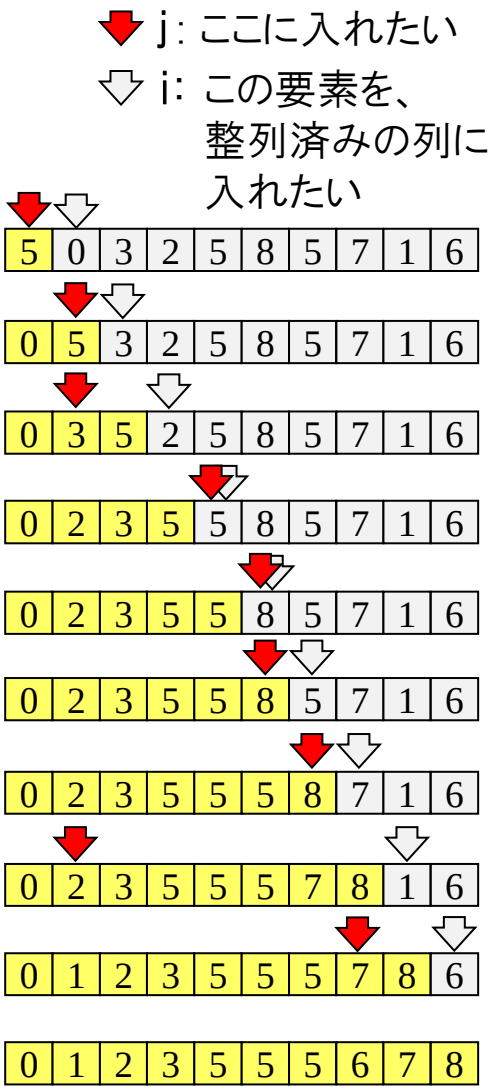
step 1: i←1

step 2: i≥nならば停止
そうでなければ temp←A[i], j←i

step 3: j≥1かつA[j-1]>tempが成り立つ間,
A[j]←A[j-1], j←j-1を繰り返す

step 4: A[j]←temp．i←i+1としてstep 2へ

左方向へ入替ながら探索



最悪時間計算量$\Theta(n^2)$　逆順にソートされた入力の時

最良時間計算量$\Theta(n)$　ソート済の入力の時

平均時間計算量$\Theta(n^2)$

n: 要素数

挿入ソート（insertion sort）

整列済みの配列に1つずつ要素を挿入する

［アルゴリズム］

step 1: i←1

step 2: i≥nならば停止
そうでなければ temp←A[i], j←i

step 3: j≥1かつA[j-1]>tempが成り立つ間,
A[j]←A[j-1], j←j-1を繰り返す

step 4: A[j]←temp．i←i+1としてstep 2へ

Step 3
最悪時：整列済みの列をすべてチェック O(n)
最良時：A[i-1] と temp=A[i] の比較1回ですむ O(1)
この Step 3 を O(n) 回だけ繰り返す

小さい順にしたいのに、
大きい順になっている時

最悪時間計算量$\Theta(n^2)$　逆順にソートされた入力の時

最良時間計算量$\Theta(n)$　ソート済の入力の時

平均時間計算量$\Theta(n^2)$

休憩

- ここで、少し休憩しましょう。
- 深呼吸したり、肩の力を抜いてから、
  次のビデオに進んでください。

バブルソート（bubble sort）

隣り合う2つの要素を比較して，小さい順になっていなければ入れ替えるという操作を，右から左へ繰り返し行う

［アルゴリズム］

step 1: i←1

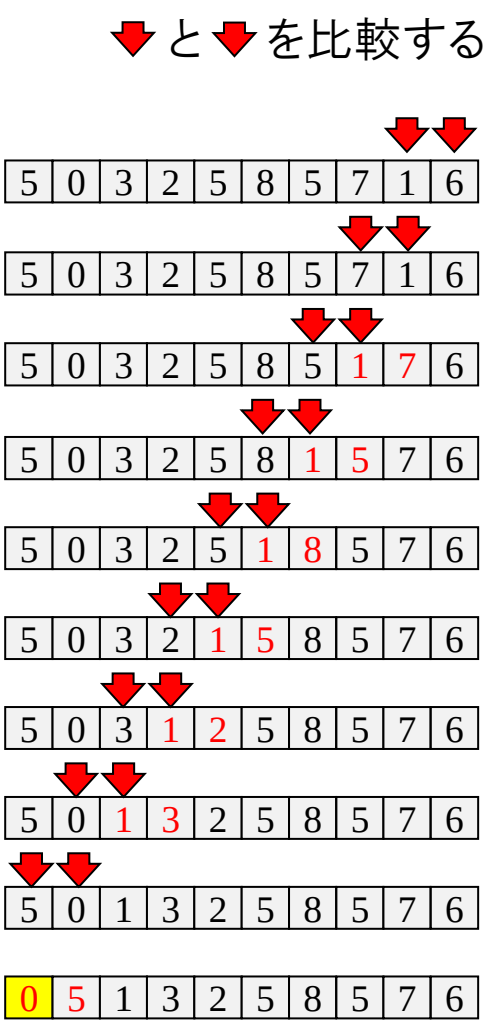
step 2: j=n-1,n-2,…,iの順に次のことを繰り返す
A[j-1]>A[j]ならばA[j-1]とA[j]を入れ替える

step 3: step 2で入れ替えが起こらなかったら停止
そうでなければi←i+1としてstep 2へ

最悪時間計算量$\Theta(n^2)$　逆順にソートされた入力の時

最良時間計算量$\Theta(n)$　ソート済の入力の時

平均時間計算量$\Theta(n^2)$



Step 2 の j のループ1回分でソート済みの部分が1つ増える

バブルソート（bubble sort）

隣り合う2つの要素を比較し
なっていなければ入れ替え
右から左へ繰り返し行う

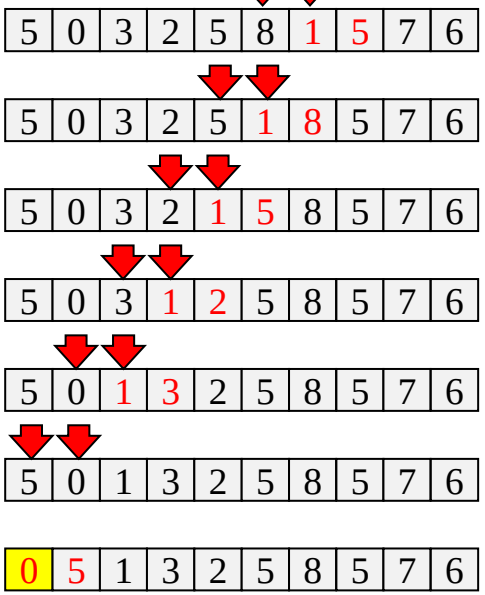
Step 3： A[j-1] > A[j] のチェックを O(n) 回
最悪時：この Step 3 を O(n) 回繰り返す
（ソート済みの部分が1つずつ増える）
最良時：1 回目 (i = 1) の Step 2 のチェックで、
すべての j に対して A[j-1] < A[j] なので、
Step 3 まで来たところで停止
つまり、Step 3 を1回しか実行しない

［アルゴリズム］

step 1: i←1

step 2: j=n-1,n-2,…,iの順に次のことを繰り返す
A[j-1]>A[j]ならばA[j-1]とA[j]を入れ替える

step 3: step 2で入れ替えが起こらなかったら停止
そうでなければi←i+1としてstep 2へ



Step 2 の j のループ1回分で
ソート済みの部分が1つ増える

最悪時間計算量$\Theta(n^2)$　逆順にソートされた入力の時

最良時間計算量$\Theta(n)$　ソート済の入力の時

平均時間計算量$\Theta(n^2)$

シェルソート（shell sort）

シェルさんが
提案しました

- 挿入ソート（or バブルソート）は、
  入力がだいたいソートされていると速い

  最悪時間計算量$\Theta(n^2)$　逆順にソートされた入力の時
  最良時間計算量$\Theta(n)$　ソート済の入力の時

挿入ソートやバブル
ソートの一般化

- 何本かのソート済の列をたばねて、
  それに対して挿入ソート（or バブルソート）を実行する
  （という操作を繰り返す）

シェルソート（shell sort）

発展

等間隔の部分列に挿入ソート（あるいはバブルソート）を適用し，
それを徐々に間隔を小さくしながら繰り返す

挿入ソートやバブルソートの一般化

［アルゴリズム］

// h_1(=1), h_2, … ：自然数の数列（増分列increment sequence）

step 1: i←arg max{ j: h_j<n}

h_j < n を満たす最大の j

step 2: j=0,1,…,h_i-1に対する各部分列A[j],A[j+h_i],A[j+2h_i],…を挿入ソートで整列

step 3: i=1なら停止
そうでないならi←i-1としてstep 2へ

離れた位置をソートすることで高速化を図る

最悪時間計算量 $O(n(\log n)^2)$
最良時間計算量 $O(n \log n)$

増分列h_iの取り方で
時間計算量が異なる

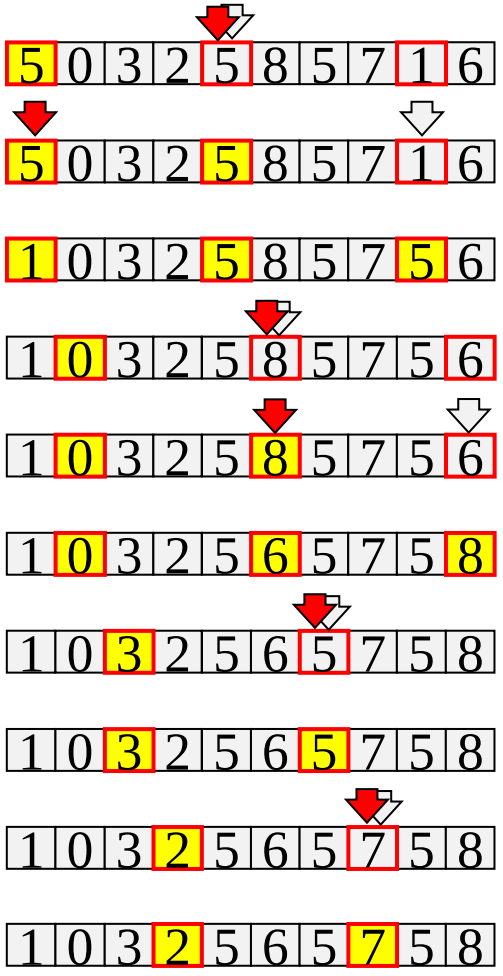
Shell (オリジナル): $1, \dots, \frac{N}{8}, \frac{N}{4}, \frac{N}{2}$ -> $O(n^2)$
Hibbard: $1, 3, 7, \dots, 2^k - 1$ -> $O(n^{1.5})$
Knuth: $1, 4, 13, \dots, \frac{3^k-1}{2}$ -> $O(n^{1.5})$

シェルソートの動き

発展

増分列を $h_{i+1}=3h_i+1$, $h_1=1$ とする　$h_2=4$, $h_3=13$

まず$h_2=4$ 毎の要素をソート



A[0], A[4], A[8] をソート

A[1], A[5], A[9] をソート

A[2], A[6] をソート

A[3], A[7] をソート

:挿入する位置

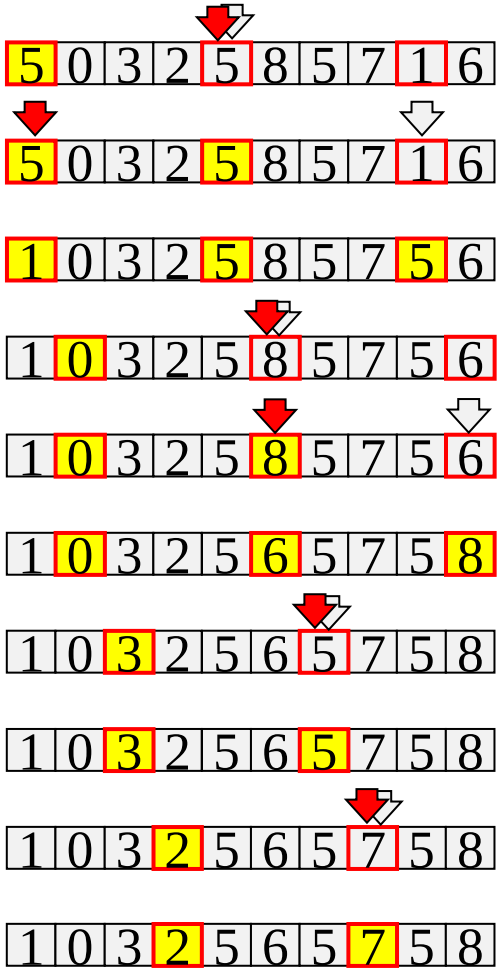
:挿入する要素

シェルソートの動き

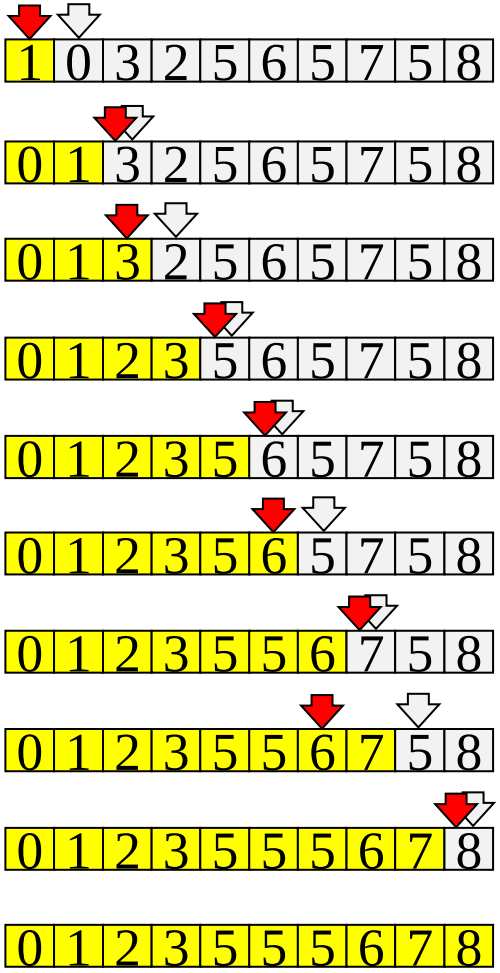
発展

増分列を $h_{i+1}=3h_i+1$, $h_1=1$ とする　$h_2=4$, $h_3=13$

まず$h_2=4$ 毎の要素をソート



次に$h_1=1$ 毎の要素をソート



:挿入する位置

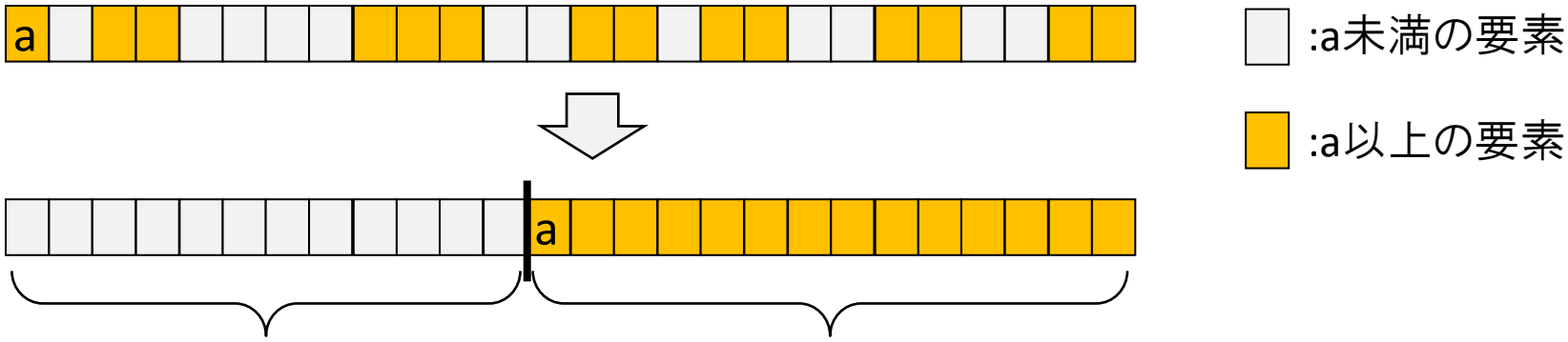
:挿入する要素

休憩

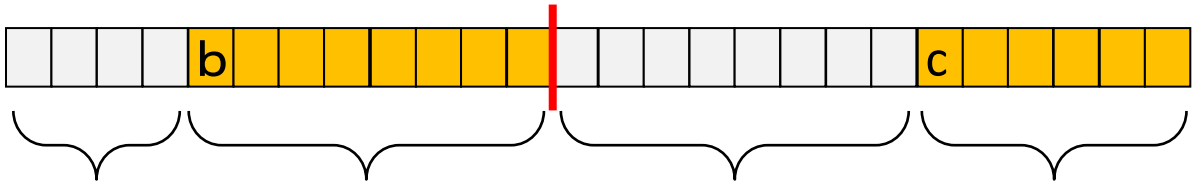
- ここで、少し休憩しましょう。
- 深呼吸したり、肩の力を抜いてから、
  次のビデオに進んでください。

クイックソート（quick sort）

データをある値（軸要素の値）以上のものと以下（or未満）のものに分けることを再帰的に行う、**分割統治法**による整列アルゴリズム



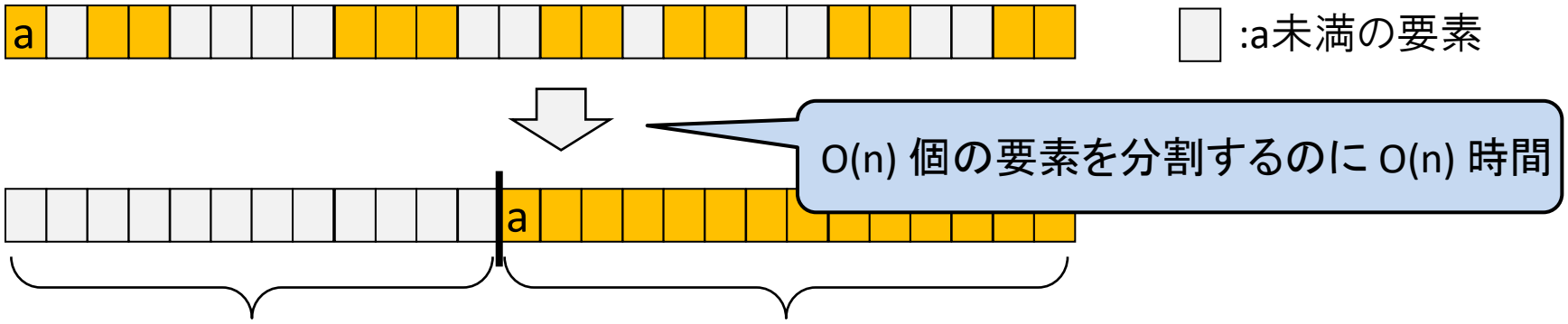
それぞれの部分に対し，新たな軸要素を用いて同じ操作を行う



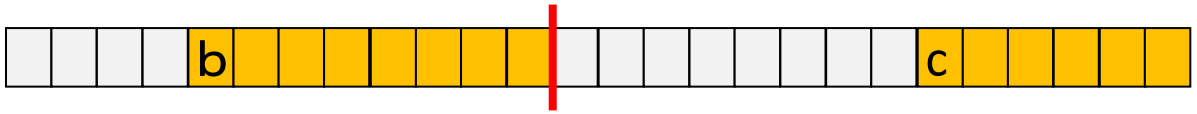
・各部分が要素数が 1 になるまで同じ操作
・要素数 1 の部分は、それ以上分割しない
（この部分はソート済となる）

クイックソート（quick sort）

データをある値（軸要素の値）以上のものと以下（or未満）のものに分けることを再帰的に行う、**分割統治法**による整列アルゴリズム



それぞれの部分に対し，新たな軸要素を用いて同じ操作を行う



最悪時間計算量 $\Theta(n^2)$ 逆順にソートされた入力の時

最良時間計算量 $\Theta(n \log n)$

平均時間計算量 $\Theta(n \log n)$

毎回、半分ずつに分割できれば、
$O(\log n)$ 回で要素数 1 まで分割できる

分割統治法（divide-and-conquer method）

重要

- 大きな問題に対して，次のようにして解を求める方法のこと
  1. **部分問題に分割する**
  2. **各部分問題を解く**
  3. **各部分問題の解を統合する**
- 部分問題を解くとき，さらに分割統治法を用いて再帰的に問題を小さくしていくことができる
- **問題が十分小さければ，自明な方法で解を決定できることが多い**

発展 ただし，問題を小さくした際に，同じ部分問題が何度も現れる場合があり，そのときは計算量が非常に大きくなってしまうこともある
それに対する対処法として，一度解いたことのある部分問題の解を記憶すること（**メモ化**）で解決できる場合もある

競技プログラミングとかで、よく使われる

クイックソートの動作例

quicksort(A,i,j),: A[i],A[i+1],…,A[j]を整列する

step 1: i≥jならば何もしないでリターン

step 2: a←A[i]

step 3: 要素の並べ替えを行い，以下のように
グループ分割する.

A[i],…,A[ℓ-1]: a以下の要素

A[r+1],…,A[j]: a以上の要素

step 4: quicksort(A,i,ℓ-1)とquicksort(A,r+1,j)を実行

// グループ分割の手順

step 1: ℓ←i, r←j

step 2: A[ℓ]<aの間 ℓ← ℓ+1 を繰り返す

step 3: A[r]>aの間 r←r-1 を繰り返す

step 4: ℓ≥rであれば停止
そうでなければA[ℓ]とA[r]を入れ替える

step 5: ℓ←ℓ+1, r←r-1としてstep 2 へ

5	0	3	2	5	8	5	7	1	6

クイックソートの動作例

quicksort(A,i,j),: A[i],A[i+1],…,A[j]を整列する

step 1: i≥jならば何もしないでリターン

step 2: a←A[i]

step 3: 要素の並べ替えを行い，以下のように
グループ分割する.

A[i],…,A[ℓ-1]: a以下の要素

A[r+1],…,A[j]: a以上の要素

step 4: quicksort(A,i,ℓ-1)とquicksort(A,r+1,j)を実行

// グループ分割の手順

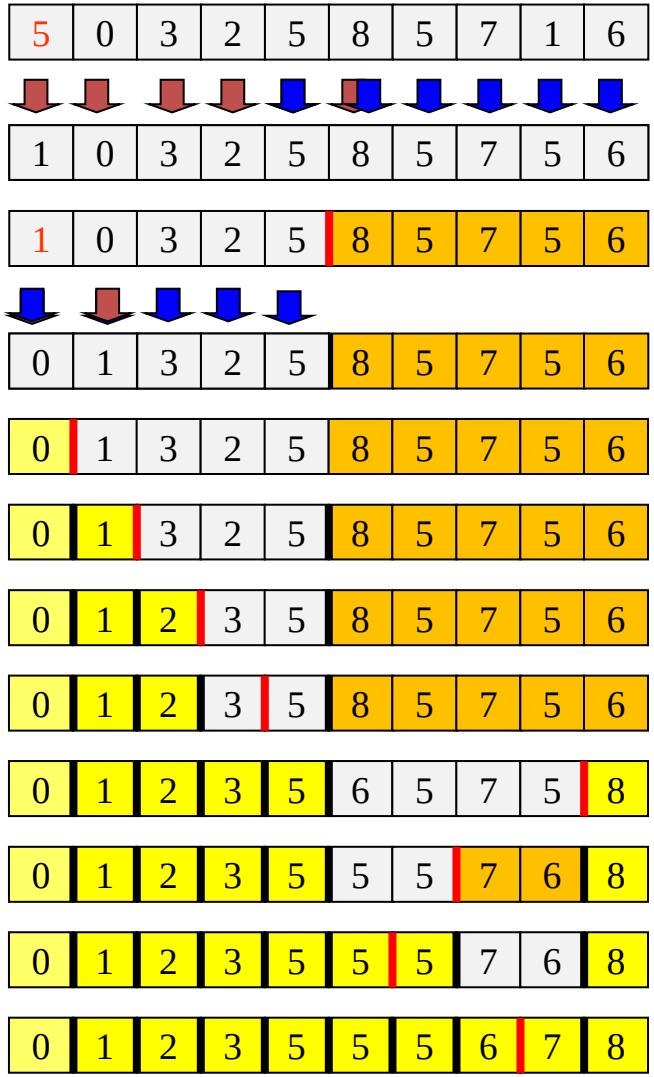
step 1: ℓ←i, r←j

step 2: A[ℓ]<aの間 ℓ← ℓ+1 を繰り返す

step 3: A[r]>aの間 r←r-1 を繰り返す

step 4: ℓ≥rであれば停止
そうでなければA[ℓ]とA[r]を入れ替える

step 5: ℓ←ℓ+1, r←r-1としてstep 2 へ



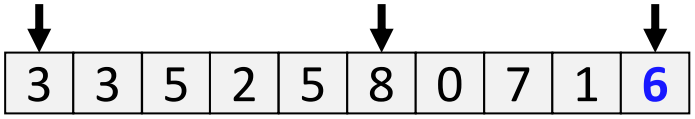
軸要素の選び方

クイックソートにおいては，**軸要素の選び方が処理時間に影響する**

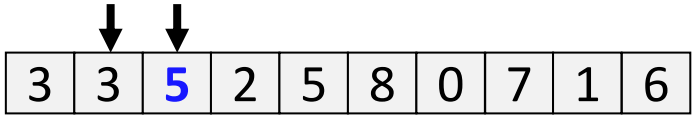
［軸要素の選び方］

案1. 左端の要素
案2. ランダムに選んだ位置の要素
案3. 左端，中央，右端の要素の中央値の要素
案4. 左からみて最初に得られた2つの異なる値の大きい方の要素

案3の選び方



案4の選び方



※ 案 1～3 の選び方だと最小値が選ばれる可能性があり，その場合は未満と以上の分け方だとうまくいかない．よって，アルゴリズムを，軸要素の値以上と以下に分けるように変える必要がある（軸要素の値はどちらに含まれてもよい）

平均時間計算量 $O(n \log n)$ の証明

発展

［仮定］全ての要素は値が異なる.
　　入力される要素の順列は一様分布により発生する.

［証明］$T(n)$ を n 要素のクイックソートに要する平均時間とする.
n 要素のグループ分割に必要な時間計算量は明らかに$O(n)$.
したがって十分大きな定数 C_0 に対し，グループ分割の計算時間を$C_0 n$ で上から抑えることができる. 今，i 番目に大きい要素を軸に選んだとする. $i = 1$ のときは配列は1個と$n-1$個に，その他の場合は$i-1$個と$n-i+1$個に分割される. i 番目の要素を選択する確率は仮定より $1/n$ なので,

$$T(n) \leq \frac{1}{n}\left(T(1) + T(n-1) + C_0 n + \sum_{i=2}^{n} \left(T(i-1) + T(n-i+1) + C_0 n\right)\right).$$

iと無関係なので∑の外に出て$C_0 n(n-1)$

$\sum T(i') = \sum T(n-i')$

$i' = i - 1$

$T(1)$ は定数時間なので，十分大きな C をとれば,

$$T(n) \leq \frac{2}{n}\sum_{i=1}^{n-1} T(i) + \frac{1}{n} T(n-1) + Cn.$$

証明のつづき

発展

$n \geq 2$ のとき，適当な定数 d を用いて $T(n) \leq dn\log_2 n$ が成り立つことを数学的帰納法で示す．

いま，$d = \max\left\{\frac{T(2)}{2}, 8C\right\}$ とおくと，$n = 2$ のとき，

$$T(2) \leq 2d = d \cdot 2 \cdot \log_2 2$$

$d \geq \frac{T(2)}{2}$

$\log_2 2 = 1$

より成り立つ．$2 \leq i < n$ に対して，$T(i) \leq d\, i \log_2 i$ が成り立っていると仮定する．このとき，

$$T(n) \leq \frac{2d}{n}\sum_{i=1}^{n-1} i\log_2 i + \frac{d}{n}(n-1)\log_2(n-1) + Cn$$

前のページの式から

$$\leq \frac{2d}{n}\left(\sum_{i=1}^{\lfloor n/2 \rfloor} i\log_2\lfloor n/2 \rfloor + \sum_{i=\lfloor n/2 \rfloor+1}^{n-1} i\log_2 n\right) + d\log_2 n + Cn.$$

和を分割して$\log i$を$\log n$に

$\log_2 i > 0$

n が偶数のとき　$T(n) \leq dn\log_2 n - dn/4 - d/2 + Cn$

n が奇数のとき　$T(n) \leq dn\log_2 n - dn/4 + d/4n + Cn$ が示せる．

$n = 2m + 1$ として展開！

$d \geq 8C$ であるから，いずれの場合も$T(n) \leq dn\log_2 n$ が成立する．

よって，$T(n) = O(n\log n)$ である．

赤字の部分はマイナス

証明のつづき（うまくいかない版）

発展

$n \geq 2$ のとき，適当な定数 d を用いて $T(n) \leq dn \log_2 n$ が成り立つことを数学的帰納法で示す．いま$d \geq T(2)/2$ とおくと，$n = 2$ のとき，

$$T(2) \leq 2d = d \cdot 2 \cdot \log_2 2$$

となり成り立つ．$2 \leq i < n$ に対して，$T(i) \leq d\, i \log_2 i$ が成り立っていると仮定する．このとき，

$$\begin{aligned} T(n) &\leq \frac{2}{n}\sum_{i=1}^{n-1} T(i) + \frac{1}{n}T(n-1) + Cn \\ &\leq \frac{2}{n}\sum_{i=1}^{n-1} (d\, i \log_2 i) + \frac{d}{n}(n-1)\log_2(n-1) + Cn \\ &\leq \frac{2d}{n} \cdot \frac{(n-1)n}{2} \cdot \log_2 n + d \log_2 n + Cn \\ &\leq dn \log_2 n + Cn. \end{aligned}$$

Cnの項が消えてくれない

今日のまとめ

- 整列（ソート, sorting）
- 整列アルゴリズムの種類と特徴

- $O(n^2)$ 時間の整列アルゴリズム
  - 選択ソート、挿入ソート、バブルソート
- $O(n(\log n)^2)$ 時間の整列アルゴリズム
  - シェルソート（挿入ソートやバブルソートの一般化）
- 平均時$O(n \log n)$ 時間の整列アルゴリズム
  - クイックソート（分割統治法によるソート）
  - 平均時間計算量の証明は、結構むずかしい！