KYOTO UNIVERSITY

アルゴリズムとデータ構造③
～ ソートとヒープ ～

鹿島久嗣
（計算機科学コース）

DEPARTMENT OF INTELLIGENCE SCIENCE
AND TECHNOLOGY

整列（ソート）のアルゴリズム

整列問題（ソート）：
要素を小さい順に並び替える問題

- 整列問題（sorting）
  - 入力：n個の数 $a_1, a_2, \ldots, a_n$ が入った配列
  - 出力：$a_{1'} \leqq a_{2'} \leqq \ldots \leqq a_{n'}$ を満たす入力列の置換
- 例：入力 $(4, 5, 2, 1) \to$ 出力 $(1, 2, 4, 5)$
- ソートのアルゴリズム
  - 2つの数（要素）の比較：2つの要素のどちらかが大きいか、あるいは等しいことが1ステップで判定できる
  - できるだけ少ない比較回数で並べかえを完了したい

単純なソートアルゴリズム：
ソート済み領域を左から順に拡大していく

- ある時点において、現在の位置よりも左の部分は整列済みとする

1	3	8	12	5	4	6	9

整列済み　　現在の位置

- 現在の位置から左に見ていき、順序が保たれるところまで移動する

1	3	5	8	12	4	6	9

整列済みの領域がひとつ拡大された

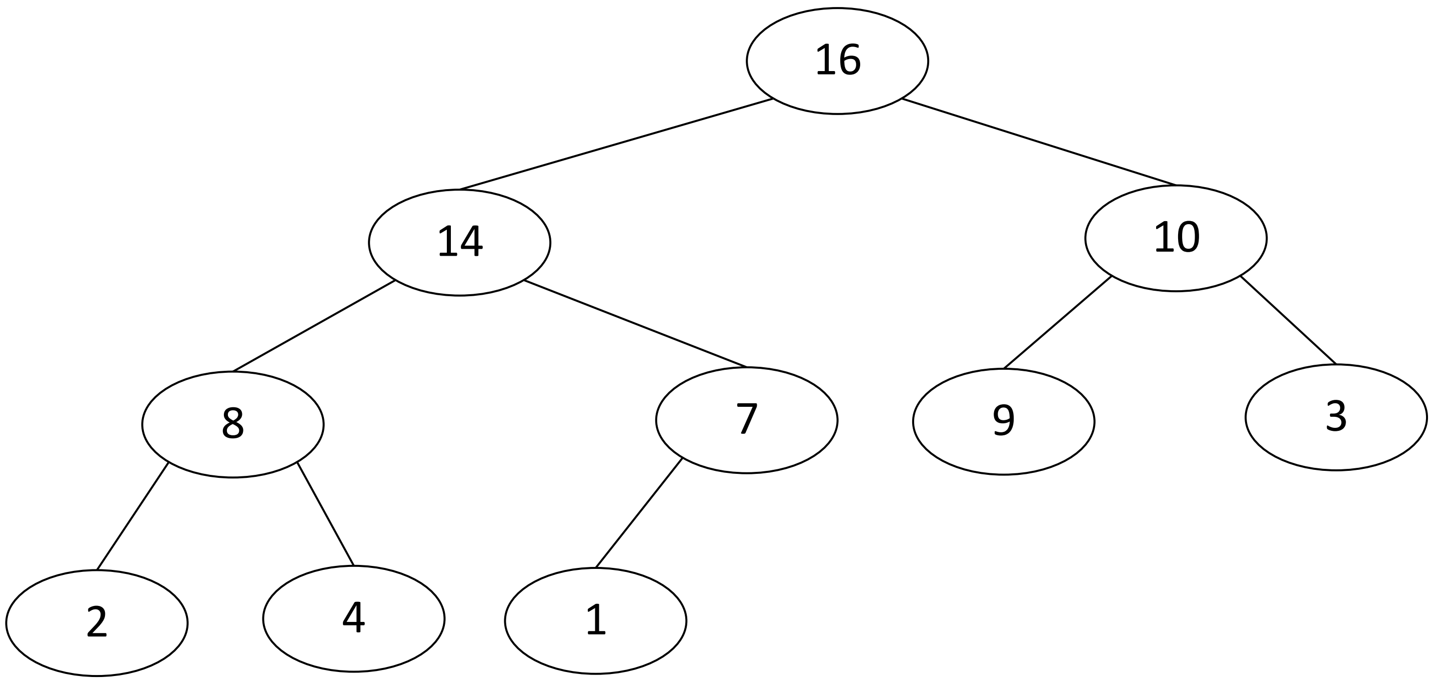
単純なソートアルゴリズムの計算量：
計算効率はそれほど良くないが省スペースで実行可能

- 「現在の位置から左に見ていき、順序が保たれるところまで移動する」アルゴリズムを考える
  - 現在の位置をjとすると、この操作には$O(j)$回の比較・交換が必要（最悪ケースでは先頭まで到達）
  - これを $j = 1,2,..,n$ まで行うと、総比較回数は $\Sigma_{j=1,\ldots,n} O(j) = O(n^2)$ になる
- $O(n^2)$のソートアルゴリズムはあまり効率はよくない
  - 最も効率の良いアルゴリズムは $O(n \log n)$ （後述）
  - ただし、「その場でのソート」が可能なので省スペース
    - 入力配列以外に定数個の領域しか使用しない

ヒープソート

ヒープソート：データ構造「ヒープ」を使った$O(n \log n)$のソート法

- 「ヒープ」とよばれるデータ構造の一種を用いたソート法
- $O(n \log n)$ で動く「その場での」ソート法
  - $O(n \log n)$はソートの最悪計算量としてはベスト

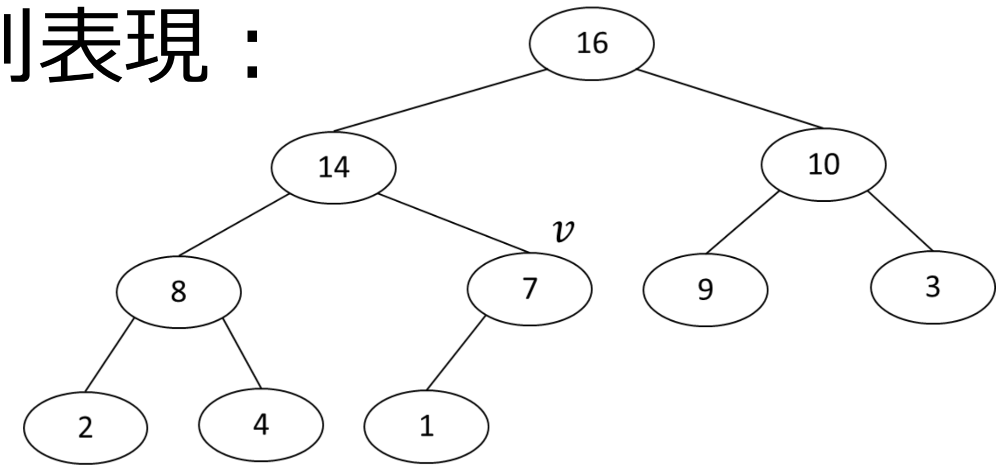


ヒープ：
ヒープ条件をみたす、ほぼ完全2分木のデータ構造

- ヒープは、ほぼ完全2分木である
  - 2分木：全頂点の子数が最大2個の根付き木
  - 完全2分木：葉以外の頂点の子がちょうど2個で、すべての葉の高さが等しい2分木
- ヒープの各頂点はデータをひとつずつもち、必ず「ヒープ条件」を満たしていなければならない
  - ヒープ条件：任意の頂点iのデータの値は、その親のもつデータの値以下である
$$A[\mathrm{parent}(i)] \geq A[i]$$
- n頂点をもつヒープの高さは $\Theta(\log n)$

ヒープの表現：
ヒープは配列で一意に表現できる

- ヒープと等価な配列表現：



$\Rightarrow$ A

16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

- 配列表現の性質：
  - 頂点iの左の子は$2i$番目、右の子は$2i+1$番目
  - 頂点iの親は$\lfloor i/2 \rfloor$番目に入っている

ヒープソート：
「根の値の取出し」と「木の更新」を繰り返してソートを実行

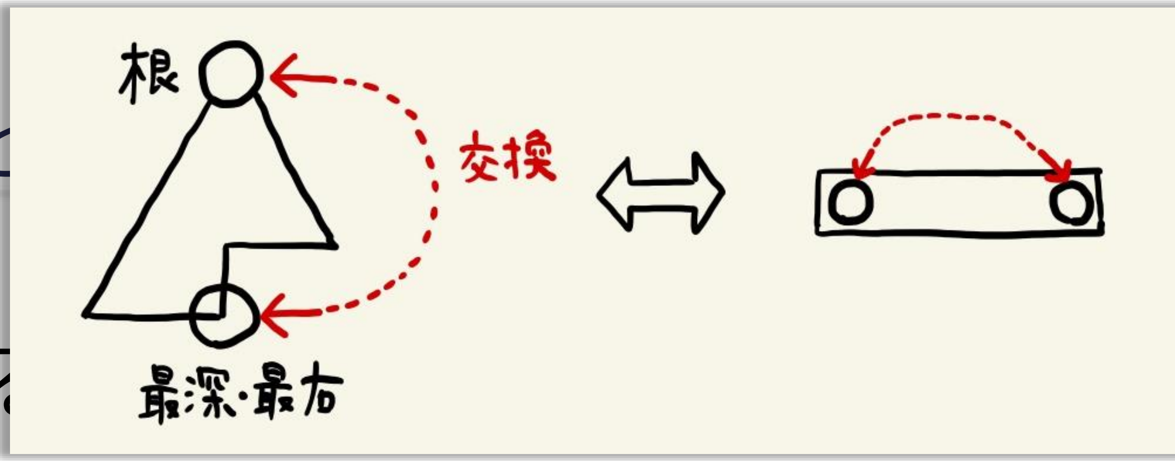
- （定義より）ヒープの根には最大の値が入っている
- 「ヒープの根の取り出し」を繰り返せば、要素を大きい順に取り出せるはず
  - これらを逆順（小さい順）に並べ直せば、ソートが完了
- ただし、「ヒープの根の取り出し」はヒープ構造を壊すため、取り出す度に、これを修復、すなわち「木の更新」を行う必要がある

ヒープソート：「根の値の取出し」と「木の更新」を繰り返してソートを実行

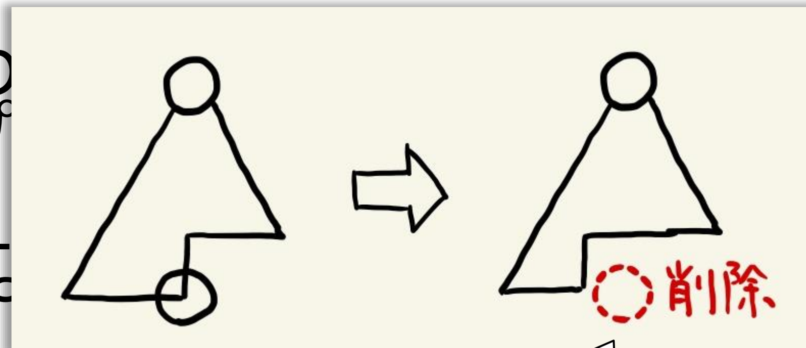
- 大まかな戦略：
  1. ヒープを構成する（$O(n)$：後述）
  2. 根と、最も深く、最も右にある頂点（＝配列表現の場合は一番最後の要素）と交換する
  3. 木（＝配列）のサイズをひとつ小さくする
  4. 根が入れ替わったことでヒープ条件が満たされなくなっているので、ヒープを更新（$O(\log n)$）する
  5. 以上を頂点がなくなるまで繰り返す（→ステップ2）

ヒープソート：
「根の値の取出し」と…行

- 大まかな戦略：



1. ヒープを構成する
2. **根と、最も深く、最も右にある頂点（＝配列表現の場合は一番最後の要素）と交換する**
3. **木（＝配列）のサイズをひとつ小さくする**
4. 根が入れ替わっ…たされなくなっているので、ヒープ…る
5. 以上を頂点がな…ステップ2）



ソート済みの要素として確定するということ

ヒープソート：「根の値の取出し」と「木の更新」を繰り返してソートを実行

- 大まかな戦略：
  1. ヒープを構成する（ $O(n)$ ：後述 ）
  2. 根と、最も深く、最も右にある頂点（＝配列表現の場合は一番最後の要素）と交換する
  3. 木（＝配列）のサイズをひとつ小さくする
  4. **根が入れ替わったことでヒープ条件が満たされなくなっているので、ヒープを更新（ $O(\log n)$ ）する**
  5. 以上を頂点がなくなるまで繰り返す（→ステップ2）

根のヒープ条件の回復：根から下に辿り$O(\log n)$でヒープ条件を回復

- 以下の「HEAPIFY(A, i)」関数を考える：
  - 配列A（を木としてみたときの）の頂点i以下の頂点をヒープ条件を満たすように更新する関数
  - ただし、頂点iの２つの子を根とする部分木はすでにヒープ条件を満たしているとする
    - 今回、変更されたのは根だけなので、この条件が成立
- HEAPIFY(A, i)関数は、自身を再帰的に呼び出しながら、木の上から下へ向かって降りていく
  - $O(\log n)$で葉に到達する

根のヒープ条件の回復（詳細）
根から下に辿り$O(\log n)$でヒープ条件を回復

$\mathrm{HEAPIFY}(A, i)$

1. i からスタート
2. i とその左右の子を比較
   - if i が最大 then 終了
   - else
     - 大きい方をi と入れ替える
     - $i \leftarrow$入れ替えられた先の位置
     - $\mathrm{HEAPIFY}(A, i)$：自分自身を呼ぶ

i と２つの子の間のヒープ条件が満たされる

新しいi とその子の間のヒープ条件の成立はまだ不明

- 計算量はiの高さをhとして$O(h) \le O(\log n)$

ヒープの構成：
木の下方から上方に向かって構成する

- 手続き：木の下から上に向かって（ヒープになっていない）木（＝配列）をヒープにする

- BUILD_HEAP(A)

子のある頂点を添え字の大きいほうから順に

1. for $i \leftarrow \lfloor \mathrm{length}(A)/2 \rfloor$ down to 1
2. do HEAPIFY(A, i)
3. end for

i番目の頂点を根とする部分木がヒープ条件を満たすように更新する

- HEAPIFYが$\mathrm{O}(\log n)$ステップ、これを$\mathrm{O}(n)$回呼び出すので全体としては$\mathrm{O}(n \log n)$の計算量
  – 実は、注意深く評価すると$\mathrm{O}(n)$（※ あとで示す）

ヒープへの挿入：$O(\log n)$で実行可能

- ヒープに新たなデータxを挿入する

$\mathrm{HEAP\_INSERT}(A, x)$

1. 配列A の最後にxを付け加える
2. xとparent(x)を比較する
   - if $x \leq \mathrm{parent}(x)$ then 終了
   - else xとparent(x) を入れ替える
3. $x \leftarrow \mathrm{parent}(x)$
4. go to 2

ヒープ条件の確保

繰り返し回数は
$O(\log n)$

- これを繰り返すことでヒープ構成も可能$O(n \log n)$

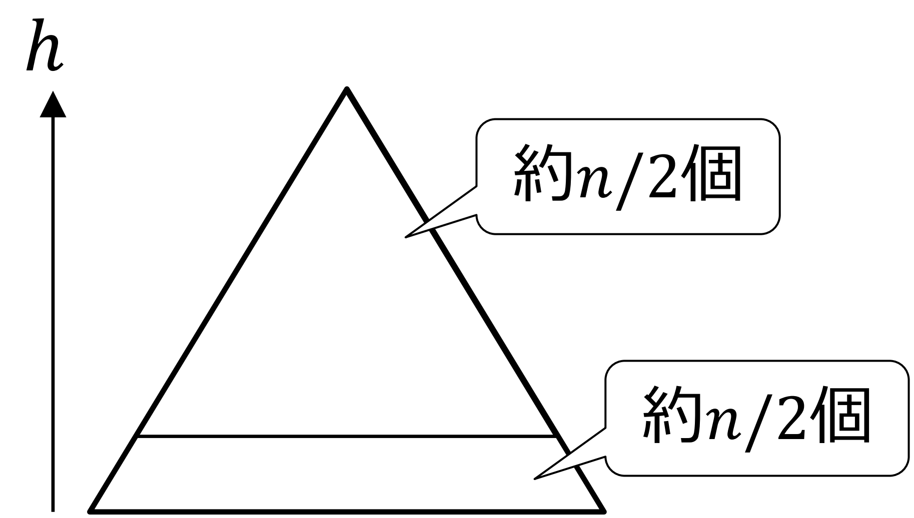
ヒープ構成の計算量：挿入の繰り返しでも構成可能だが遅くなる

- HEAPIFYとHEAP_INSERTのどちらもヒープを構成可能：
  - HEAPIFYは上から下に向かってヒープ条件を回復
  - HEAP_INSERTは下から上に向かってヒープ条件を回復
- 計算量は異なる：
  - HEAPIFYを使った構成は$O(n)$（後述）
  - HEAP_INSERTは$O(n \log n)$
  - 計算量の差はどこからくるか？：
    - 2分木では、木の下方の頂点数が多い
    - ほとんどの頂点にとって 根からの距離 > 葉への距離

根より葉に近い頂点が多い

ヒープ構成の計算量：
HEAPIFYなら線形時間でヒープを構成可能

- 高さhの位置に約 $n/2^h$個の頂点がある
  - 一番下の段にほぼ半分が
  - 次の段には、残りのうちほぼ半分が
  - ・・・
- $\sum_{h=1}^{\log n} h \cdot \frac{n}{2^h} = 2n$ なので$O(n)$



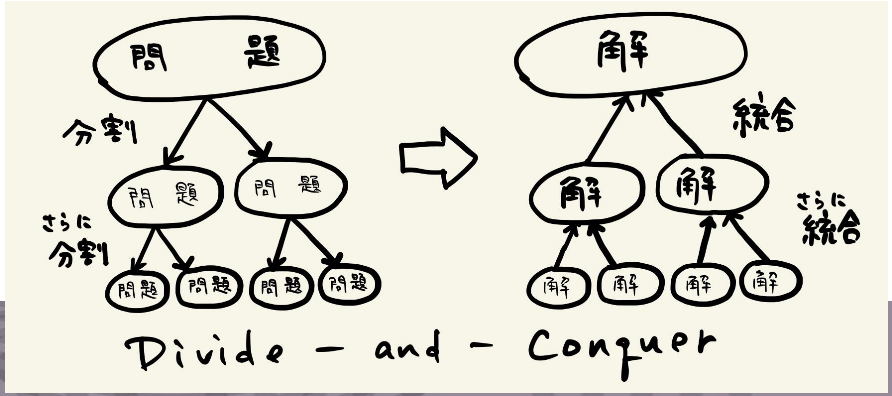
ヒープの応用：
プライオリティ・キュー

- 優先度順にオブジェクトを取り出す仕組み
- 計算機のジョブ割り当て：
  - ジョブが終了 or 割り込み → 最大優先度のものを取り出す
  - 新しいジョブはINSERT
- シミュレーション：
  - 優先度 = 時間として、時刻順にイベントを取り出す

分割統治法

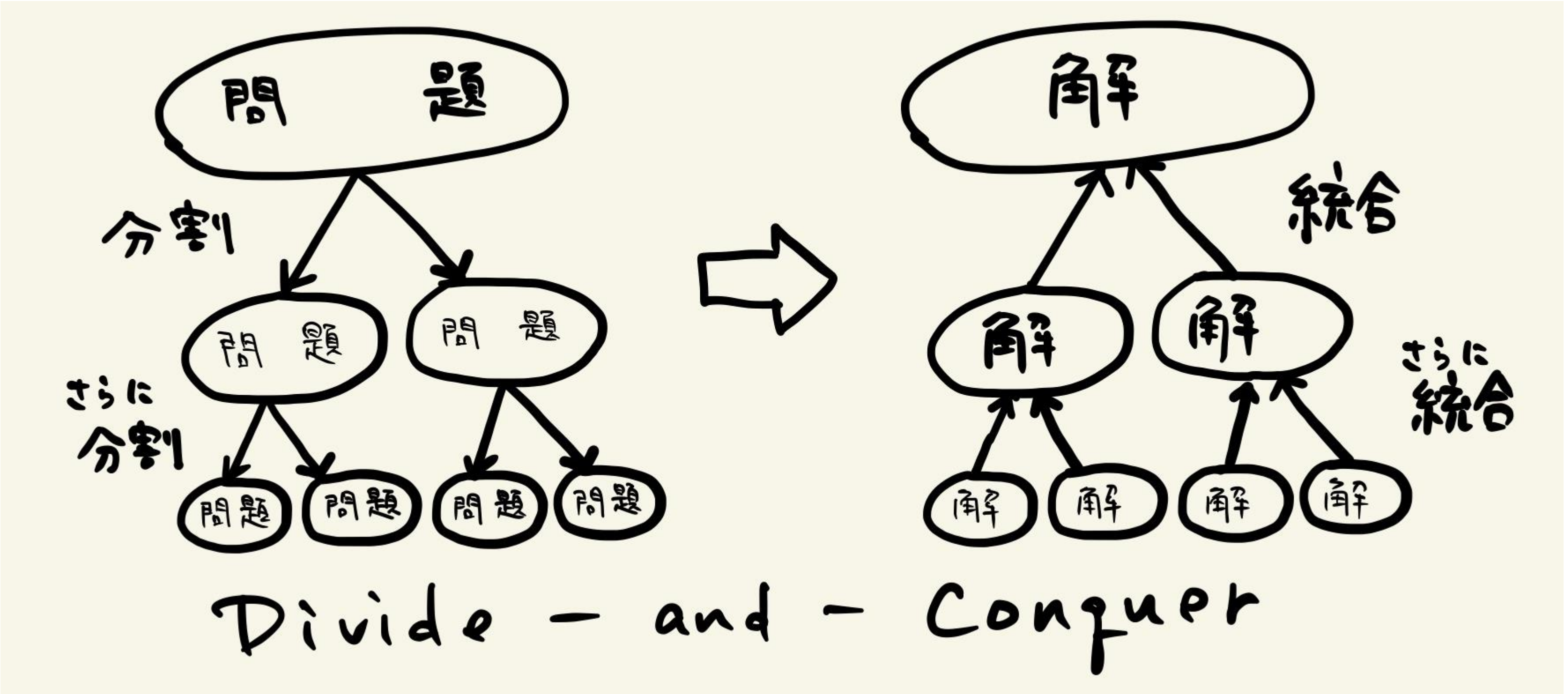
分割統治法：
アルゴリズム設計指針の1つで、問題を小問題に分割して解く

- 特定の問題に対するアドホックな個別の解法ではなく、多くの問題に適用可能なアルゴリズムの一般的な設計指針
  - 分割統治法、動的計画法、…
- 分割統治法：
  - 元の問題を、同じ構造をもった小さな問題に分割
  - 小さな問題の解を統合して元の問題の解を得る



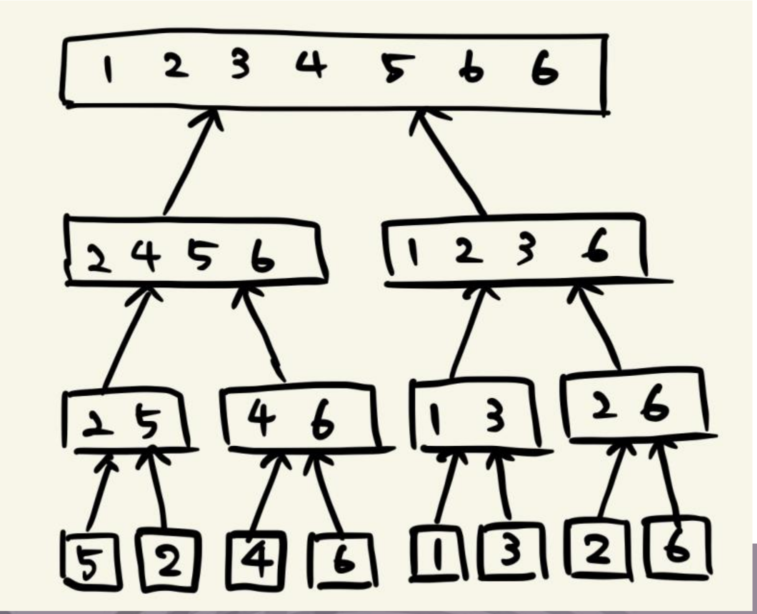
分割統治法：
アルゴリズム設計指針の1つで、問題を小問題に分割して解く

- 分割統治法（Divide-and-conquer）：
  - 分割：元の問題を、同じ構造をもった小さな問題に分割
  - 統合：小さな問題の解を統合して元の問題の解を得る



分割統治法の例：
マージソート

- 入力された配列を前後に分割し、それぞれに対してマージソートを適用する
  - 再帰的に行うことで、サイズ1の配列まで到達する
  - 逆向きに統合して解を構成する
    - 例：配列 (5,2,4,6,1,3,2,6)→ (5,2,4,6) と (1,3,2,6)



マージソートの計算量：
$O(n\log n)$ はソートの計算量としては最良

- $n = 2^k$として$O(n\log n)$で計算できる
  - 実用的には次に紹介するクイックソートが速い
  - オーダー評価では最良
- 計算量評価の再帰式：

$$T(n) = \begin{cases} O(1) & (n = 1) \\ \underbrace{2T(n/2)}_{\text{再帰}} + \underbrace{O(n)}_{\text{統合}} & (n \geq 2) \end{cases} = O(n\log n)$$

マージソート：
マージソートの計算量は$O(n\log n)$

- 計算量評価の再帰式：

$$T(n)=\begin{cases} O(1) & (n=1) \\ 2T(n/2)+O(n) & (n\geq 2) \end{cases}$$

- $T(n)=2T(n/2)+cn=2\left(T\left(\frac{n}{2^2}\right)+c\frac{n}{2}\right)+cn$

$$=2\left(2\left(\cdots\left(2\left(\underbrace{T\left(\frac{n}{2^k}\right)}_{c}+c\frac{n}{2^{k-1}}\right)+c\frac{n}{2^{k-2}}\right)\cdots\right)+c\frac{n}{2}\right)+cn$$

$$=c2^k+\underbrace{cn+\cdots+cn}_{k}=O(n\log n)$$

分類定理（簡易版）：
計算量の再帰式から計算量を導く定理

- $T(n)$の漸化式から$T(n)$のオーダーを導く
- 定理：大きさnの問題を大きさ$\frac{n}{b}$の問題 a 個に分割した
  - つまり、$T(n) = \begin{cases} c & (n = 1) \\ aT\left(\frac{n}{b}\right) + cn & (n \geq 2) \end{cases}$
  - このとき：$T(n) = \begin{cases} O(n) & (a < b) \\ O(n \log n) & (a = b) \\ O\left(n^{\log_b a}\right) & (a > b) \end{cases}$