

# 3. ソート問題とアルゴリズム

3-1. ソート問題について

3-2. 簡単なソートアルゴリズム

3-3. 高度なソートアルゴリズム

3-4. 比較によらないソートアルゴリズム

3-5. ソート問題の下界(高速化の限界)

## 3-1:ソート問題

- 入力: データ数 $n$ と $n$ 個の数

$$n, a_0, a_1, \dots, a_{n-1}$$

(ここで、入力サイズは、 $n$  とします。)

- 出力:

$a_0, a_1, \dots, a_{n-1}$  を小さい順にならべたもの

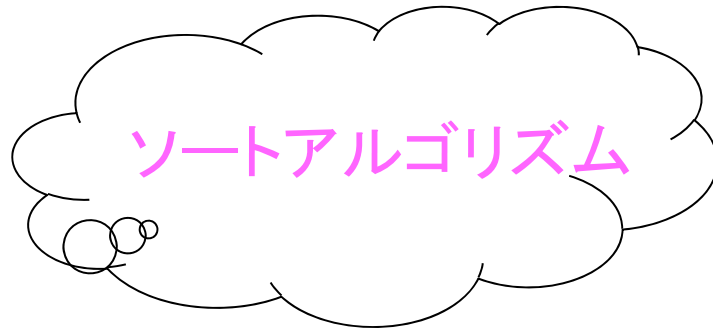
$$a'_0, a'_1, \dots, a'_{n-1} (a'_0 \leq a'_1 \leq \dots \leq a'_{n-1})$$

ここで、 $(a'_0, a'_1, \dots, a'_{n-1})$  は、  
 $(a_0, a_1, \dots, a_{n-1})$  の置換

# ソートの問題例 (問題インスタンス)

データ

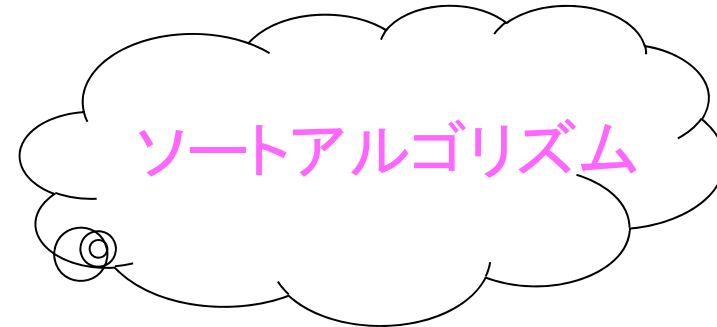
5, 3, 8, 1, 6, 21, 11



1, 3, 5, 6, 8, 11, 21

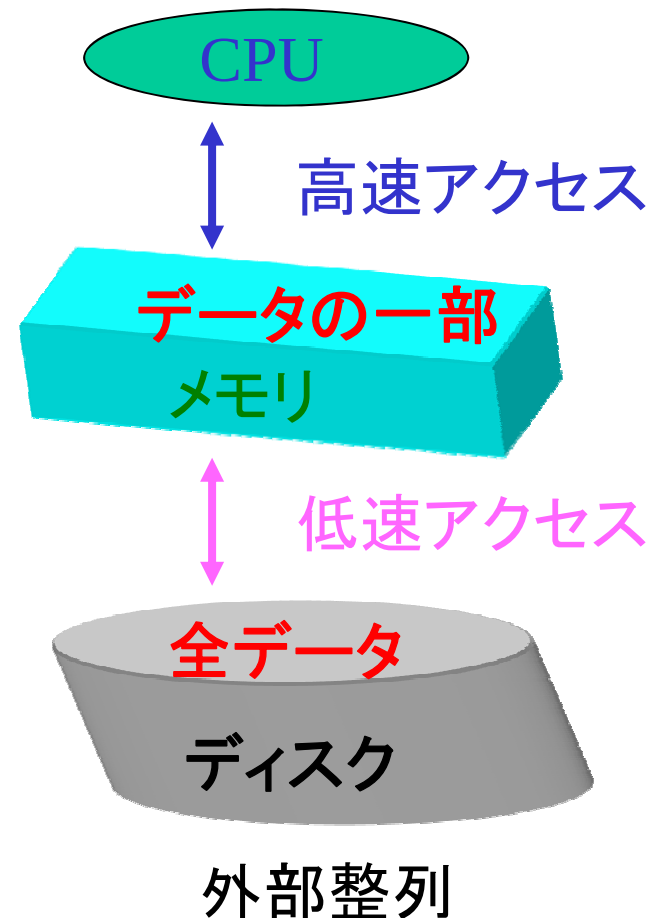
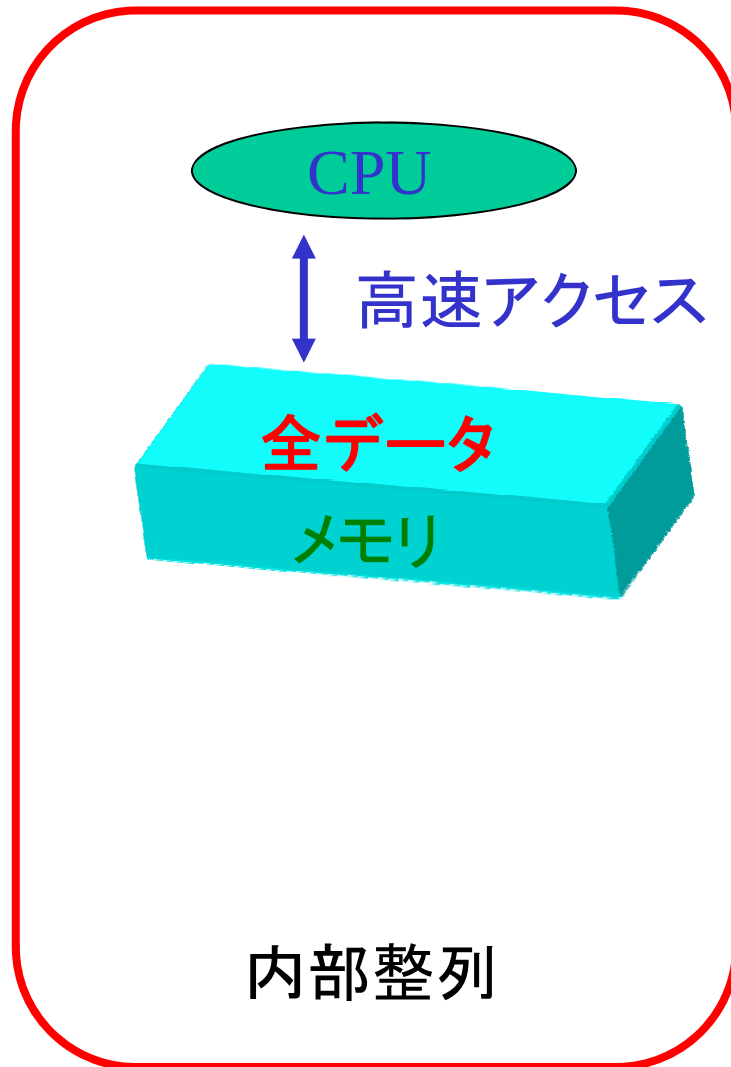
データ

k,a,l,c,d,s



a,c,d,k,l,s

# 内部整列と外部整列



## 仮定と要求

### 内部整列

- どのデータにも均等な時間でアクセスできる。  
できるだけ高速に整列したい。

(理想的な計算機上のアルゴリズムではこっち)

### 外部整列

- CPUーメモリ間のデータ転送速度より、  
ディスクーメモリ間のデータ転送速度が極端に遅い。  
全体の整列をできるだけ高速にしたい。  
(ディスクーメモリ間のデータ転送をあまり行わないように  
する。現実的な問題だが、より複雑な解析が必要である。)

# ソート問題の重要性

- 実際に頻繁に利用される。
- アルゴリズム開発の縮図
  - 繰り返しアルゴリズム(バブルソート、挿入ソート等)
  - アルゴリズムの組み合わせ(選択ソート、マージソート等)
  - 分割統治法(マージソート、クイックソート等)
  - データ構造の利用(ヒープソート、2分探索木等)
- 十分な理論解析が可能。
  - 最悪計算量と平均計算量の違い(クイックソート)
- 豊富なアイデア

# ソートアルゴリズムの種類

バブルソート

選択ソート

挿入ソート

クイックソート

マージソート

ヒープソート

バケットソート

基数ソート

# ソートアルゴリズムの分類

原理

比較による

比較によらない

時間量(速度)

$O(n^2)$

バブルソート  
選択ソート  
挿入ソート

クイックソート

$O(n \log n)$

ヒープソート  
マージソート

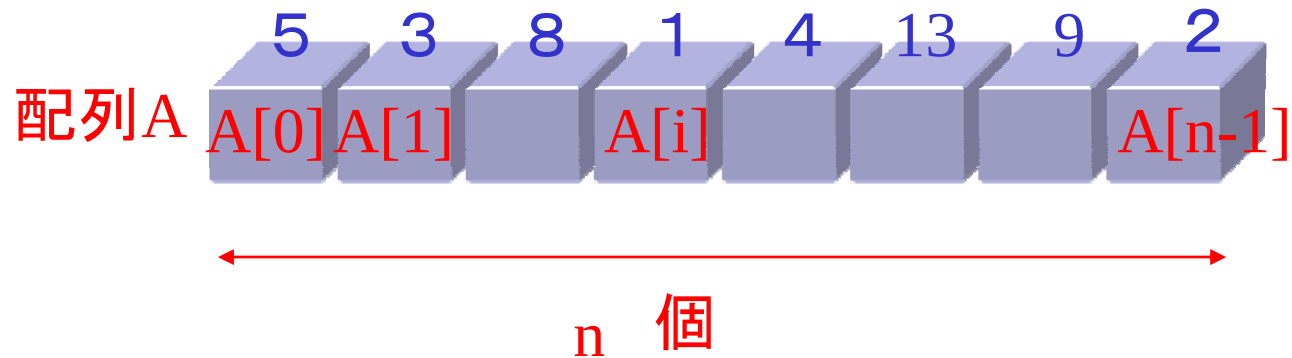
バケットソート  
基数ソート

計算量は  $O(n)$   
だけど条件付き

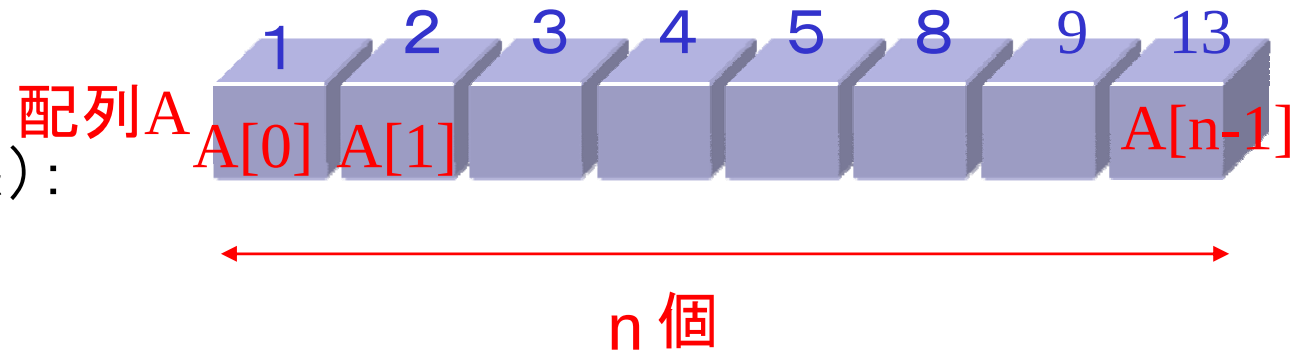


# 入出力形態

入力:



出力  
(終了状態):



入力は配列で与えられるとする。

# 交換関数(準備)

参照渡しにする  
必要があること  
に注意すること。

```
/* 交換用の関数。  
   swap(&a,&b)で呼び出す。  
*/  
1. void swap(double *a,double *b)  
2. {  
3.     double    tmp; /* データの一次保存用 */  
4.  
5.     tmp = *a;  
6.     *a = *b;  
7.     *b = tmp;  
8.  
9.     return;  
10. }
```

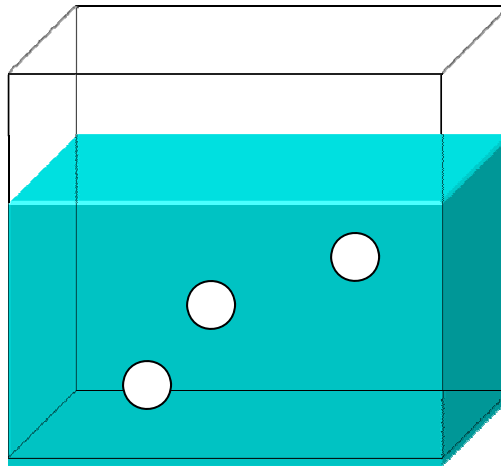
## 3-2: 簡単なソートアルゴリズム

# バブルソート

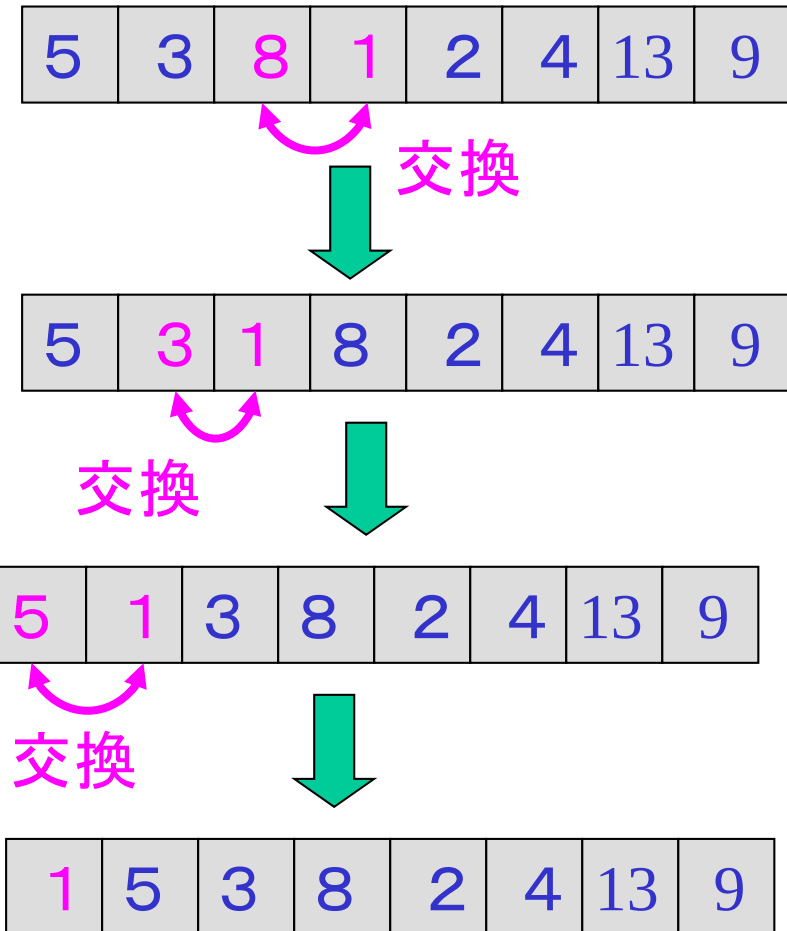
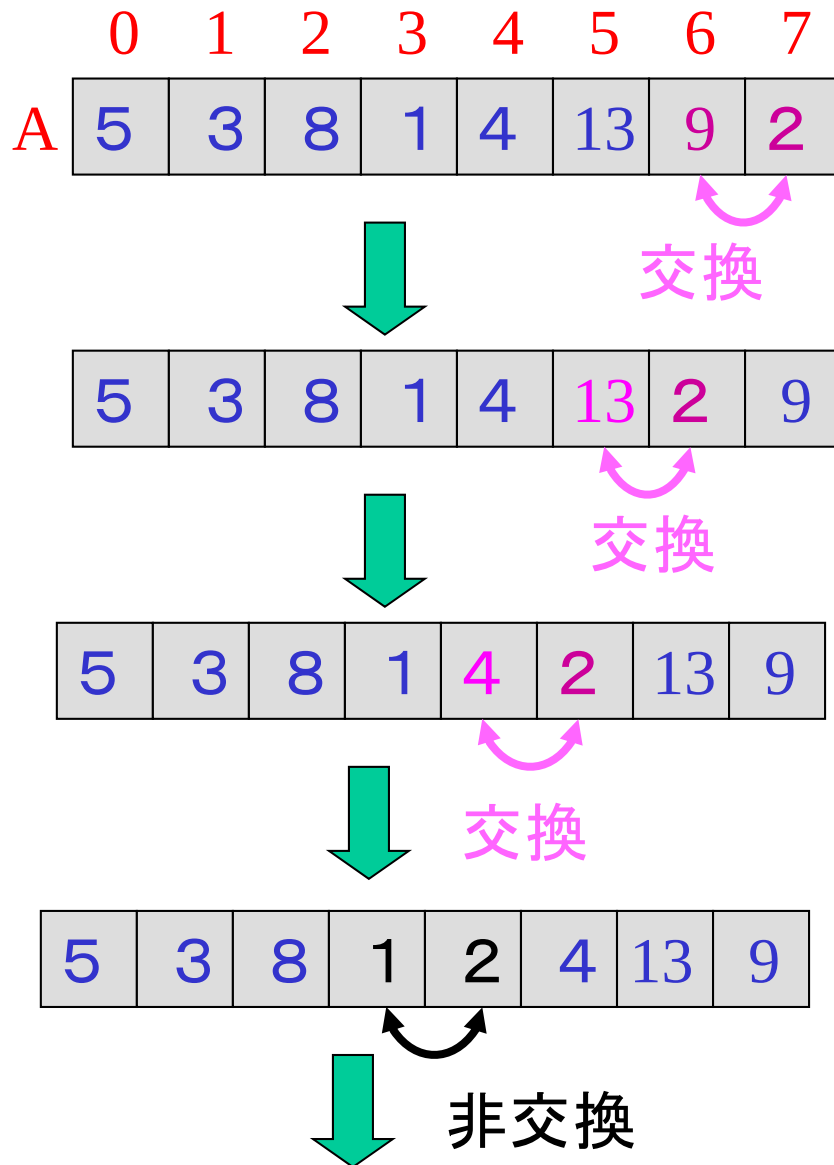
# バブルソートの方針

## 方針

- 隣同士比べて、小さいほうを上（添字の小さい方）に順にもっていく。
- 先頭の方は、ソートされている状態にしておく。
- これらを繰り返して、全体をソートする。

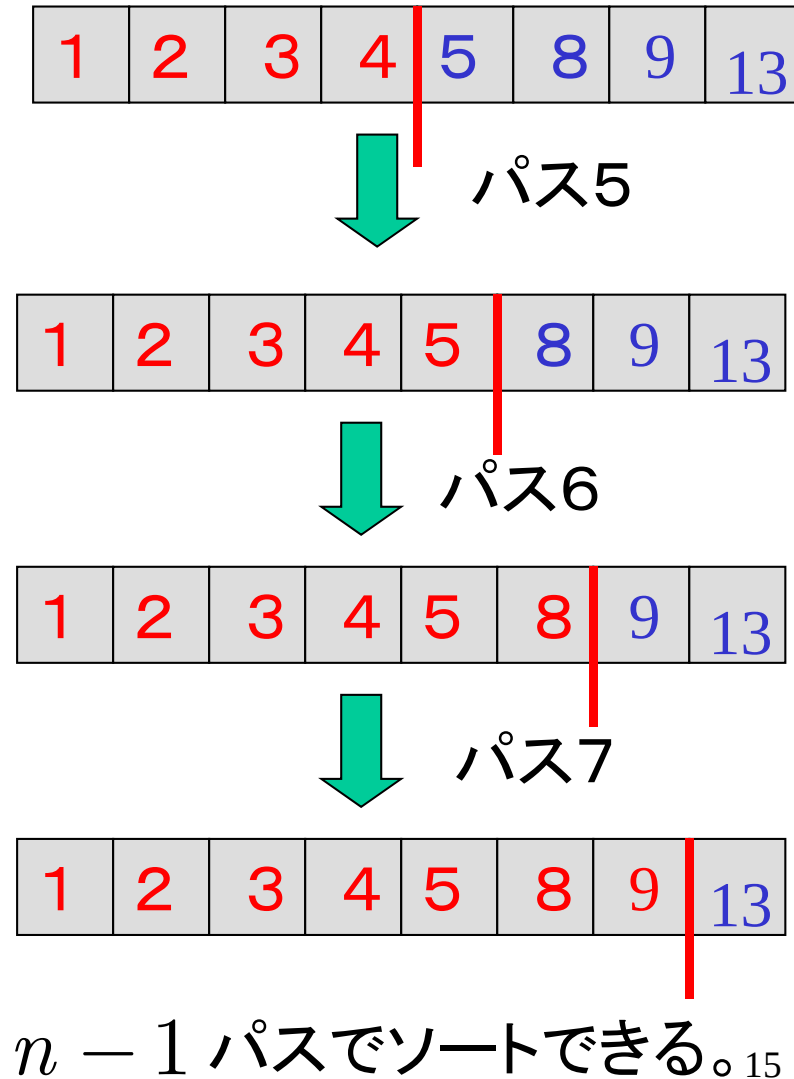
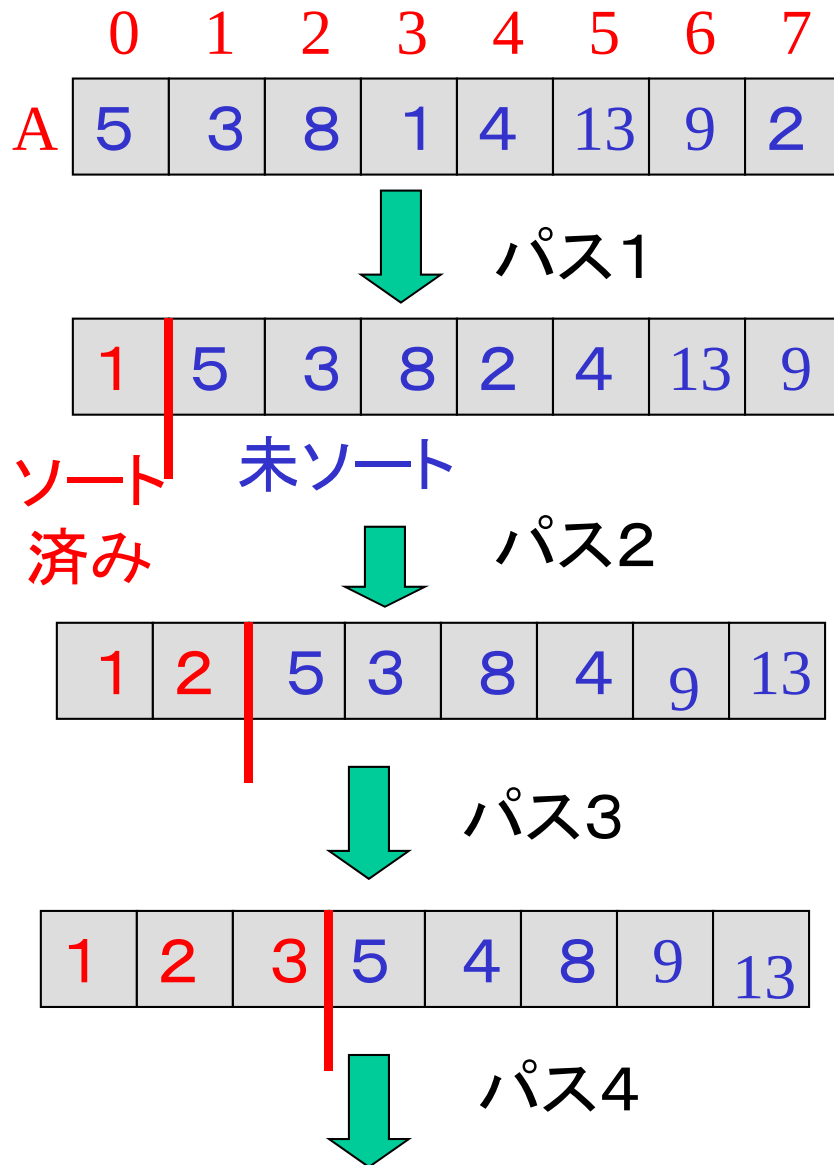


# バブルソートの動き1



この一連の動作をバブルソートの「パス」といいます。

# バブルソートの動き2



# 練習

次の配列を、バブルソートでソートするとき、  
全てのパスの結果を示せ。

|    |    |    |   |   |   |    |   |
|----|----|----|---|---|---|----|---|
| 11 | 25 | 21 | 1 | 8 | 3 | 16 | 5 |
|----|----|----|---|---|---|----|---|



# バブルソートの実現

```
/* バブルソート*/
1. void bubble()
2. {
3.     int i,j; /* カウンタ*/
4.     for(i=0;i<n-1;i++)
5.     {
6.         for(j=n-1;j>i;j--)
7.         {
8.             if(A[j-1]>A[j])
9.             {
10.                swap(&A[j-1],&A[j]);
11.            }
12.        }
13.    }
14.    return;
15.}
```

$j > 0$ としてもいい  
が時間計算量が  
約2倍になる

### 命題B1 (bubbleの正当性1)

内側のforループ(ステップ6)がk回繰り返されたとき、  
A[n-k]からA[n-1]までの最小値が  
A[n-k]に設定される。

#### 証明

k-1回の繰り返しによって、  
A[n-k-1]にA[n-k-1]からA[n-1]までの最小値が  
保存されていることに注意する。

したがって、k回目の繰り返しにより、

$$\begin{aligned} & \min\{A[n-k], A[n-k-1]\} \\ &= \min\{A[n-k], \min\{A[n-k-1], \dots, A[n-1]\}\} \end{aligned}$$

がA[n-k]に設定される。

(より厳密な数学的帰納法で証明することもできるが、  
ここでは省略する。)

*QED* 18

## 命題B2(bubbleの正当性2)

4.のforループがk回繰り返されたとき、  
(すなわち、パスkまで実行されたとき、)  
前半のk個( $A[0]$ - $A[k-1]$ )  
は最小のk個がソートされている。

### 証明

各パスkにおいては、 $A[k-1]$ から $A[n-1]$ の最小値が、  
 $A[k-1]$ に設定される。(命題B1より)  
このことに注意すると、数学的帰納法により、  
証明できる。(厳密な証明は省略する。)

*QED* 19

# バブルソートの計算量

パス1で、 $n-1$ 回の比較と交換

パス2で、 $n-2$

▪

▪

▪

パス $n-1$ で、1回の比較と交換

よって、 $(n-1) + (n-2) + \cdots + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

時間計算量  $O(n^2)$  のアルゴリズム

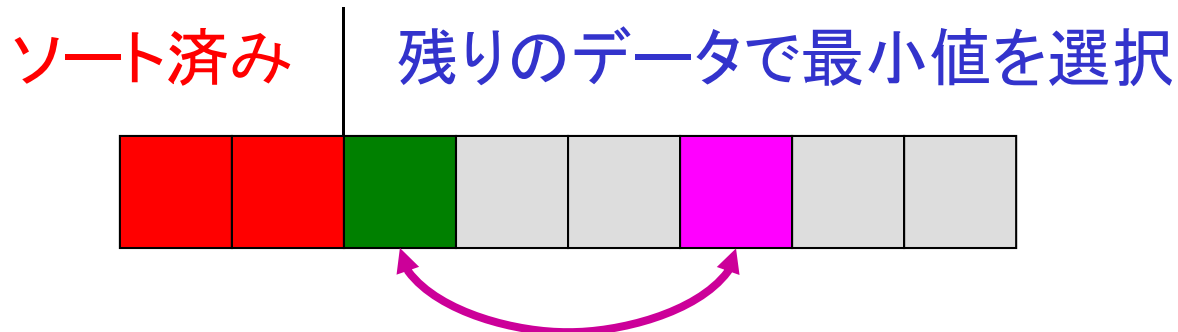
領域計算量は  $O(n)$ 。

# 選択ソート

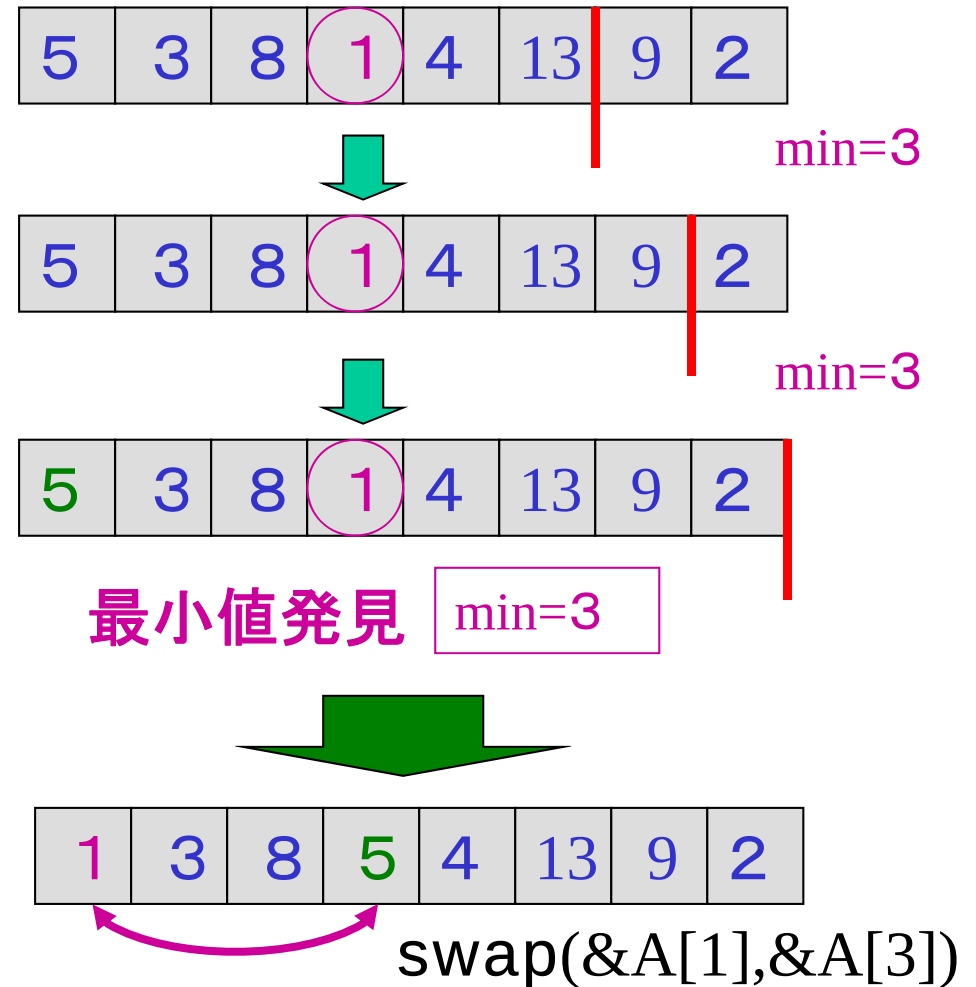
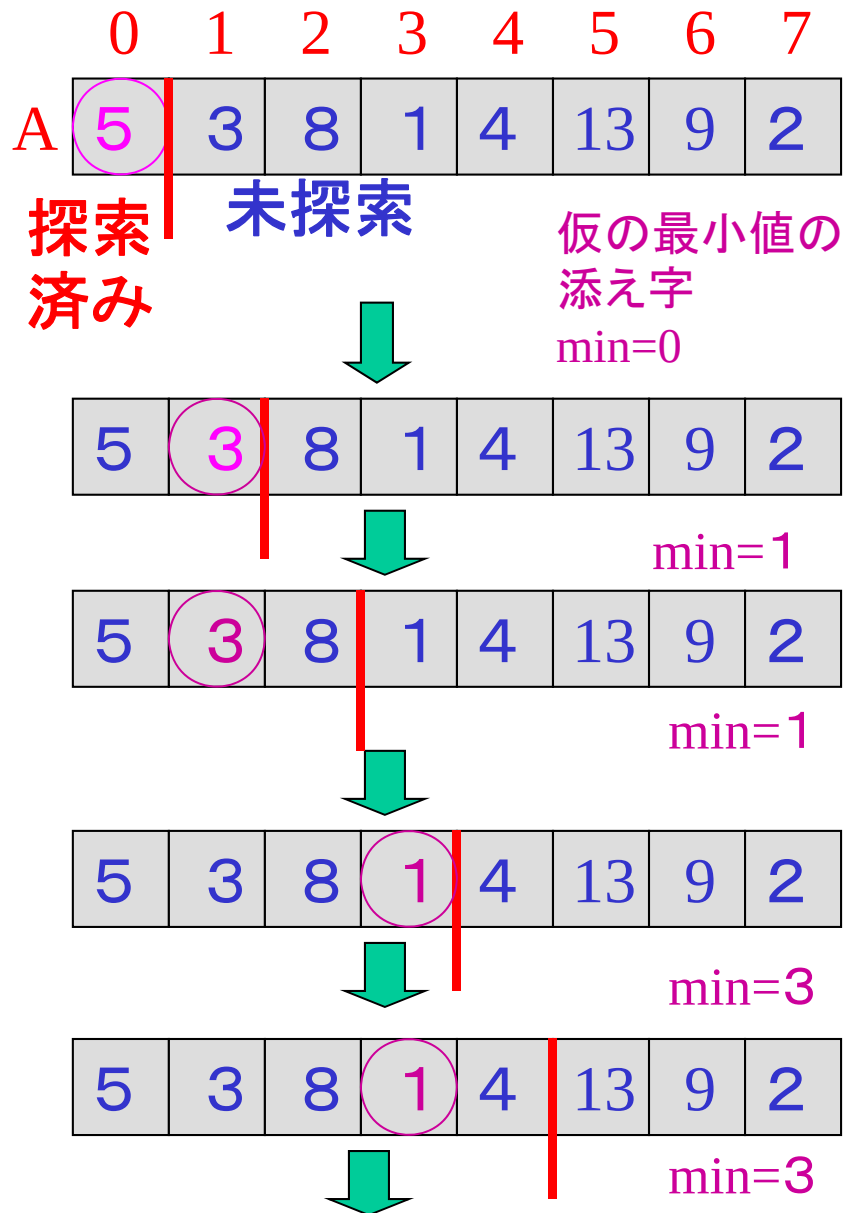
# 選択ソートの方針

## 方針

- 先頭から順に、その位置に入るデータを決める。  
(最小値を求める方法で選択する。)
- その位置のデータと選択されたデータを交換する。
- これらを繰り返して、全体をソートする。

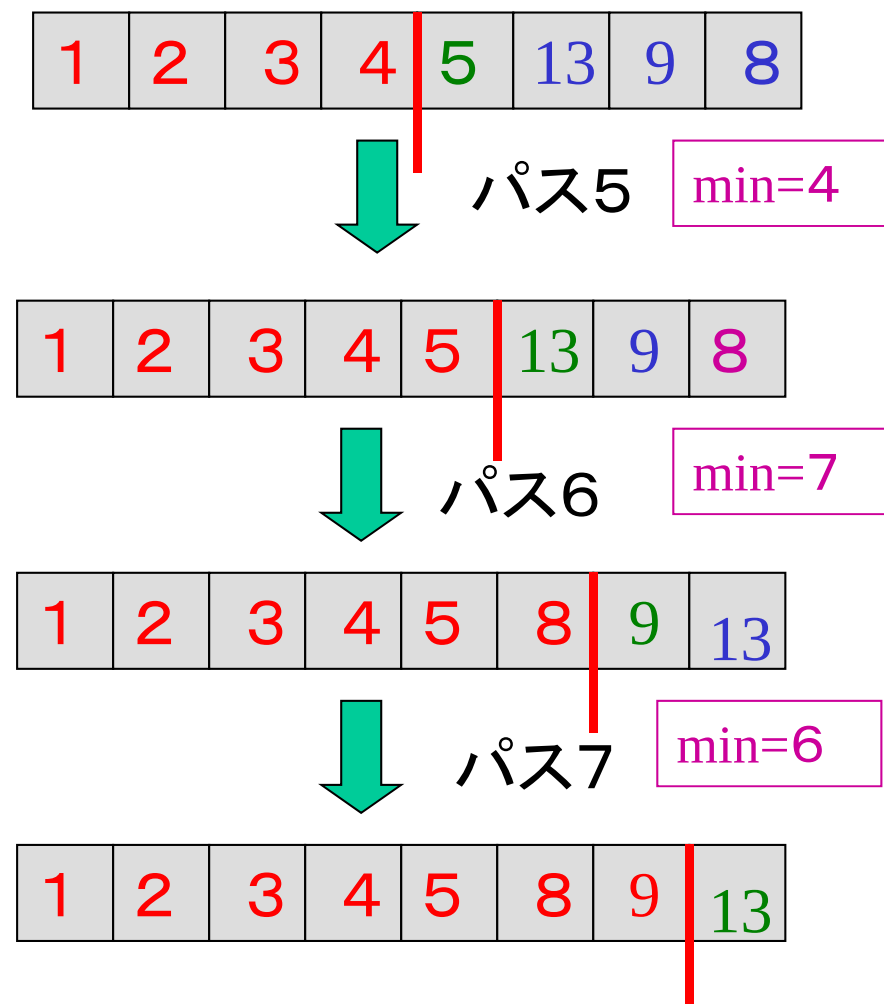
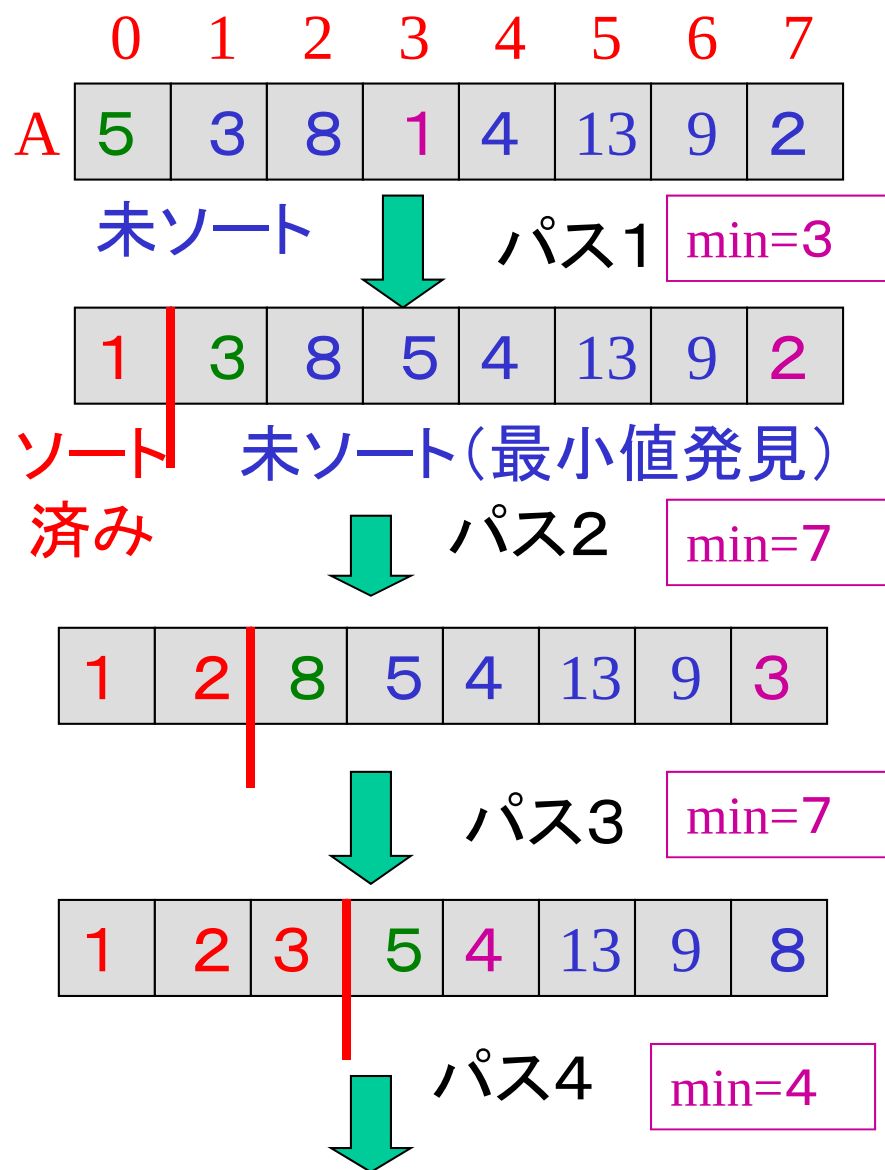


# 選択ソートの動き1 (最小値発見)



この一連の動作を選択ソートの「パス」といいます。

# 選択ソートの動き2



$n - 1$  パスでソートできる。24



# 練習

次の配列を、選択ソートでソートするとき、  
全てのパスの結果を示せ。

|    |    |    |   |   |   |    |   |
|----|----|----|---|---|---|----|---|
| 11 | 25 | 21 | 1 | 8 | 3 | 16 | 5 |
|----|----|----|---|---|---|----|---|

# 選択ソートの実現1

## (最小値を求めるアルゴリズム)

```
/* 選択用の関数、A[left]からA[right]
   までの最小値を求める */
1. int find_min(int left,int right)
2. {
3.     int min=left; /* 仮の最小値の添字 */
4.     int j=left; /* カウンタ */
5.
6.     min=left;
7.     for(j=left+1;j<=right;j++)
8.     {
9.         if(a[min]>a[j]){ min=j; }
10.    }
11.    return min;
12.}
```

# 選択ソートの実現2

```
/* 選択ソート*/  
1. void slection_sort()  
2. {  
3.     int i; /* カウンタ*/  
4.     int min; /* 最小値の添字*/  
5.     for(i=0;i<n-1;i++)  
6.     {  
7.         min=find_min(i,n-1);  
8.         swap(&A[i],&A[min]);  
9.     }  
10.    return;  
11.}
```

なお、説明の都合上、関数find\_minを作ったが、関数呼び出しで余分に時間がとられるので、実際は2重ループにするほうが速いと思われる。(でも、オーダーでは、同じ。)

### 命題S1(選択ソートの正当性1)

`find_min(left, right)`は、 $A[\text{left}] - A[\text{right}]$ 間の最小値を添え字を求める。

証明

1回目の資料の命題1と同様に証明される。

*QED* 28

## 命題S2(選択ソートの正当性2)

5. のforループが $i+1$ 回繰り返されたとき、  
(パス $i$ まで実行されたとき、)  
 $A[0]-A[i]$ には、小さい方から $i+1$ 個の要素が  
ソートされてある。

### 証明

先の命題S1を繰り返して適用することにより、  
この命題S2が成り立つことがわかる。  
(厳密には数学的帰納法を用いる。)

*QED* 29

# 選択ソートの計算量

パス1 find\_minで、n-1回の比較

パス2 n-2

・  
・  
・

パスn-1のfind\_minで、1回の比較

よって、 $(n-1) + (n-2) + \dots + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$  回の比較

交換は、n-1回

時間計算量  $O(n^2)$  のアルゴリズム

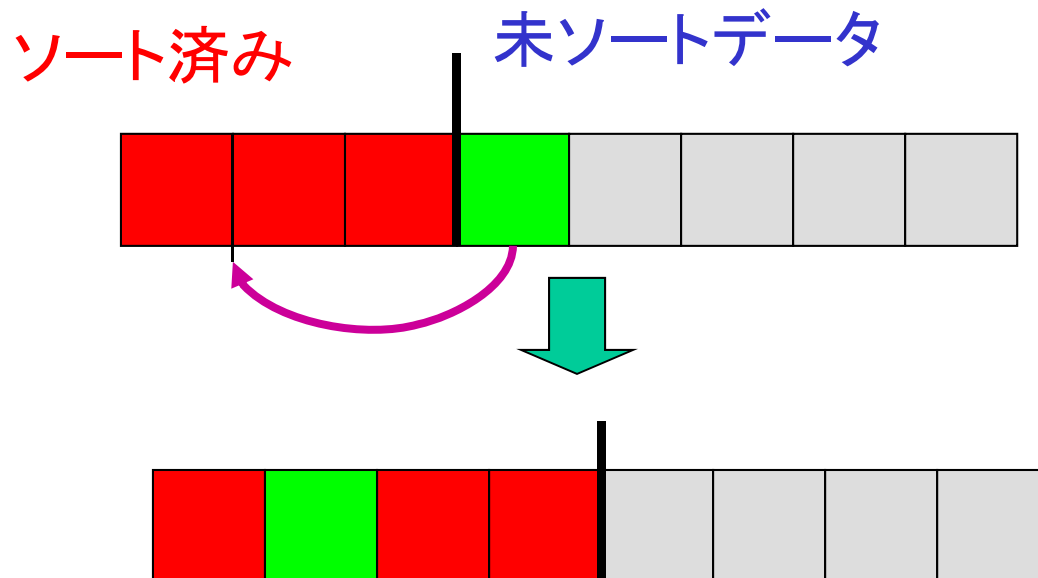
領域計算量は  $O(n)$ 。

# 挿入ソート

# 挿入ソートの方針

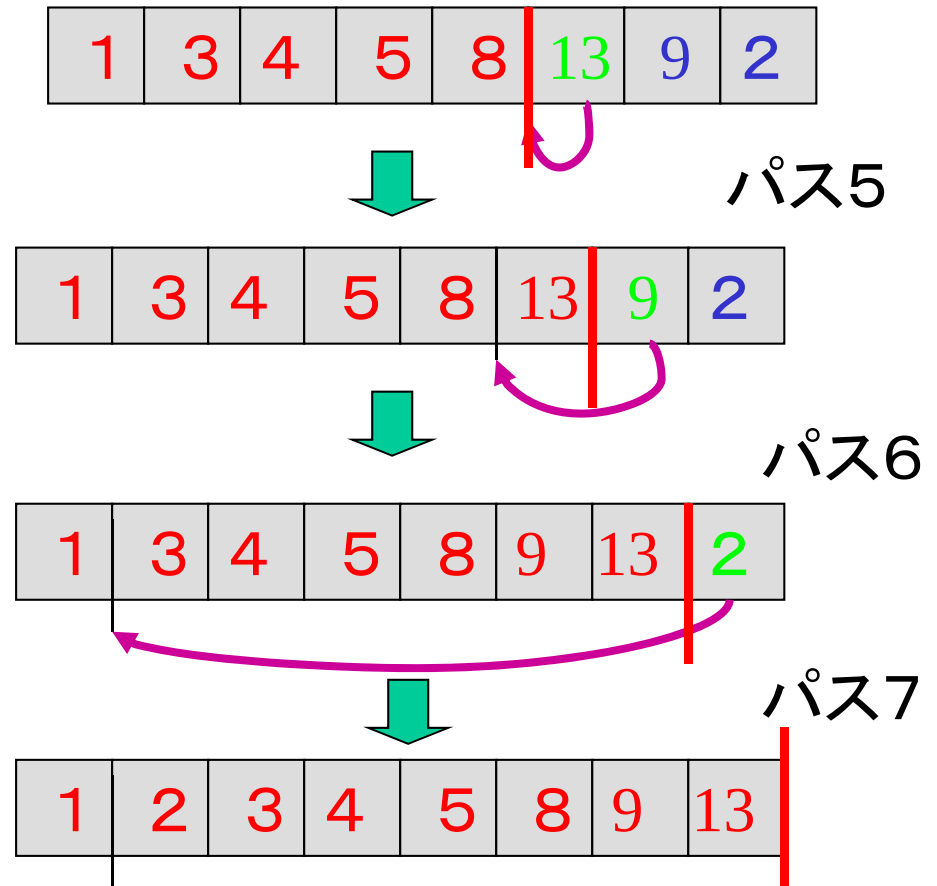
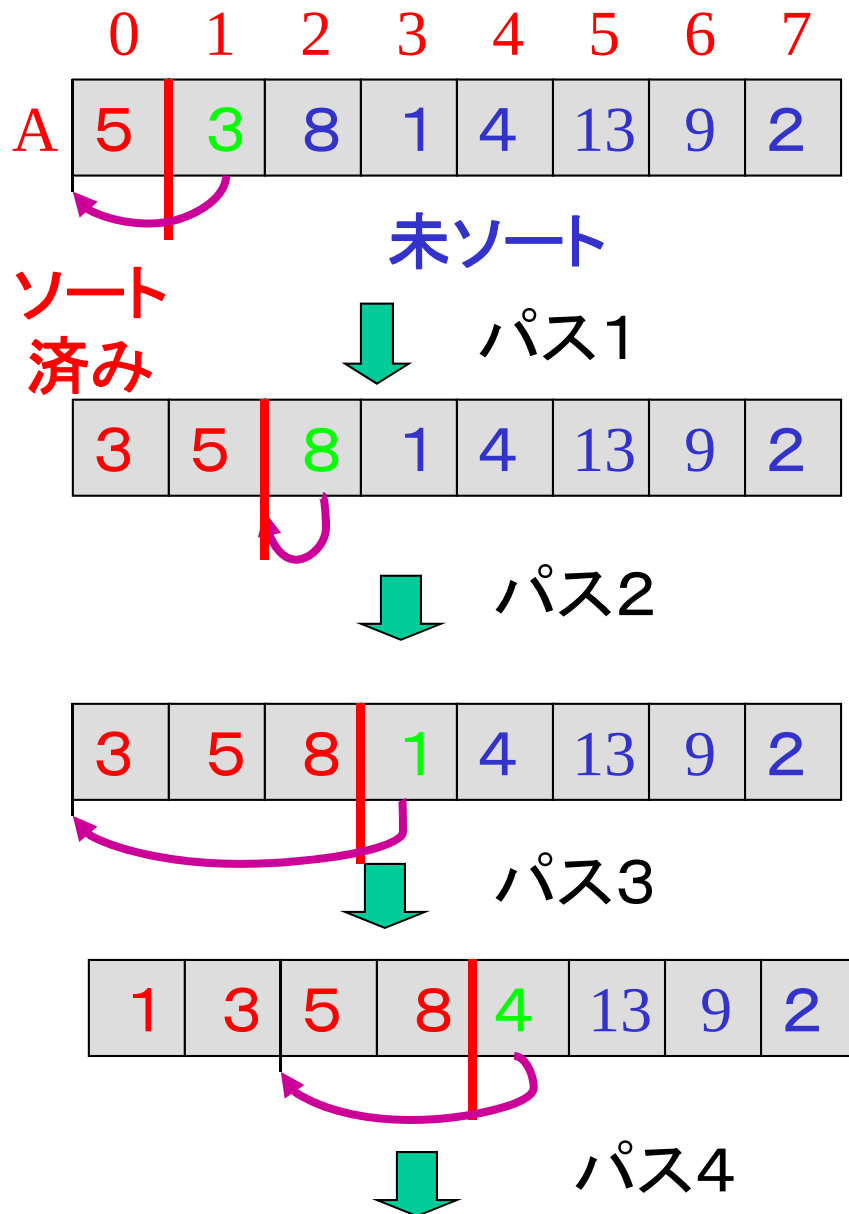
## 方針

- 先頭の方は、ソート済みの状態にしておく。
- 未ソートのデータを、ソート済みの列に挿入し、ソート済みの列を1つ長くする。
- これらを繰り返して、全体をソートする。



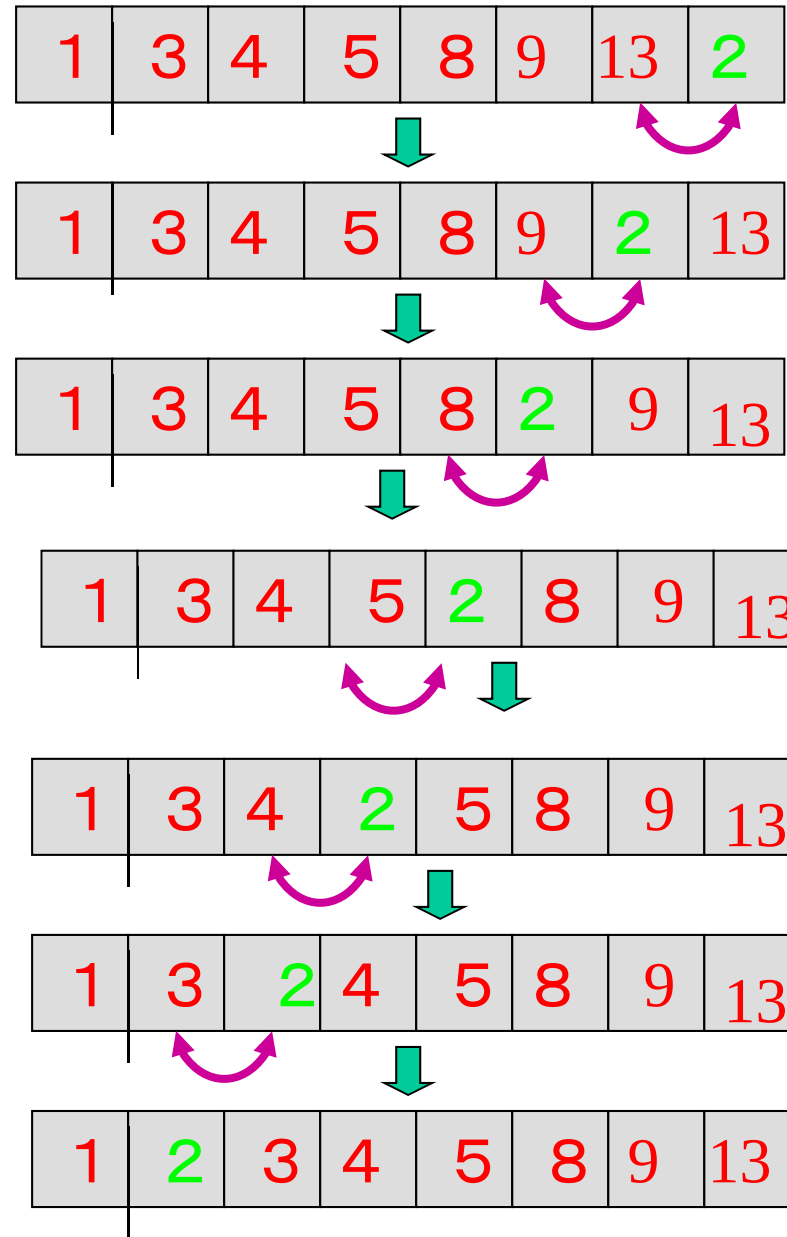
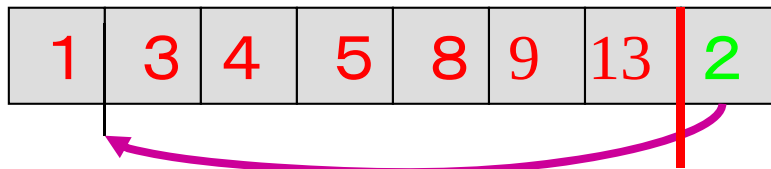


# 挿入ソートの動き1



この各回の挿入操作を、  
挿入ソートの「パス」といいます。  
n-1パスで挿入ソートが実現できる。

## 挿入ソートの動き2 (挿入動作詳細)



# 練習

次の配列を、挿入ソートでソートするとき、  
全てのパスの結果を示せ。

|    |    |    |   |   |   |    |   |
|----|----|----|---|---|---|----|---|
| 11 | 25 | 21 | 1 | 8 | 3 | 16 | 5 |
|----|----|----|---|---|---|----|---|

# 挿入ソートの実現1 (挿入位置を求める)

```
/* 挿入位置を見つける関数、  
A[left]からA[right-1]までソート済みのとき、  
A[right]の順番を求める。*/  
1. int find_pos(int left,int right)  
2. {  
3.     int j=left;      /* カウンタ */  
4.  
5.     for(j=left;j<=right;j++)  
6.     {  
7.         if(A[j]>A[right]){break;}  
8.     }  
9.     return j;  
10.}
```

# 挿入ソートの実現2(挿入)

```
/* 挿入(A[right]をA[pos]に挿入する。)*/  
1. void insert(int pos,int right)  
2. {  
3.     int k=right-1; /* カウンタ*/  
4.     for(k=right-1;k>=pos;k--)  
5.     {  
6.         pos=find_pos(i,A);  
7.         for(j=n-1;j<pos;j--)  
8.         {  
9.             swap(&A[k],&A[k+1]);  
10.        }  
11.    }  
12.    return;  
13.}
```

# 挿入ソートの実現3 (繰り返し挿入)

```
/* 挿入ソート */  
1. void insertion_sort()  
2. {  
3.     int i=0; /* カウンタ(パス回数) */  
4.     int pos=0; /* 挿入位置 */  
5.     for(i=1; i<n; i++)  
6.     {  
7.         pos=find_pos(0,i);  
8.         insert(pos,i);  
9.     }  
10.    return;  
11. }
```

## 命題I1(挿入ソートの正当性)

5. のforループが*i*回繰り返されたとき、  
(パス*i*まで実行されたとき、)  
 $A[0]$ から $A[i]$ はソートされてある。

### 証明

挿入find\_posによって、挿入位置を適切に見つけている  
また、insertによって、すでにソート済みの列を崩すことなく  
ソート済みの列を1つ長くしている。  
したがって、*i*回の繰り返しでは、*i*+1個のソート列が構成される。  
これらのソート列は、 $A[0]$ - $A[i]$ に保持されるので、命題  
は成り立つ。

*QED* 39

## 命題I2 (挿入ソートの停止性)

**insertion\_sortは停止する。**

### 証明

各繰り返しにおいて、ソート列が一つずつ長くなる。  
入力データは $n$ 個であるので、 $n-1$ 回の繰り返しにより、  
必ず停止する。

*QED* 40



# 挿入ソートの計算量

パス1で、1回の比較あるいは交換

パス2で、2回の

・

・

・

パス $n-1$ で、 $n-1$ の比較あるいは交換

よって、比較と交換回数の合計は、

$$1 + 2 + \dots + (n-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

時間計算量  $O(n^2)$  のアルゴリズム

領域計算量は  $O(n)$ 。

(挿入ソートを元に高速化した、シェルソートっていうものもあるが省略。)

# 簡単なソートのまとめ (最悪時間計算量)

| 方法     | 比較                          | 交換                          | 合計                              |
|--------|-----------------------------|-----------------------------|---------------------------------|
| バブルソート | $\frac{n(n-1)}{2} = O(n^2)$ | $\frac{n(n-1)}{2} = O(n^2)$ | $n(n-1) = O(n^2)$               |
| 選択ソート  | $\frac{n(n-1)}{2} = O(n^2)$ | $n-1 = O(n)$                | $\frac{(n-1)(n+2)}{2} = O(n^2)$ |
| 挿入ソート  | $\frac{n(n-1)}{2} = O(n^2)$ | $\frac{n(n-1)}{2} = O(n^2)$ | $\frac{n(n-1)}{2} = O(n^2)$     |

## 3-3: 高度なソートアルゴリズム①

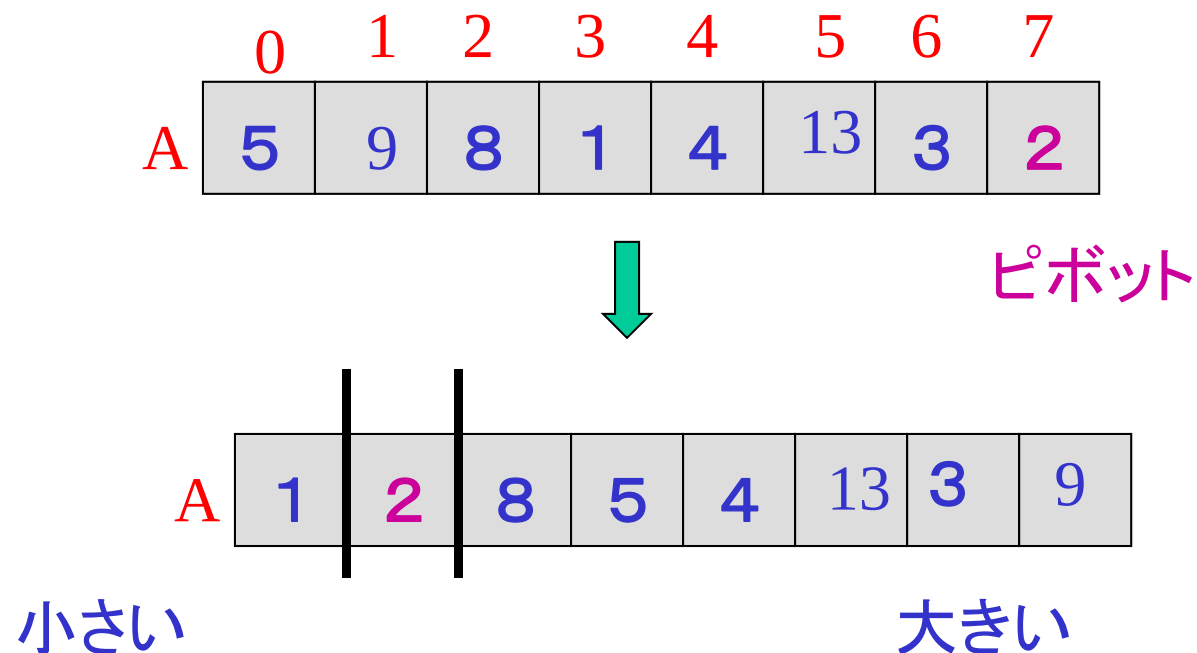
### (分割統治法にもとづくソート)

# クイックソート

# クイックソートの方針

## 方針

- 問題を小分けにして、あとで組み合わせる。(分割統治法)
- 前半部分に特定要素(ピボット)より小さい要素を集め、後半部分にピボットより大きく要素を集める。
- ピボットの位置を確定し、小分けした問題は、再帰的にソートする。

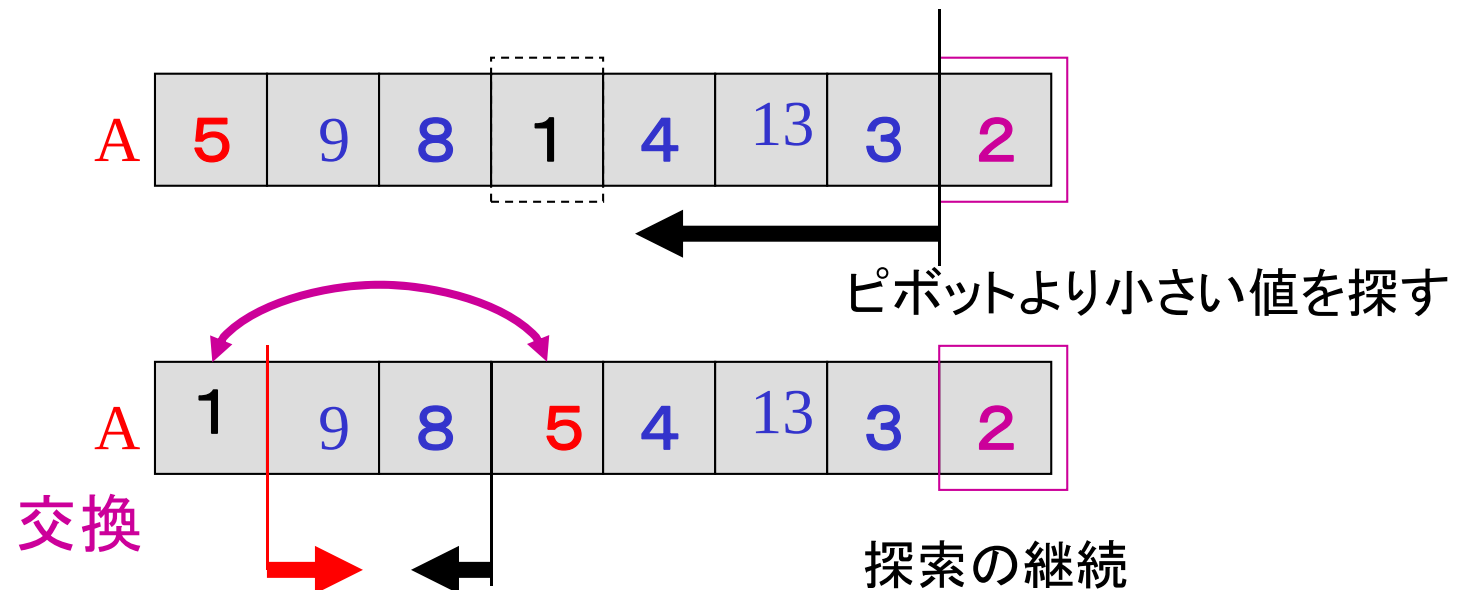
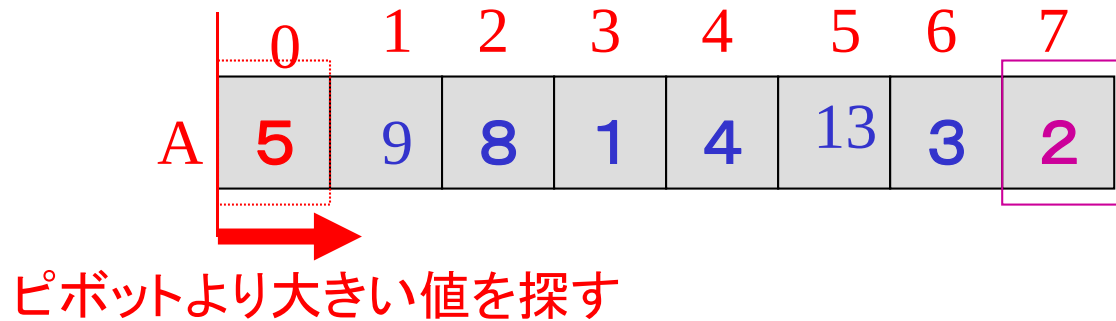


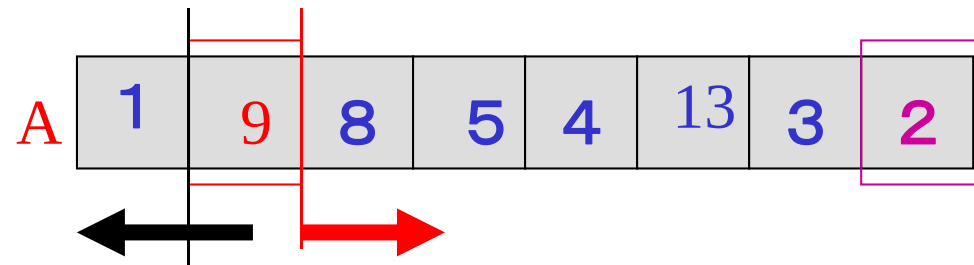
# 説明上の注意

- 全てのデータが異なるとして、説明します。
- クイックソートのアルゴリズムでは、ピボットの選び方にあいまいさがあります。  
(自由度といったほうがいいのかも。)  
ここでは、ソート範囲の最後の要素をピボットとして説明します。

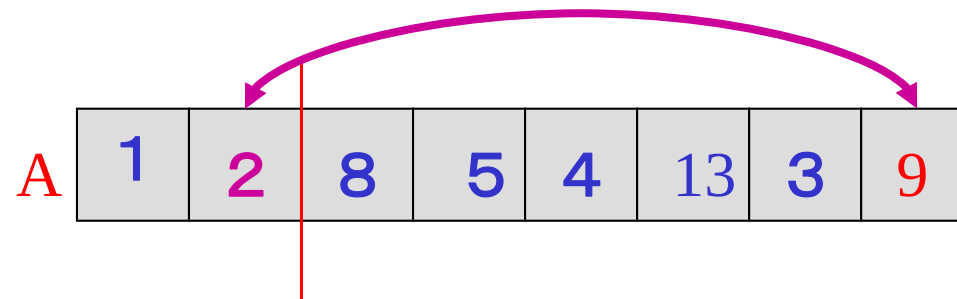
実際に、  
プログラミングするときは、  
もっといろんな状況を考えましょう。

# クイックソートの動き前半(分割1)

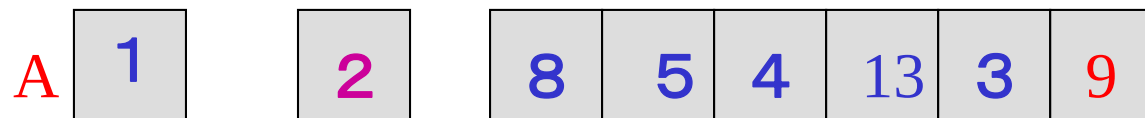




探索が交差したら分割終了。



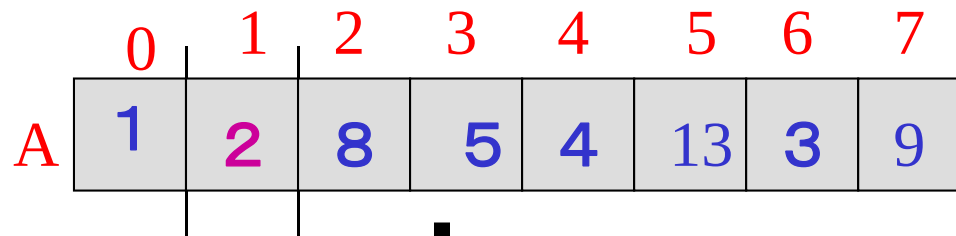
ピボットと前半最後の要素を交換し、  
あとは再帰呼び出し。



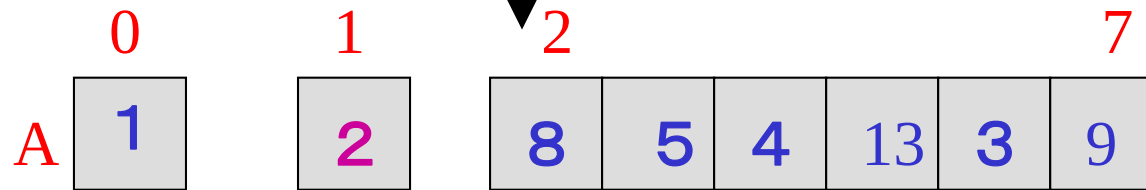
ピボットは位置確定



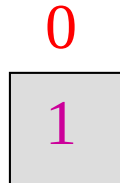
# クイックソートの動き後半(再帰)



partition(0,7)



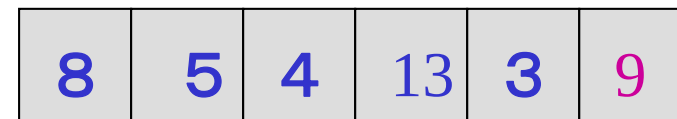
q\_sort(0,0)  
A[0]から  
A[0]までを  
ソートして



位置確定

q\_sort(2,7)

A[2]からA[7]までを  
ソートして



partition(2,7)



q\_sort(2,5)

位置確定  
以下省略

q\_sort(7,7) <sup>49</sup>

位置確定

# 練習

次の配列を、クイックソートでソートするとき、  
前のスライドに対応する図を作成せよ。

|    |    |    |   |   |   |    |   |
|----|----|----|---|---|---|----|---|
| 11 | 25 | 21 | 1 | 8 | 3 | 16 | 5 |
|----|----|----|---|---|---|----|---|

# クイックソートの実現1 (分割)

```
/* 概略です。細かい部分は省略 */  
1. int partition(int left,int right)  
2. {  
3.     int i,j;          /*カウンタ*/  
4.     i=left;  
5.     j=right-1;  
6.     while(TRUE) {  
7.         while(A[i]<pivot) {i++;}  
8.         while(A[j]>pivot) {j--;}  
9.         if(i>=j) {break;}  
10.        swap(&A[i],&A[j]);  
11.    }  
12.    swap(&A[i],&A[right]);  
13.    return(i);  
14.}
```

# クイックソートの実現2(再帰)

```
/*概略です。細かい部分は省略*/
1. void q_sort(int left,int right)
2. {
3.     int pos;    /*分割位置 */
4.     if(left >= right)
5.     {
6.         return;
7.     }
8.     else
9.     {
10.        pos=partition(left,right);
11.        q_sort(left,pos-1);
12.        q_sort(pos+1,right);
13.        return;
14.    }
15.}
```

### 命題Q1 (クイックソートの停止性)

$q\_sort(left, right)$  は必ず停止する。

#### 証明

$left \leq pos \leq right$  が常に成り立つことに注意する。

$k \equiv right - left$  に関する帰納法で証明する。

基礎:  $k \leq 0$  のとき。

このときは、明らかにステップ6により終了する。

帰納:  $k \geq 1$  のとき。

$0 \leq k' < k$  なる全ての整数に対して、 $q\_sort(left, left+k')$  が終了すると仮定する。(帰納法の仮定。)

q\_sort(left, left+k)の停止性を考える。

このとき、else節(10-13)が実行される。

ステップ10で得られる pos の値に対して、

$$left \leq pos \leq left + k$$

が成り立つ。

ステップ11で呼び出す q\_sort(left, pos-1)において、  
その適用される列の長さは

$$pos - 1 - left \leq left + k - 1 - left = k - 1 < k$$

である。

したがって、帰納法の仮定より、  
q\_sort(left, pos-1)は停止する。

ステップ12で呼び出す  $q\_sort(pos+1, left+k)$  において、  
その適用される列の長さは

$$left + k - (pos + 1) \leq left + k - left - 1 = k - 1 < k$$

である。

したがって、帰納法の仮定より、  
 $q\_sort(pos+1, left+k)$  は停止する。

以上より、ステップ10-13の全ての行において、  
かく再帰式は停止する。

したがって、アルゴリズム  $q\_sort(left, right)$  は停止する。

# 停止しないクイックソート

例えば、次のようなクイックソート(?)は、  
停止するとは限らない。

```
1.    if(left >= right)
2.        {
3.            return;
4.        }
5.    else
6.        {
7.            pos = partition(left, right);
8.            q_sort(left, pos);
9.            q_sort(pos, right);
10.           return;
11.        }
12. }
```

サイズが小さくなる  
とは限らない。



## 命題Q2(クイックソートの正当性1)

ピボットに選択された値は、partition実行により、ソート済みの順列と同じ位置に設定される。

証明 ソート済みの順列を  $L_S$  とし、  
アルゴリズムの途中の順列を  $L$  とする。  
また、ピボット  $p$  の各順列における順位をそれぞれ、  
 $L_S(p)$ 、 $L(p)$  と表すものとする。

このとき、 $L_S$  において、 $p$  未満の要素数は  $L_S(p) - 1$  であり、  
 $p$  より大きい要素数は  $n - L_S(p) - 1$  である。  
一方、 $L$  における  $p$  未満の要素数は  $L(p) - 1$  であるが、  
これは  $L_S(p) - 1$  と同じはずである。  
したがって、

$$L_S(p) = L(p)$$

*QED* 57

### 命題Q3(クイックソートの正当性2)

全ての要素はピボットに選択されるか、あるいは  
列の長さ1の再帰呼び出しにより位置が決定される。

#### 証明

再帰呼び出しにおいて、サイズが減少することに  
注意すると、ピボットとして選ばれるか、サイズが  
1の再帰呼び出しされる。

*QED*

# クイックソートの計算量

クイックソートは、  
最悪時の計算量と平均の計算量が異なります。  
これらは、  
ピボットの選び方にもよりますが、  
どんな選び方によっても最悪のデータ初期配置があります。

ここでは、最悪計算量と、平均計算量の両方を考えます。

# クイックソートの最悪計算量

まず、関数partition(i,j)の1回の時間量は、 $j-i+1$ に比例した時間量です。

再帰の同じ深さで、partition()の時間量を総計すると  $O(n)$  になります。

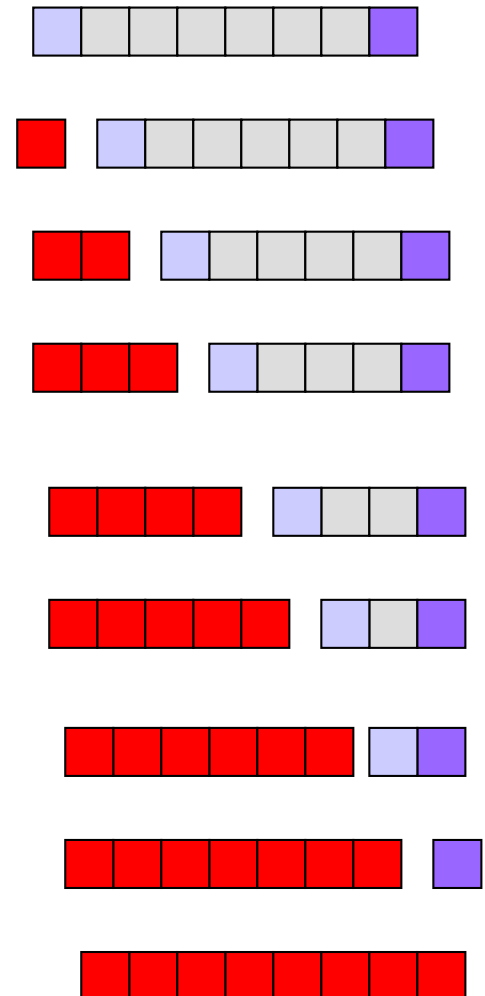
いつも0個、ピボット、残りのように分割されるのが最悪の場合です。つまり、ピボットとしていつも最小値が選択されたりするのが最悪です。

(他にも最悪のケースはあります。)

このときでも、partition(i,j)の実行には、 $j-i+1$ 回の演算が必要です。

これは、結局選択ソートの実行と同じようになり、

最悪時間計算量  $O(n^2)$  のアルゴリズム。



# クイックソートの平均時間計算量

- クイックソートの平均時間の解析は、複雑である。
- 順を追って解析する。

# 漸化式の導出

初期状態として、 $n!$  通りの並びがすべて等確率だとしましょう。

クイックソートの時間量を  $T(n)$  とします。

ピボットが  $i$  番目のときには、以下の漸化式を満たす。

$$T(n) \leq T(i-1) + T(n-i) + c_1(n-1)$$

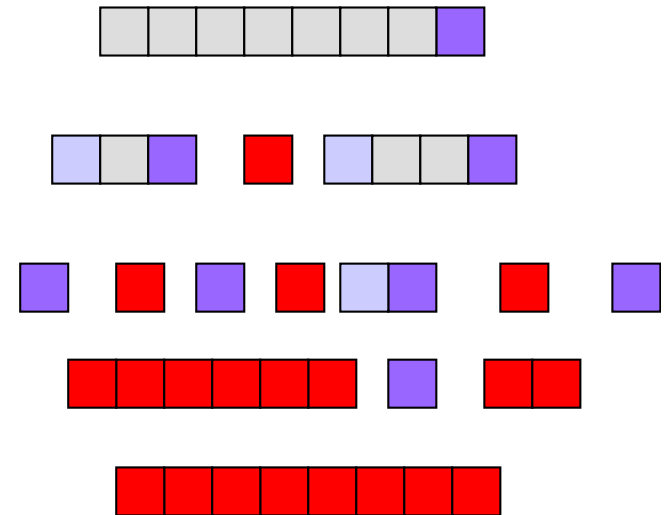
小さい方の分割を  
再帰的にソートする分

大きい方の分割を  
再帰的にソートする分

partition()分

ピボットの順位は、 $n$ 通り全て均等におこるので、それらを総和して、 $n$ で割ったものが平均時間量

$$T(n) \leq \frac{1}{n} \sum_{i=1}^n \{T(i-1) + T(n-i) + c_1(n-1)\}$$



したがって、入力順列がすべて均等に起こるという仮定では、クイックソートの平均時間計算量は、次の漸化式を満たす。

$$\begin{cases} T(0) = c_2 & n = 0 \\ T(n) \leq \frac{1}{n} \sum_{i=1}^n \{T(i-1) + T(n-i) + c_1(n-1)\} & n > 0 \end{cases}$$

# 漸化式の解法

漸化式における再帰式を個々に分解して調べる。

$$\begin{aligned} & \frac{1}{n} \sum_{i=1}^n \{T(i-1) + T(n-i) + c_1(n-1)\} \\ &= \frac{1}{n} \sum_{i=1}^n \{T(i-1)\} + \frac{1}{n} \sum_{i=1}^n \{T(n-i)\} + \frac{c_1}{n} \sum_{i=1}^n \{(n-1)\} \end{aligned}$$

まず、

$$\begin{aligned} \frac{c_1}{n} \sum_{i=1}^n \{(n-1)\} &= \frac{c_1}{n} \left( \underbrace{(n-1) + (n-1) + \cdots + (n-1)}_n \right) \\ &= \frac{c_1}{n} \{n(n-1)\} \\ &= c_1(n-1) \end{aligned}$$



次に、

$$\sum_{i=1}^n \{T(i-1)\} = T(0) + T(1) + \cdots + T(n-1)$$

$$\sum_{i=1}^n \{T(n-i)\} = T(n-1) + T(n-2) + \cdots + T(0)$$

$$\therefore \sum_{i=1}^n \{T(i-1)\} = \sum_{i=1}^n \{T(n-i)\}$$

したがって、

$$T(n) \leq \frac{2}{n} \sum_{i=0}^{n-1} \{T(i)\} + c_1(n-1)$$

$$\therefore nT(n) \leq 2 \sum_{i=0}^{n-1} \{T(i)\} + c_1 n(n-1)$$

$n$  に  $n-1$  を代入して、

$$(n-1)T(n-1) \leq 2 \sum_{i=0}^{n-2} \{T(i)\} + c_1(n-1)(n-2)$$

両辺の差をとる。

$$nT(n) - (n-1)T(n-1) \leq 2T(n-1) + c_1 n(n-1) - c_1 (n-1)(n-2)$$

$$\therefore nT(n) - (n+1)T(n-1) \leq c_1 2(n-1)$$

両辺を  $n(n+1)$  で割る。

$$\frac{T(n)}{n+1} - \frac{T(n-1)}{n} \leq c_1 2 \frac{(n-1)}{n(n+1)} \leq 2c_1 \frac{n+1}{n(n+1)} = 2c \frac{1}{n}$$

この式を辺々加える。

$$\frac{T(n)}{n+1} - \cancel{\frac{T(n-1)}{n}} \leq 2c_1 \frac{1}{n}$$

$$\cancel{\frac{T(n-1)}{n}} - \cancel{\frac{T(n-2)}{n-1}} \leq 2c_1 \frac{1}{n-1}$$

$\vdots$

$$\cancel{\frac{T(3)}{4}} - \cancel{\frac{T(2)}{3}} \leq 2c_1 \frac{1}{3}$$

$$+ ) \quad \cancel{\frac{T(2)}{3}} - \frac{T(1)}{2} \leq 2c_1 \frac{1}{2}$$


---

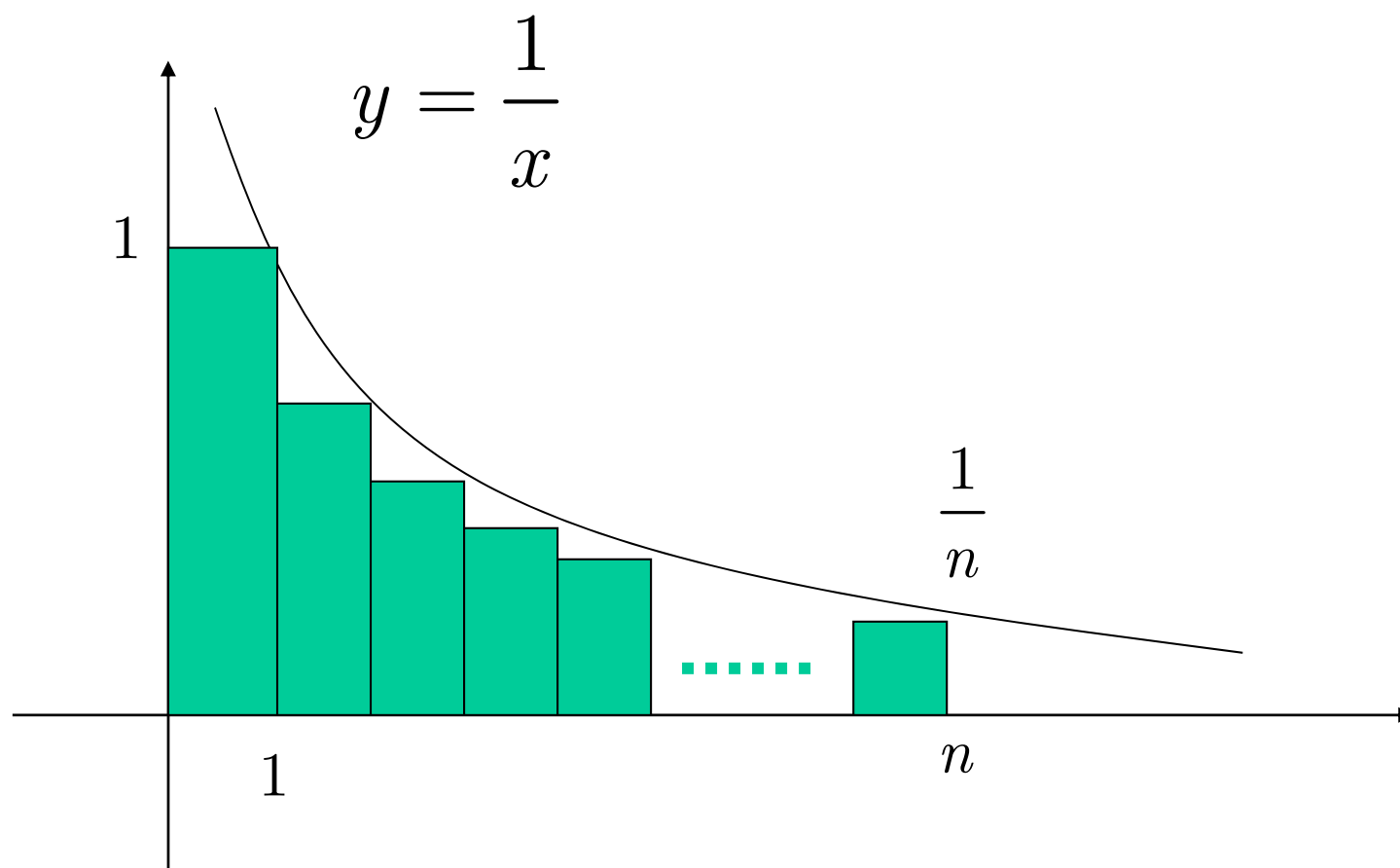
$$\therefore \frac{T(n)}{n+1} - \frac{c_2}{2} \leq 2c_1 \left( \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2} \right) = 2c_1 (H_n - 1)$$

ここで、

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

調和級数  
(Harmonic **Series**)

# 調和級数の見積もり



$$H_n - 1 < \int_{x=1}^n \frac{1}{x} dx = \log_e n$$

$$\therefore \frac{T(n)}{n+1} - \frac{c_2}{2} \leq 2c_1(H_n - 1) < 2c_1 \log_e n$$

$$\therefore T(n) \leq 2c_1(n+1) \log_e n + \frac{c_2}{2}(n+1)$$

$$\therefore T(n) = O(n \log n)$$

以上より、クイックソートの平均計算時間量は、

$$O(n \log n)$$

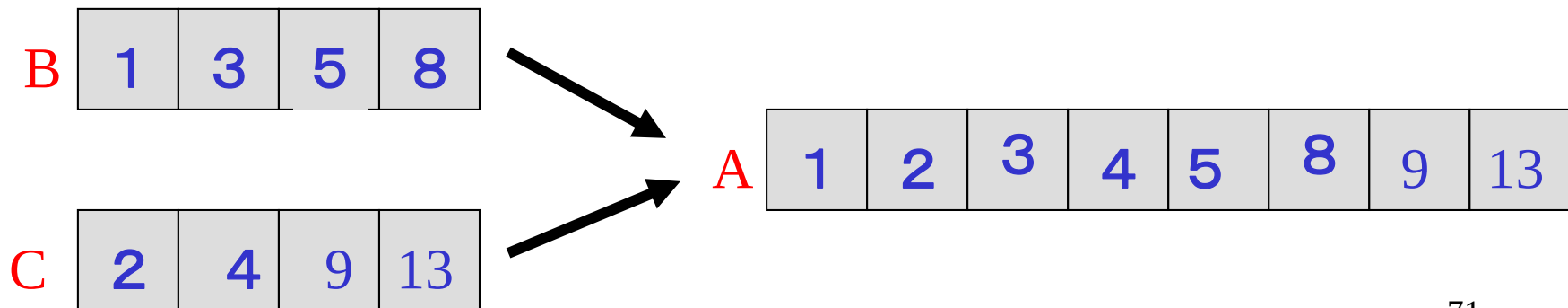
である。

# マージソート

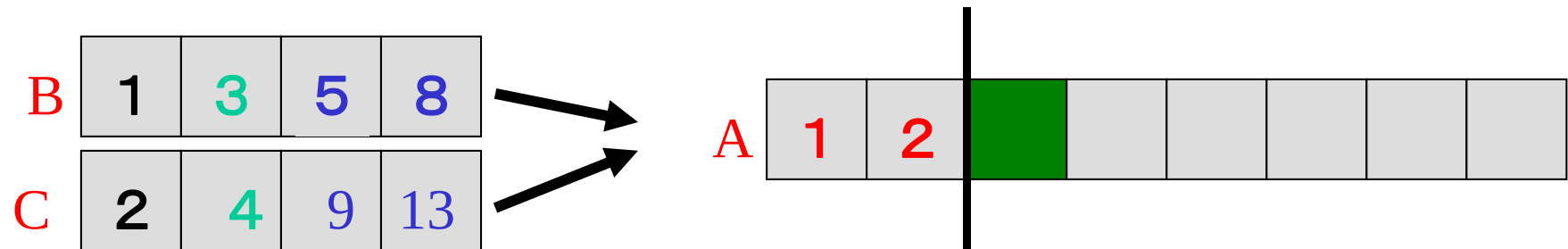
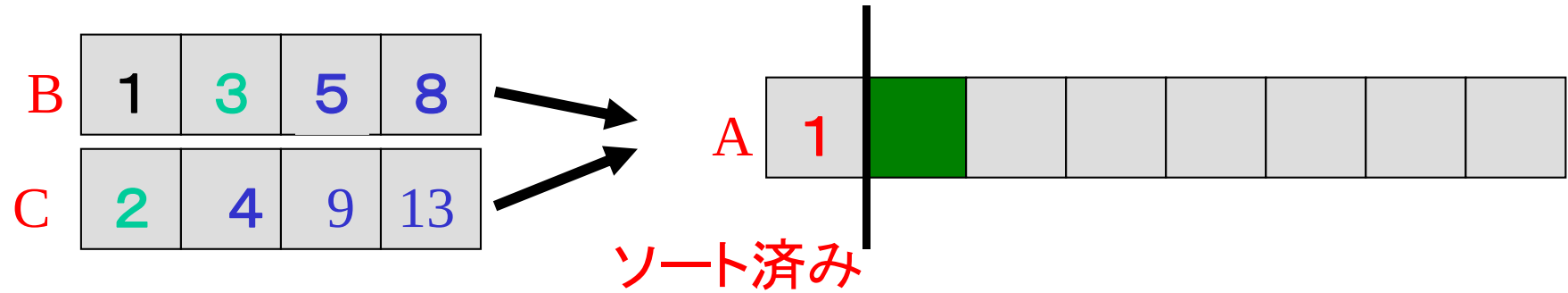
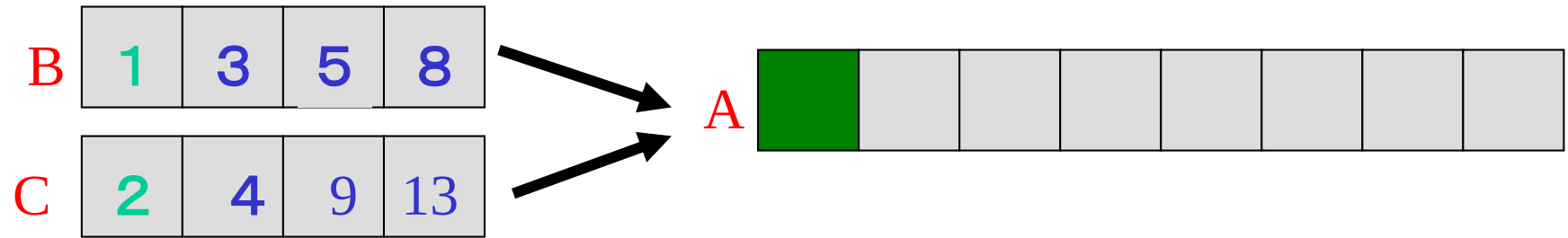
# マージソートの方針

## 方針

- 問題を小分けにして、あとで組み合わせる。(分割統治法)
- 小分けした問題は、再帰的にソートする。
- もしソートされた2つの配列があれば、それらのそれらを組み合わせて、大きいソートの列をつくる。(マージ操作)
- 1要素だけの列はソート済みとみなせる。



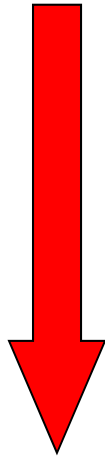
# マージの動き





# 分割

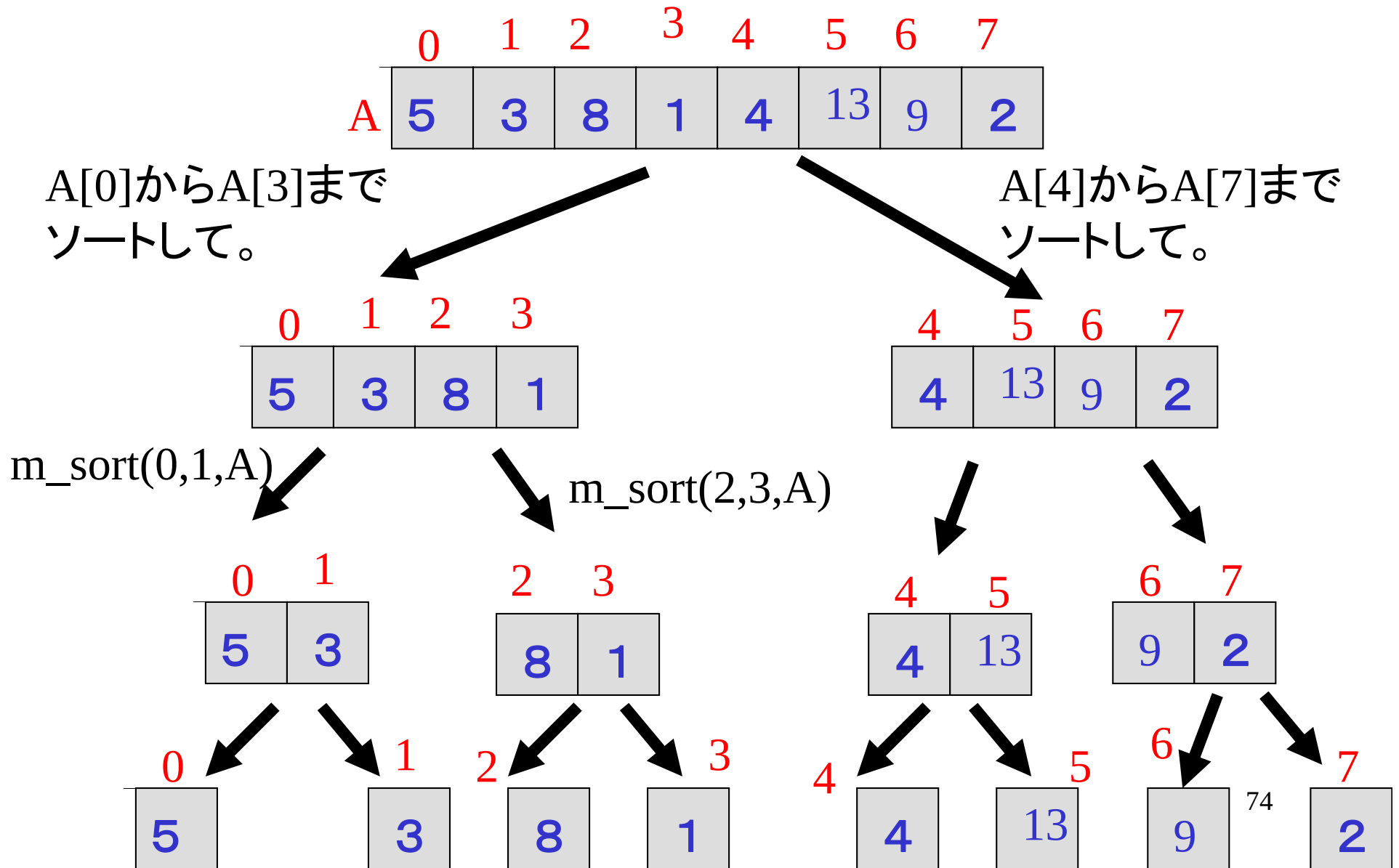
- もし2つのソート列があったら、マージ操作によって、長いソート列がえられることがわかった。
- どうやって、2つのソート列を作るのか？



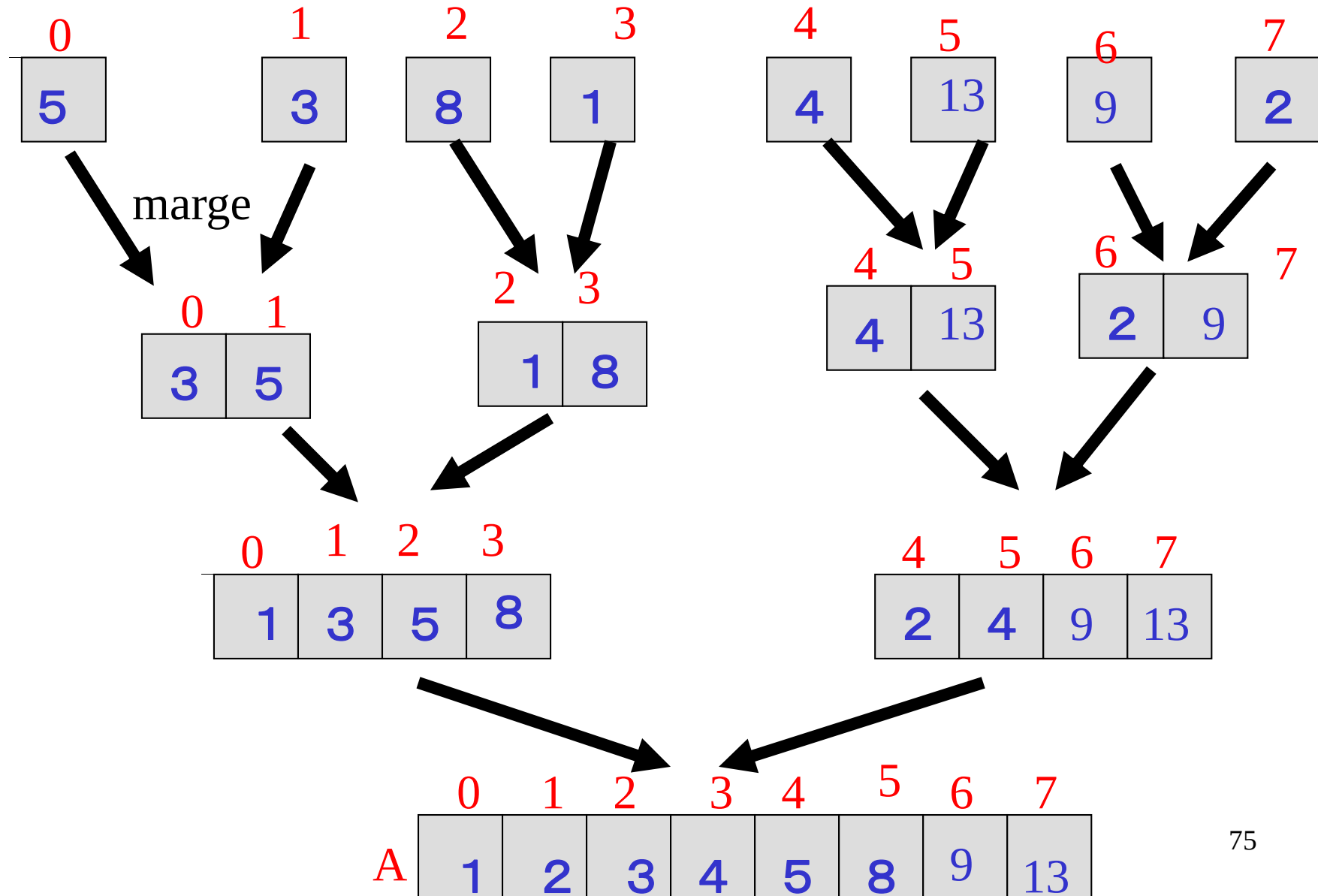
おなじ問題で、  
問題のサイズが小さくなっていることに  
注意する。

列を二等分にして、再帰的にソートする。

# マージソート動き前半(分割)



# マージソート動き後半(マージ)



# 練習

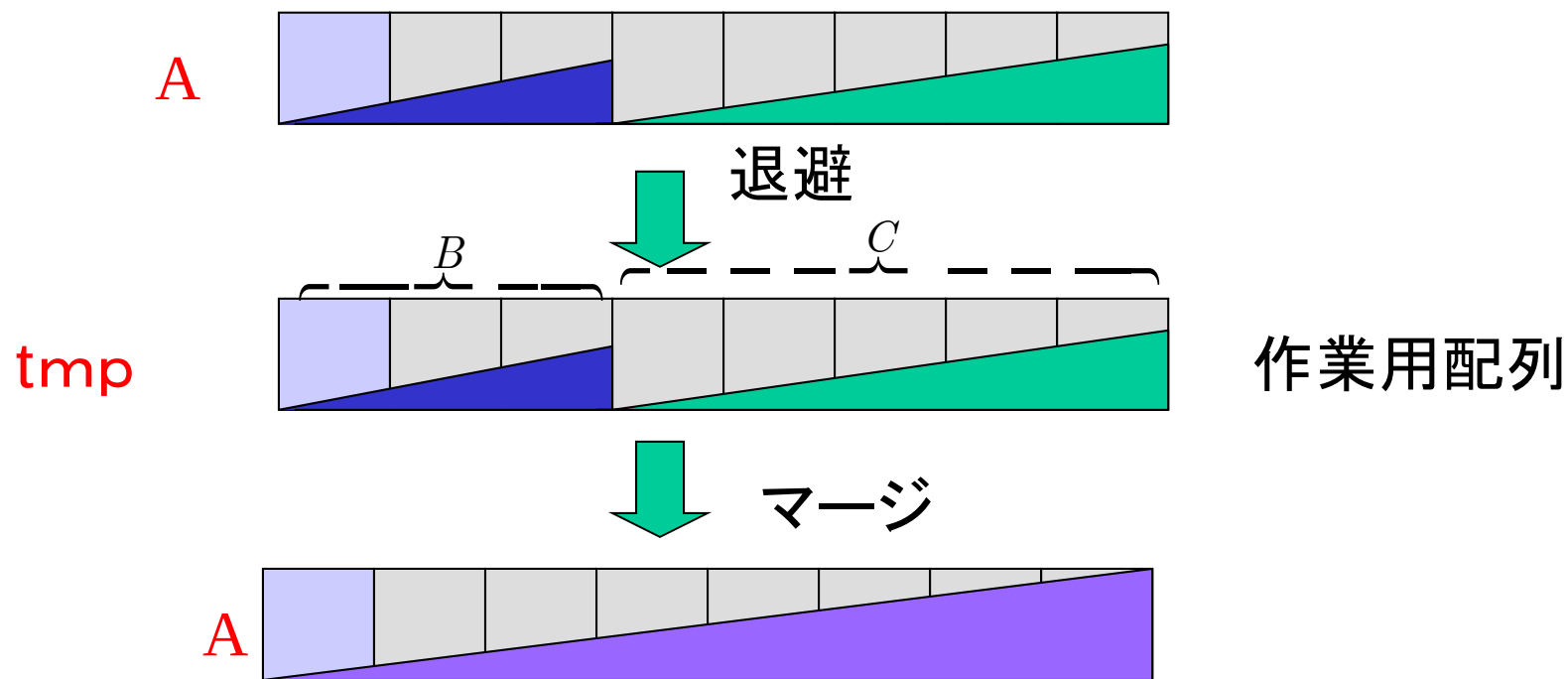
次の配列を、マージソートでソートするとき、  
前のスライドに対応する図を作成せよ。

|    |    |    |   |   |   |    |   |
|----|----|----|---|---|---|----|---|
| 11 | 25 | 21 | 1 | 8 | 3 | 16 | 5 |
|----|----|----|---|---|---|----|---|

# マージに関する注意

マージでは、配列の無いようにいったん別の作業用配列に蓄える必要がある。

## 作業用の配列が必要



## データ退避の実現

```
/* A[left]-A[right]をtmp[left]-tmp[right]に書き出す。*/  
void write(int left,int right)  
{  
    int i;  
    for(i=left;i<=right;i++){  
        tmp[i]=a[i];  
    }  
    return;  
}
```

## マージの実現

```
/* tmp[left]-tmp[mid]とtmp[mid+1]-tmp[right]を  
A[left]-A[right]にマージする。(細かい部分は省略) */  
void marge(int)  
{  
    int l=left,r=mid+1; /* tmp走査用 */  
    int i=left; /* A走査用 */  
    for(i=left;i<=right;i++){  
        if(tmp[l]<=tmp[r] && l<=mid){  
            A[i]=tmp[l];l++;  
        }else if(tmp[r]<tmp[l] && r<=right){  
            A[i]=tmp[r];r++;  
        }else if(l>mid){  
            A[i]=tmp[r];r++;  
        }else if(r>right){  
            A[i]=tmp[l];l++;  
        }  
    }  
    return;  
}
```

# マージソートの実現

```
/* 概略です。細かい部分は省略 */  
void merge_sort(int left, int right)  
{  
    int mid;    /* 中央 */  
    if (left >= right) {  
        return;  
    } else {  
        mid = (left + right) / 2;  
  
        merge_sort(left, mid);  
        merge_sort(mid + 1, right);  
  
        write(left, right);  
        merge(left, mid, right);  
        return;  
    }  
}
```



### 命題M1 (マージの正当性)

マージにより、2つの短いソート列から、  
一つの長いソート列が得られる。

### 証明

配列Aの走査用のカウンタに関する帰納法で  
証明することができる。(厳密な証明は省略)

*QED* 81

## 命題M2 (マージソートの正当性)

マージソートにより、配列が昇順にソートされる。

### 証明

再帰の深さに関する帰納法や、  
あるいはソートされている部分列の長さに関する帰納法  
で証明できる。(厳密な証明は省略。)

*QED* 82

### 命題M3(マージソートの停止性)

マージソートは停止する。

#### 証明

再帰呼び出しにおいて、必ずサイズが小さくなる(約半分)ことに注意する。

また、要素数が1以下の時には、停止することにも注意する。  
これらの考察から、帰納法で証明できる。  
(厳密な証明は省略。)

*QED* 83

# マージソートの計算量

まず、マージの計算量  $M(n)$  を考えます。

明らかに、出来上がるソート列の長さに比例した時間量です。

$$\therefore M(n) = O(n)$$

マージソートの時間量を  $T(n)$  とします。

以下の再帰式を満たします。

$$\begin{cases} T(1) = c_1 \\ T(n) \leq T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + M(n) = 2T\left(\frac{n}{2}\right) + c_2 n \end{cases}$$

解析を簡単にするため、データを  $n = 2^k$  個あると仮定します。

$$\left\{ \begin{array}{l} T(1) = c_1 \\ T(n) \leq 2T\left(\frac{n}{2}\right) + c_2 n \end{array} \right. \quad \longrightarrow \quad \left\{ \begin{array}{l} T'(0) = c_1 \\ T'(k) \leq 2T'(k-1) + c_2 2^k \end{array} \right.$$

$$\begin{aligned} T'(k) &\leq 2T'(k-1) + c_2 2^k \\ &\leq 2(2T'(k-2) + c_2 2^{k-1}) + c_2 2^k = 4T'(k-2) + 2c_2 2^k \\ &\leq 4(2T'(k-3) + c_2 2^{k-2}) + 2c_2 2^k = 8T'(k-3) + 3c_2 2^k \\ &\vdots \\ &\leq 2^k T'(0) + c_2 k 2^k = (c_1 + c_2 k) 2^k \end{aligned}$$

$$\therefore T(n) \leq n(c_2 \log n + c_1) = c_2 n \log n + c_1 n$$

$$\therefore T(n) = O(n \log n)$$

$n \neq 2^k$  であるような一般の入力サイズに対しては、  
もう一段階解析の途中を考察する。

任意の  $n$  に対して、 $2^l \leq n < 2^{l+1}$  を満たす  $l$  が必ず存在する。

$$\text{よって、 } T(2^l) \leq T(n) < T(2^{l+1})$$

$$\therefore T(n) \leq T(2^{l+1}) = \{c_1 + c_2(l+1)\} 2^{l+1}$$

$$\text{一方 } l \leq \log n < l+1$$

$$\therefore \log n - 1 < l \leq \log n$$

$$\text{したがって、 } \therefore T(n) \leq \{c_1 + c_2(\log n + 1)\} 2^{\log n + 1}$$

$$= 2c_1n + 2c_2n \log n + 2c_2n$$

$$= 2c_2n \log n + 2n(c_1 + c_2)$$

$$\therefore T(n) = O(n \log n)$$

結局、どのような入力に対しても、マージソートの最悪時間計算量は、

$$O(n \log n)$$

である。

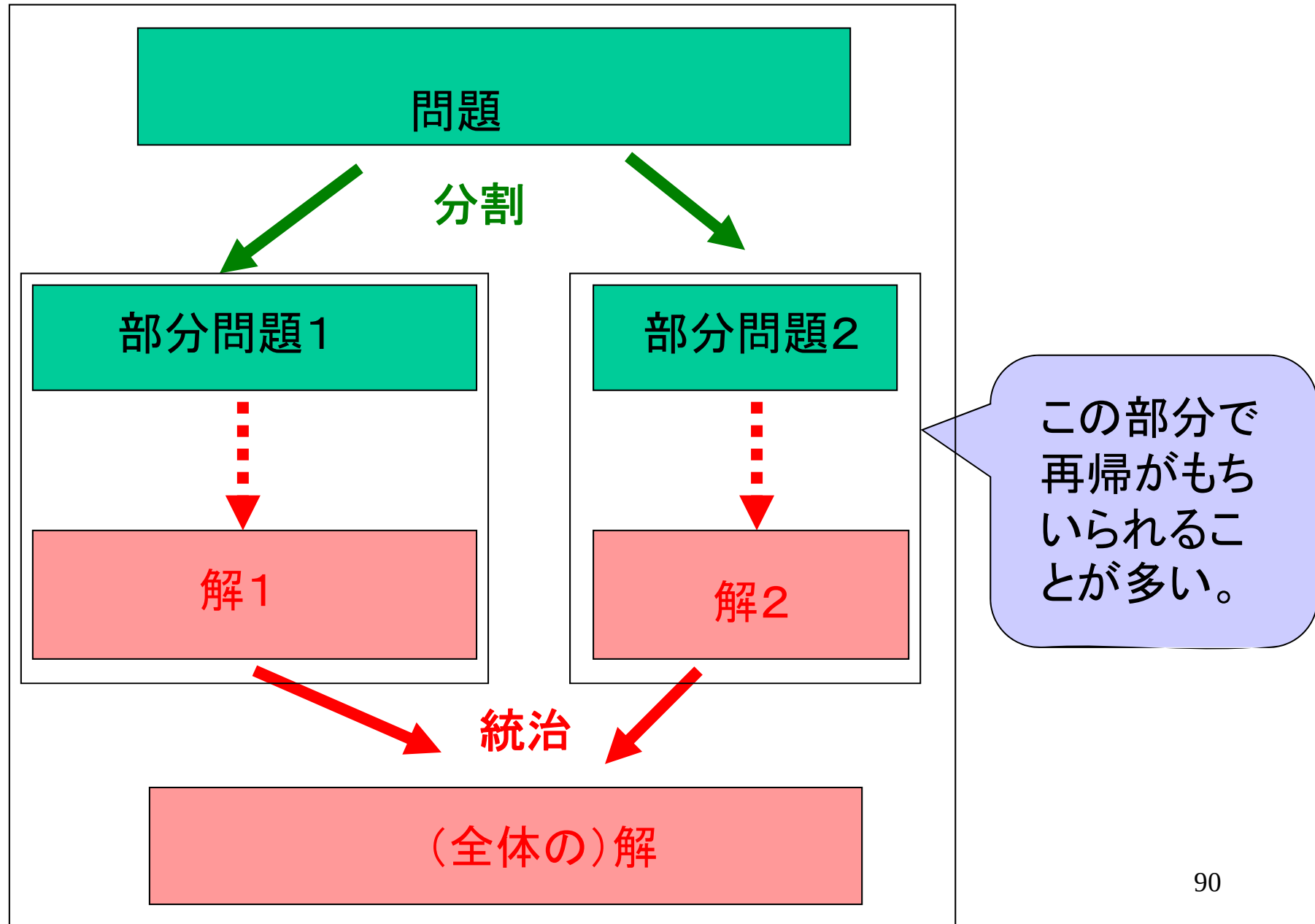
# 分割統治法について



# 分割統治法とは

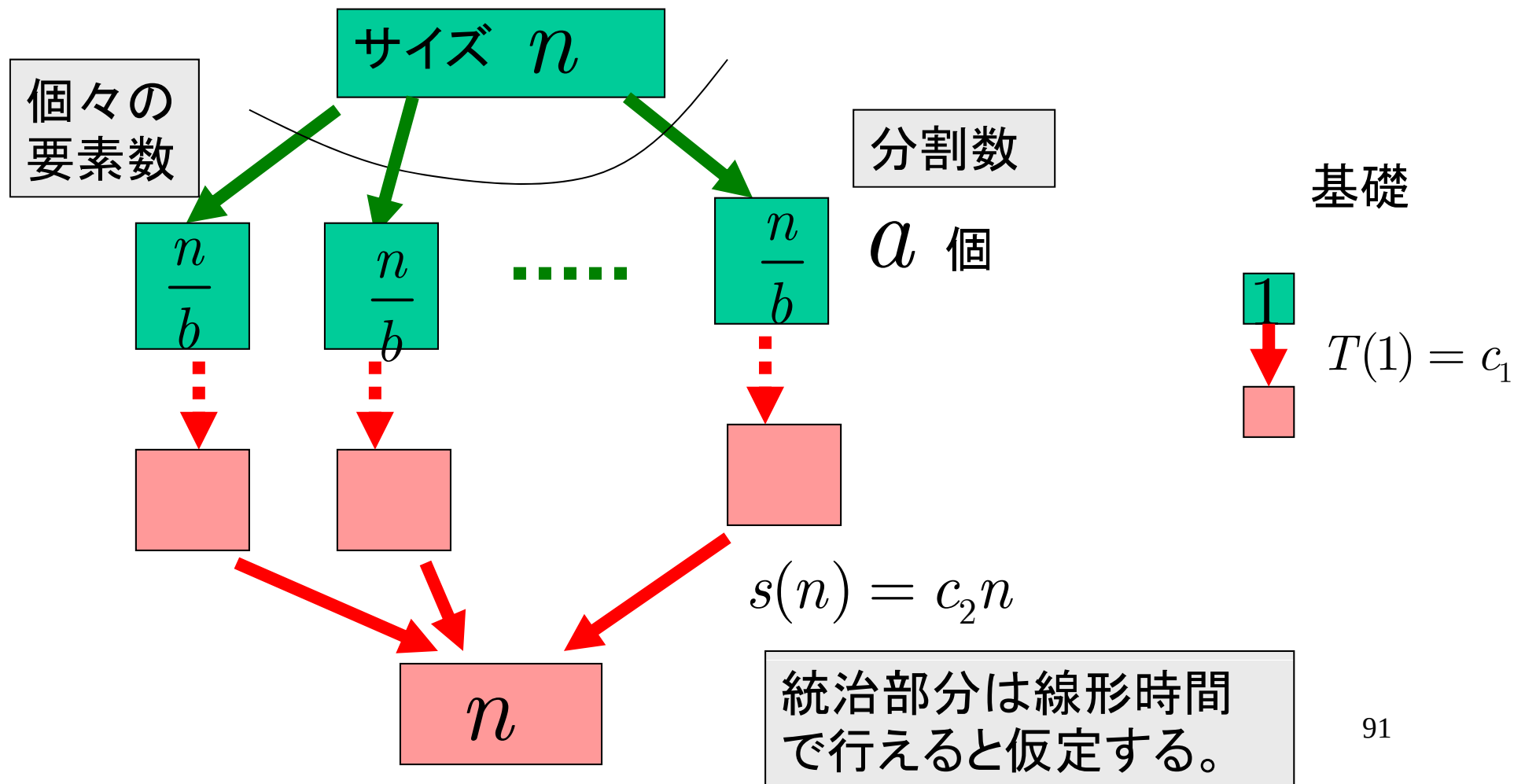
- 元の問題をサイズの小さいいくつかの部分問題に**分割**し、
- 個々の部分問題を何らかの方法で解決し、
- それらの解を**統合**することによって、元の問題を解決する方法のことである。
- (分割統治法に基づくアルゴリズムは、再帰を用いると比較的に容易に記述することができる。)

# 分割統治法のイメージ



# 分割統治法の時間計算量

ここでは、より一般的な分割統治法における計算量を考察する。



一般的な分割統治法における時間計算量 $T(n)$ は、次の漸化式で表されることが多い。

$$\begin{cases} T(1) = c_1 & (n = 1) \\ T(n) = aT\left(\frac{n}{b}\right) + c_2n & (n > 1) \end{cases}$$

この漸化式を解く。

$$T(n) = aT\left(\frac{n}{b}\right) + c_2n$$

$\frac{n}{b}$  を代入して次式を得る。

$$T\left(\frac{n}{b}\right) = aT\left(\frac{n}{b^2}\right) + c_2 \frac{n}{b}$$

この式を上式に代入する。

$$\begin{aligned}
T(n) &= aT\left(\frac{n}{b}\right) + c_2n \\
&= a\left\{aT\left(\frac{n}{b^2}\right) + c_2\frac{n}{b}\right\} + c_2n = a^2T\left(\frac{n}{b^2}\right) + c_2n\left(1 + \frac{a}{b}\right) \\
&= a^2\left\{aT\left(\frac{n}{b^3}\right) + c_2\frac{n}{b^2}\right\} + c_2n\left(1 + \frac{a}{b}\right) = a^3T\left(\frac{n}{b^3}\right) + c_2n\left\{1 + \frac{a}{b} + \left(\frac{a}{b}\right)^2\right\} \\
&\vdots \\
&= a^kT\left(\frac{n}{b^k}\right) + c_2n\left\{1 + \frac{a}{b} + \cdots + \left(\frac{a}{b}\right)^{k-1}\right\} = a^kT\left(\frac{n}{b^k}\right) + c_2n\sum_{i=0}^{k-1}\left(\frac{a}{b}\right)^i
\end{aligned}$$

$a$  と  $b$  の大小関係で式が異なる。

等比級数の和

ここで、 $n = b^k$  と仮定する。 $k = \log_b n$   
 (一般の $n$ でもほぼ同様に求めることができる。)

場合1:  $a < b$  すなわち  $\frac{a}{b} < 1$  のとき

$$\begin{aligned} T(n) &= a^k T\left(\frac{n}{b^k}\right) + c_2 n \sum_{i=0}^{k-1} \left(\frac{a}{b}\right)^i \\ &= a^k T(1) + c_2 n \frac{1 - \left(\frac{a}{b}\right)^k}{1 - \frac{a}{b}} \\ &\leq c_1 b^k + c_2 n \frac{1}{1 - \frac{a}{b}} = \left(c_1 + c_2 \frac{b}{b-a}\right)n \end{aligned}$$

$$\therefore T(n) = O(n)$$

この場合は線形時間アルゴリズムが得られる。

場合2:  $a = b$  すなわち  $\frac{a}{b} = 1$  のとき

$$T(n) = a^k T\left(\frac{n}{b^k}\right) + c_2 n \sum_{i=0}^{k-1} \left(\frac{a}{b}\right)^i$$

$$= a^k T(1) + c_2 n \sum_{i=0}^{k-1} 1$$

$$= c_1 b^k + c_2 n k$$

$$= c_1 n + c_2 n \log_b n$$

$$\therefore T(n) = O(n \log n)$$

この場合は、典型的な  $O(n \log n)$  時間の  
アルゴリズムが得られる。

場合3:  $a > b$  すなわち  $\frac{a}{b} > 1$  のとき

$$\begin{aligned} T(n) &= a^k T\left(\frac{n}{b^k}\right) + c_2 n \sum_{i=0}^{k-1} \left(\frac{a}{b}\right)^i \\ &= a^k T(1) + c_2 n \frac{\left(\frac{a}{b}\right)^k - 1}{\frac{a}{b} - 1} \\ &= c_1 a^k + c_2 \frac{b}{a - b} (a^k - n) \quad (\because n = b^k) \\ &\leq \left(c_1 + c_2 \frac{b}{a - b}\right) a^k = \left(c_1 + c_2 \frac{b}{a - b}\right) n^{\log_b a} \end{aligned}$$

ここで、 $p \equiv \log_b a > 1$  とおく。

$$\therefore T(n) = O(n^p)$$

この場合は指数時間アルゴリズムになってしまう。



## 分割統治法の計算時間のまとめ

- 分割数(a)がサイズ縮小(b)より小さい場合には、線形時間アルゴリズム
- 分割数(a)とサイズ縮小(b)が等しい場合には、 $O(n \log n)$  時間のアルゴリズム  
(マージソートがこの場合に相当する。)
- 分割数(a)がサイズ縮小(b)より大きい場合  
指数時間アルゴリズムになってしまう。

## 3-3: 高度なソートアルゴリズム②

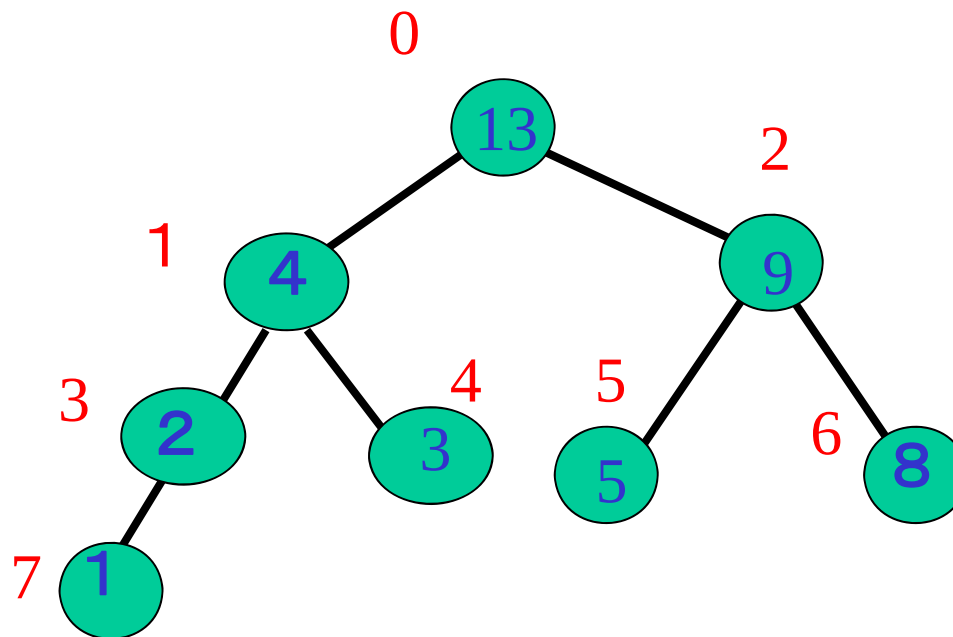
### (データ構造にもとづくソート)

# ヒープソート

# ヒープソートの方針

## 方針

- ヒープを使ってソートする。
- 先頭から順にヒープに挿入し、データ全体をヒープ化する。
- 最大値を取り出して、最後のデータにする。



# ヒープとは

データ構造の一種。

(最大や、最小を効率良く見つけることができる。)

$n$  点からなるヒープとは、次の条件を満足する2分木。

● 深さ  $\lfloor \log_2 n \rfloor - 1$  までは、完全2分木。

● 深さ  $\lfloor \log_2 n \rfloor$  では、要素を左につめた木。

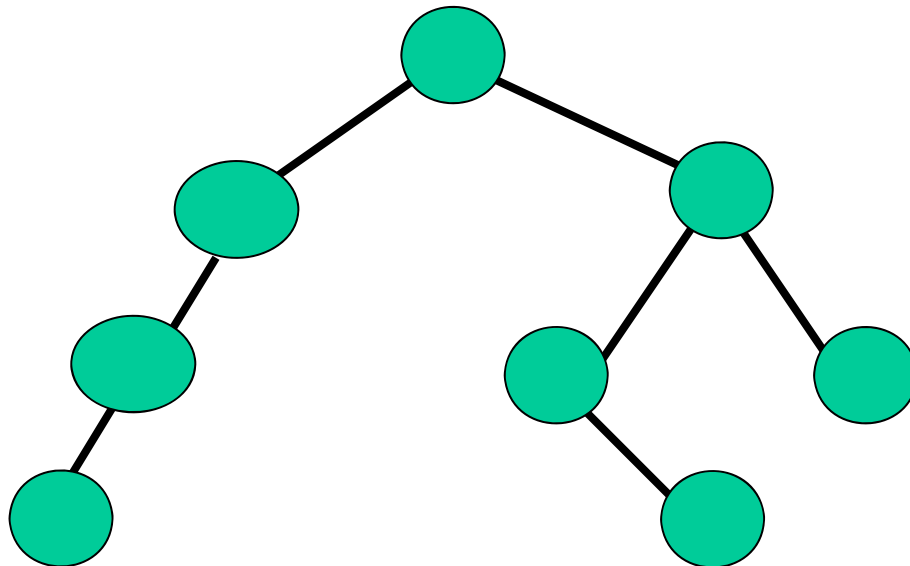
● 全ての節点において、  
親の値が子の値より小さい(大きい。)

この条件は、ある節点の値は、その子孫の節点全ての値より、小さい(大きい)とすることもできる。

まず、このデータ構造(ヒープ)に関することを順に見ていく。

# 2分木

- 高々2つ子しかない木。
- 左と右の子を区別する。

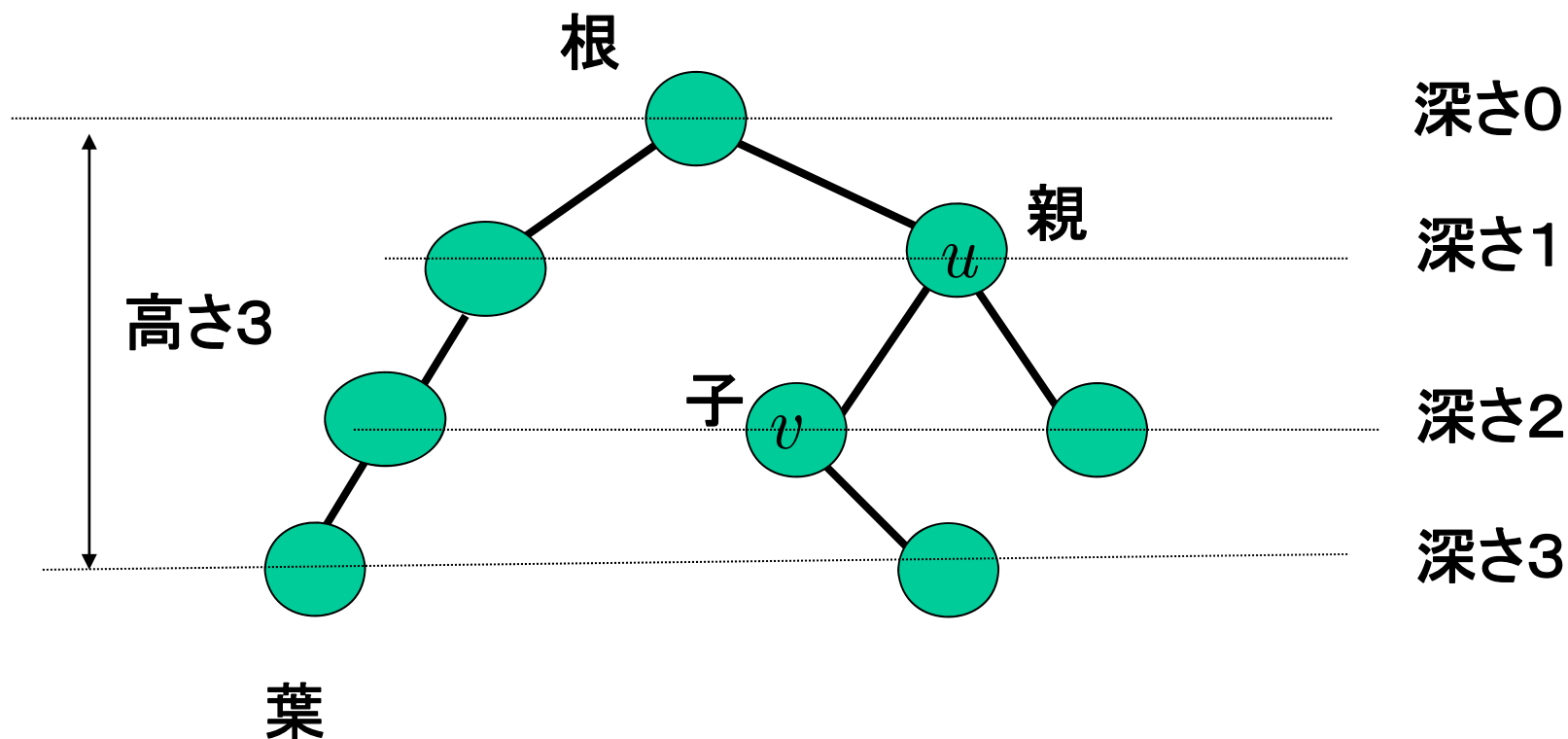


2分木においては、左と右の子を区別するので、  
次の2つの2分木は同一ではない。



# 木に関する用語

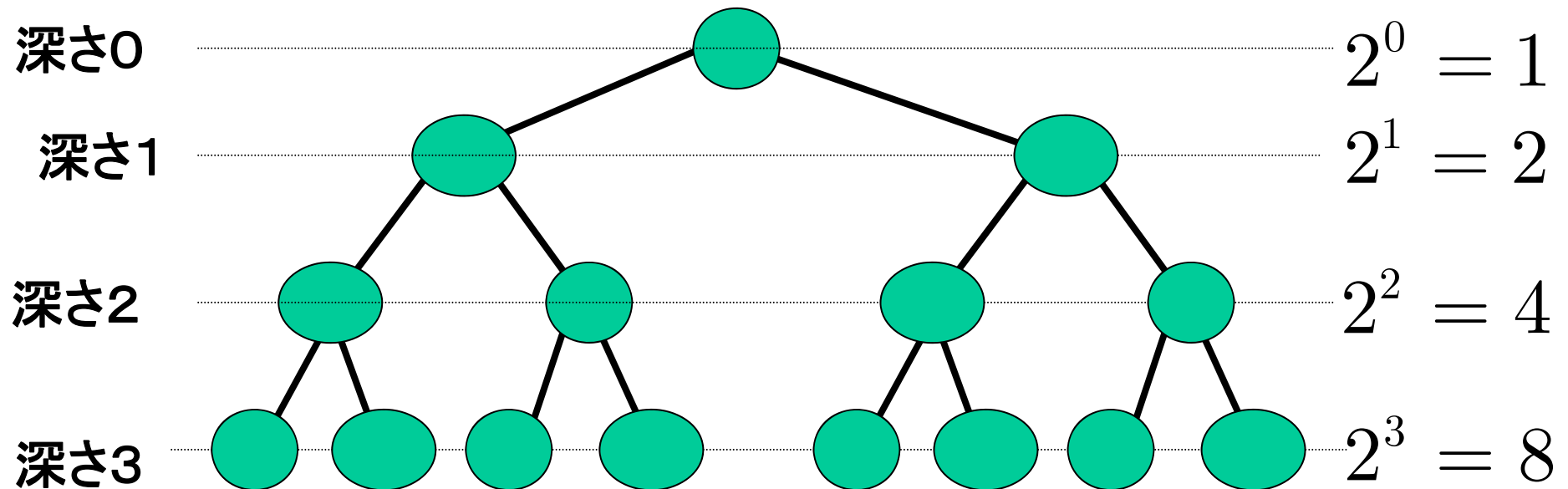
- 深さ: 根までの道の長さ
- 高さ: 木中の最大の深さ





# 完全2分木

● 全ての内部節点(葉以外の節点)が、すべて2つの子を持つ2分木。



### 命題HP1(完全2分木と節点数)

- (1) 完全2分木の、深さ  $d$  には  $2^d$  個の節点がある。
- (2) 高さ  $h$  の完全2分木には  $2^{h+1} - 1$  個の節点がある。

#### 証明

(1)

深さ $d$ に関する数学的帰納法で証明できる。

基礎:

このときは、深さ0の頂点は根ただ一つなので、命題は成り立つ。

帰納:

深さ  $d$  の節点が  $2^d$  個あると仮定する。

このとき、これらの節点すべてが、2つの子を持つので、

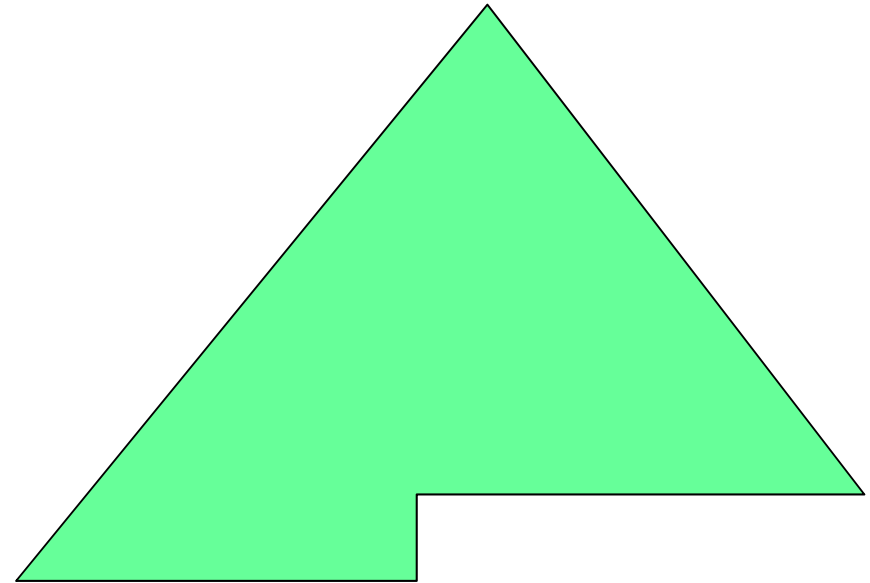
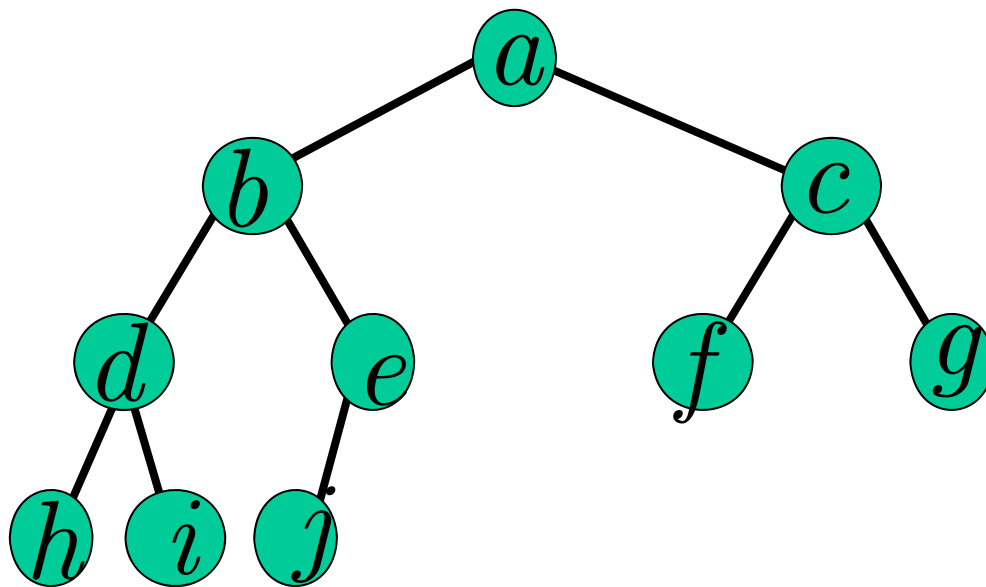
深さ  $d + 1$  の節点数は、 $2 \times 2^d = 2^{d+1}$  あり、命題は成り立つ。

(2) (1)より、節点の総数は、次式で表される。

$$\sum_{d=0}^h 2^d = \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1$$

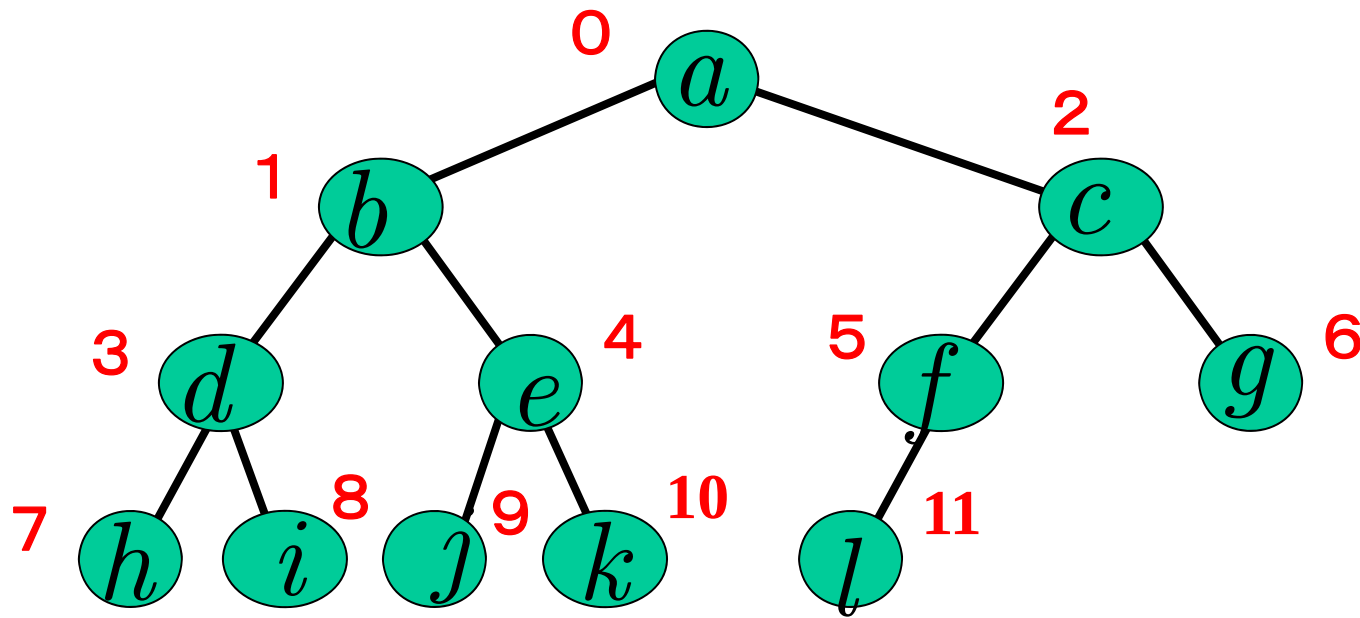
*QED*

# ヒープの形



このような形で、  
イメージするとよい。

# ヒープ番号と配列での実現



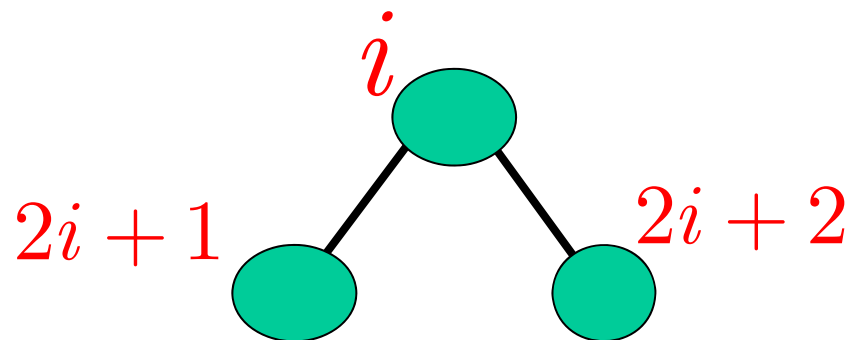
配列

|    | 0        | 1        | 2        | 3        | 4        | 5        | 6        | 7        | 8        | 9        | 10       | 11       |
|----|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| HP | <i>a</i> | <i>b</i> | <i>c</i> | <i>d</i> | <i>e</i> | <i>f</i> | <i>g</i> | <i>h</i> | <i>i</i> | <i>j</i> | <i>k</i> | <i>l</i> |

# ヒープにおける親子関係

## 命題HP2(ヒープにおける親子関係)

ヒープ番号  $i$  の節点に対して、  
左子のヒープ番号は  $2i + 1$  であり、  
右子のヒープ番号は  $2i + 2$  である。



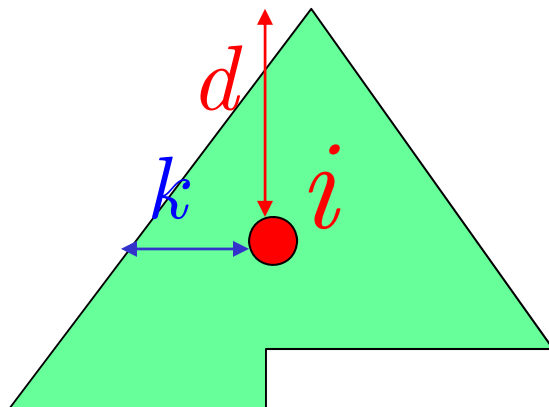
## 証明

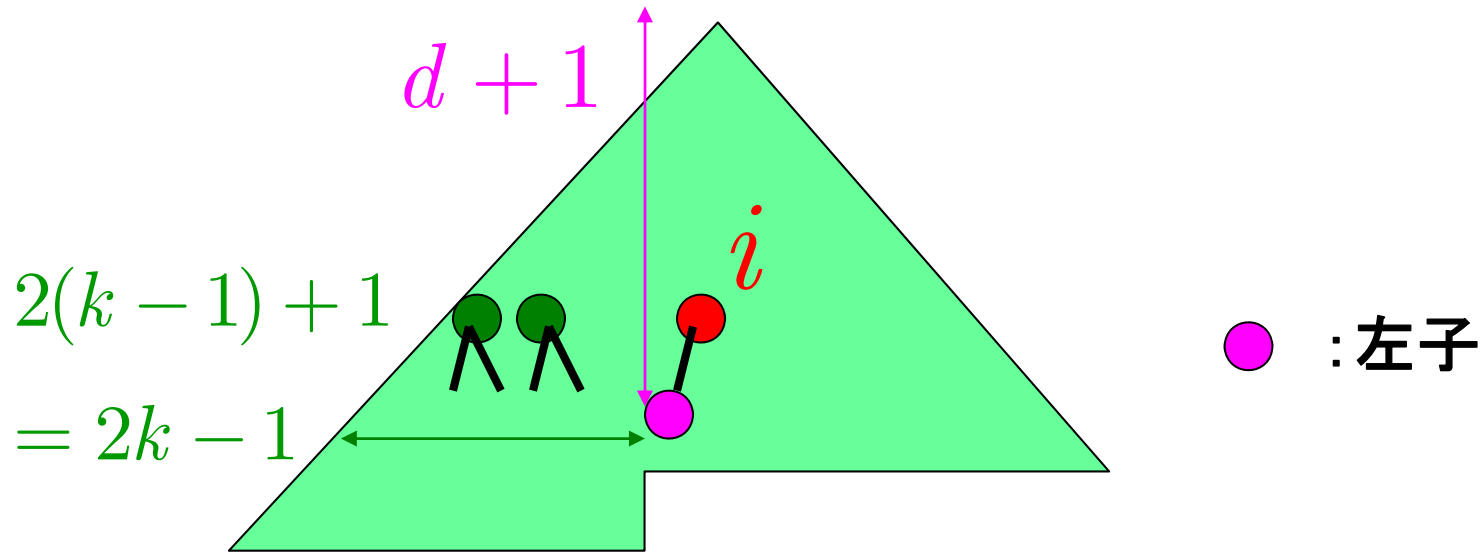
右子は、左子より1大きいヒープ番号を持つことはあきらかなので、左子が  $2i + 1$  であることだけを示す。

今、節点  $i$  の深さを  $d$  とし、左から  $k$  番目であるとする。  
すなわち、

$$i = (2^d - 1) + (k) - 1 = 2^d + k - 2$$

が成り立つ。





このとき、左子は、深さ  $d + 1$  で左から  $2k - 1$  番目の節点である。

今、左子のヒープ番号を  $h_l$  とすると次の式がなりたつ。

$$h_l = 2^{d+1} + 2k - 1 - 2$$

$$= 2(d^d + k - 2) + 1$$

$$= 2i + 1$$

*QED*<sub>112</sub>

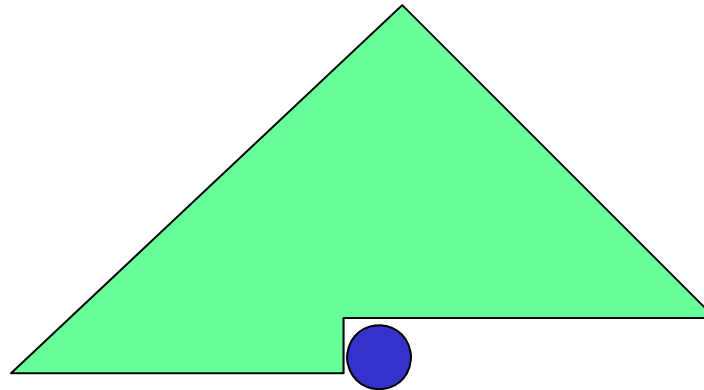


# ヒープにおける挿入

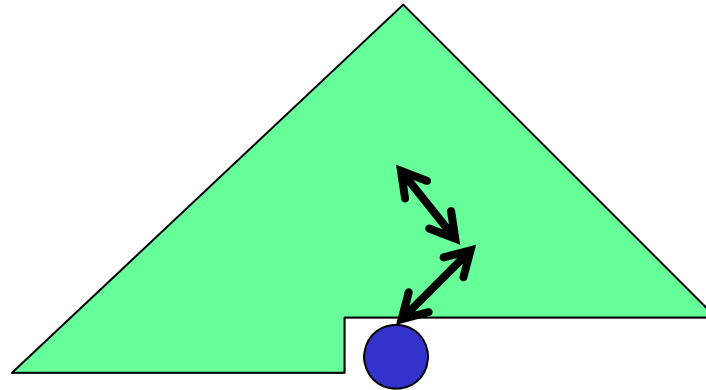
(ヒープ条件)

全ての節点において、親の値が子の値より小さい

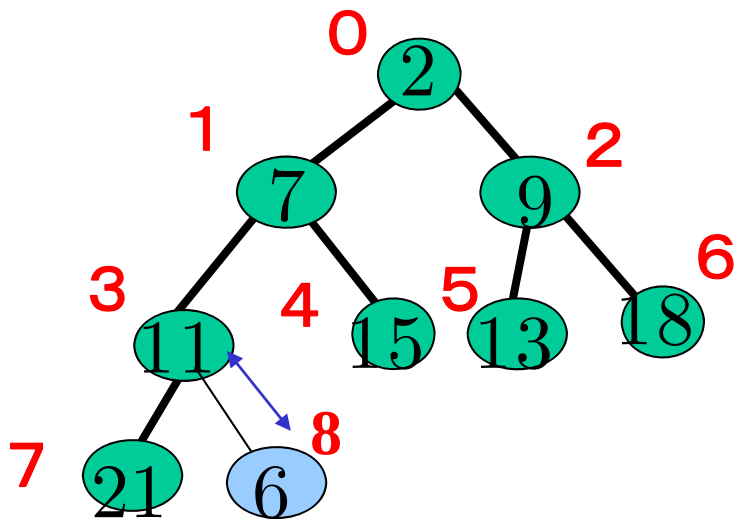
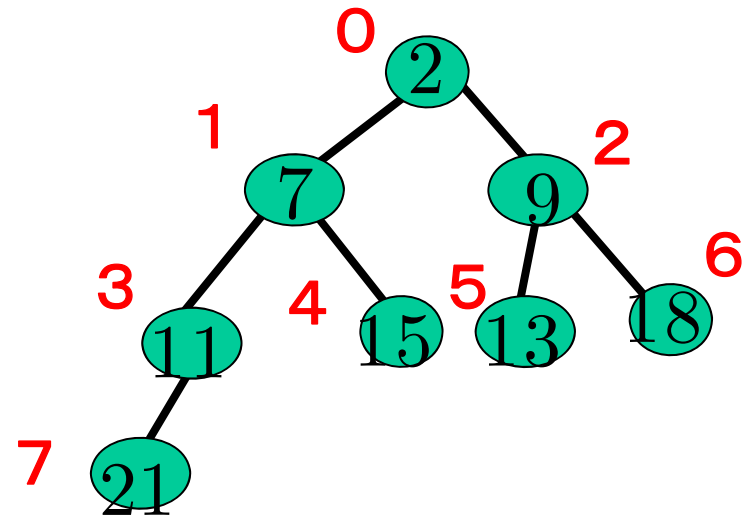
$n$  点保持しているヒープに新たに1点挿入することを考える。  
このとき、ヒープの形より、ヒープ番号  $n + 1$  の位置に最初におかれる。



しかし、これだけだと、ヒープ条件を満たさない可能性がある  
があるので、根のほうに向かって条件がみたされるまで  
交換を繰り返す。(アップヒープ)



# 挿入の例

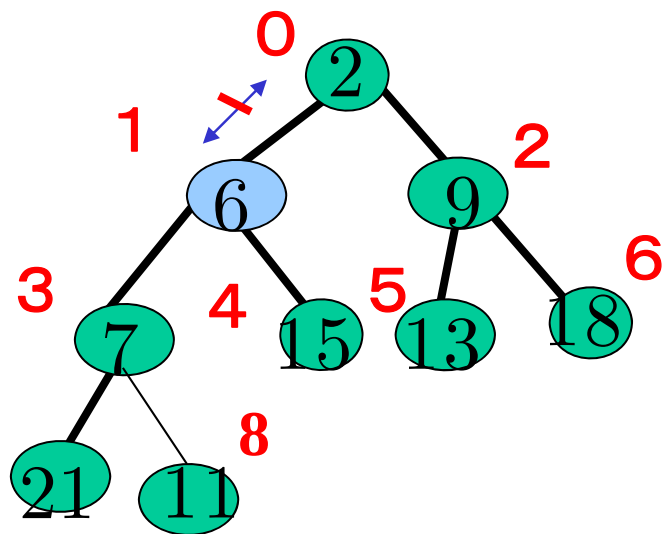
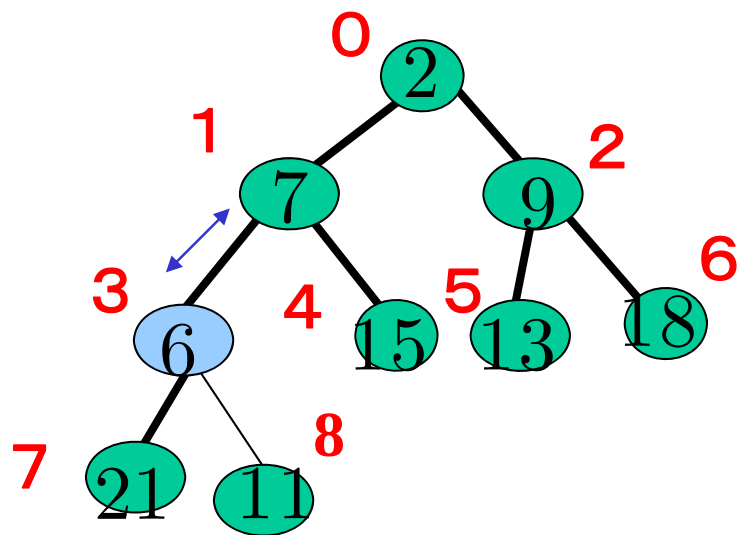


配列 HP

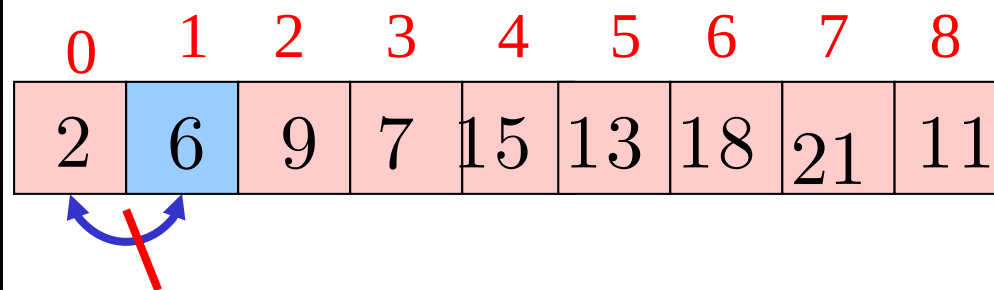
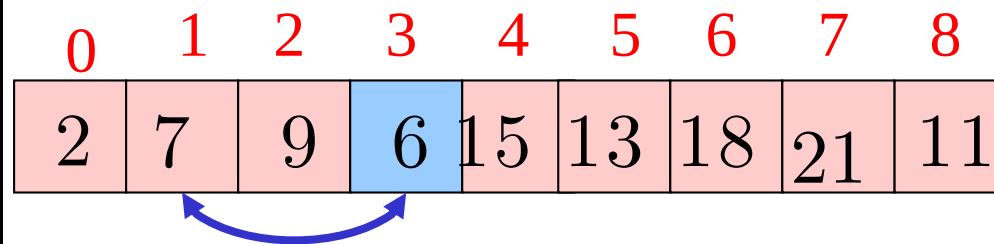
| 0 | 1 | 2 | 3  | 4  | 5  | 6  | 7  |
|---|---|---|----|----|----|----|----|
| 2 | 7 | 9 | 11 | 15 | 13 | 18 | 21 |

| 0 | 1 | 2 | 3  | 4  | 5  | 6  | 7  | 8 |
|---|---|---|----|----|----|----|----|---|
| 2 | 7 | 9 | 11 | 15 | 13 | 18 | 21 | 6 |





插入終了



# 練習

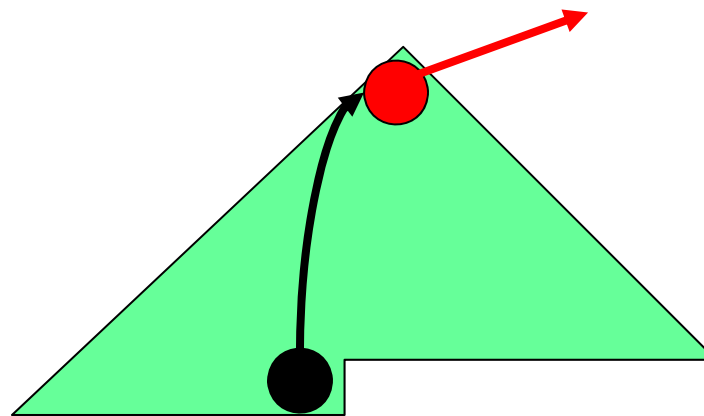
前のスライドのヒープに新たに、3を挿入し、その動作を木と、配列の両方で示せ。

# ヒープにおける削除

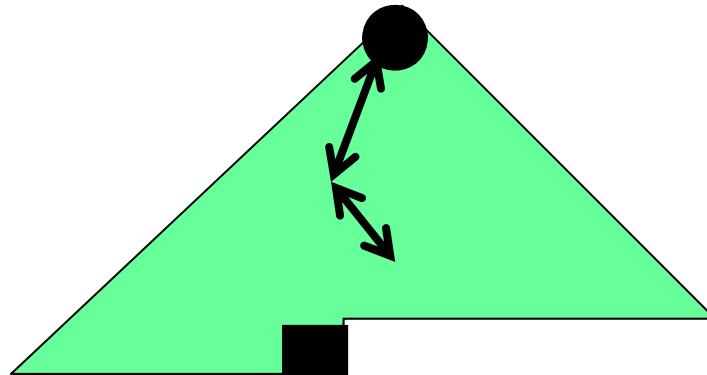
(ヒープ条件)

全ての節点において、親の値が子の値より小さい

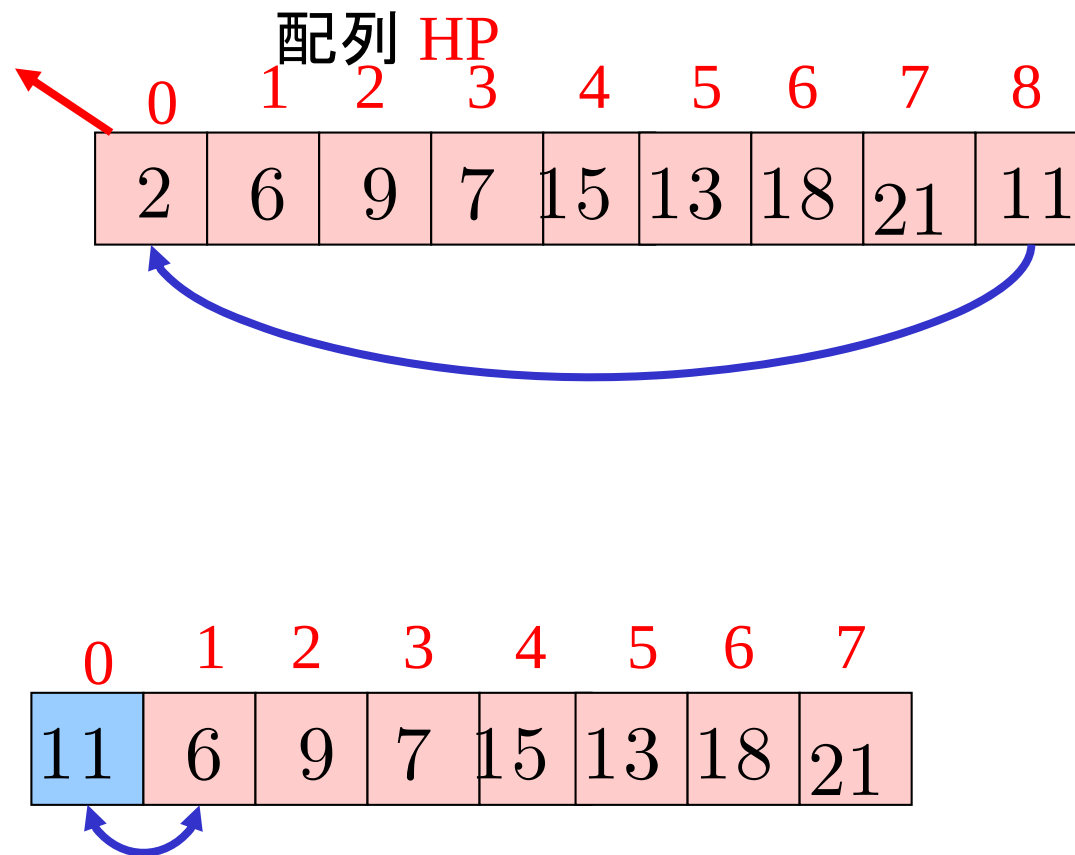
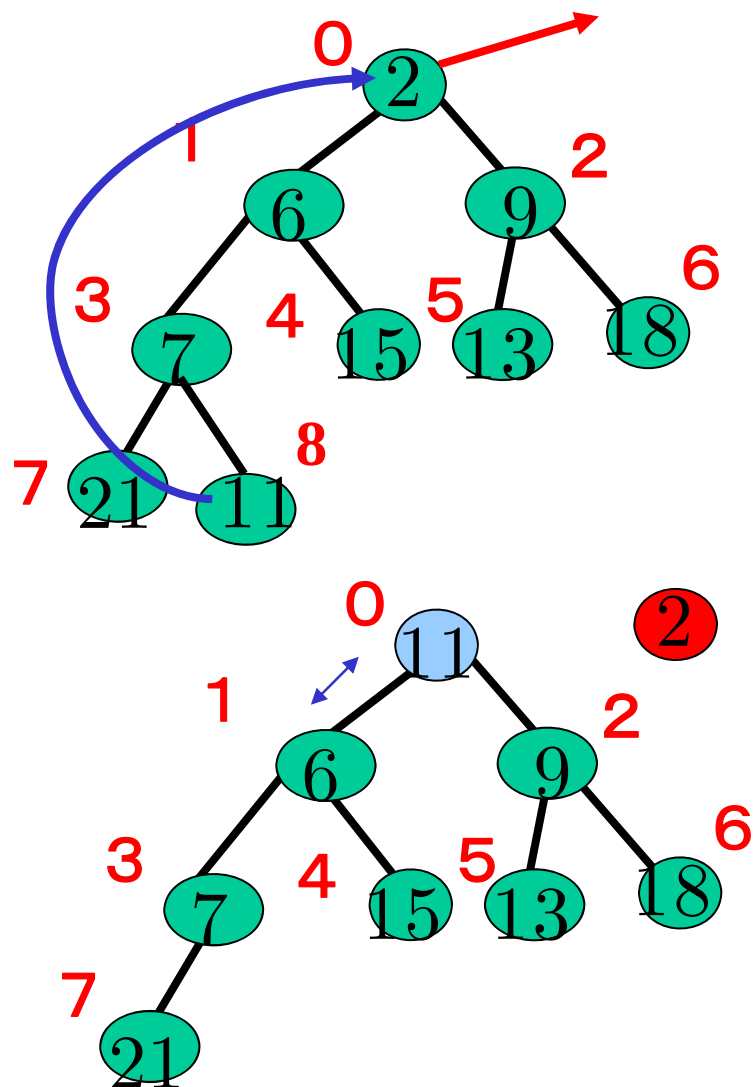
ヒープにおいては、先頭の最小値のみ削除される。  
削除の際には、ヒープの形を考慮して、  
ヒープ番号  $n$  の節点の値を根に移動する。



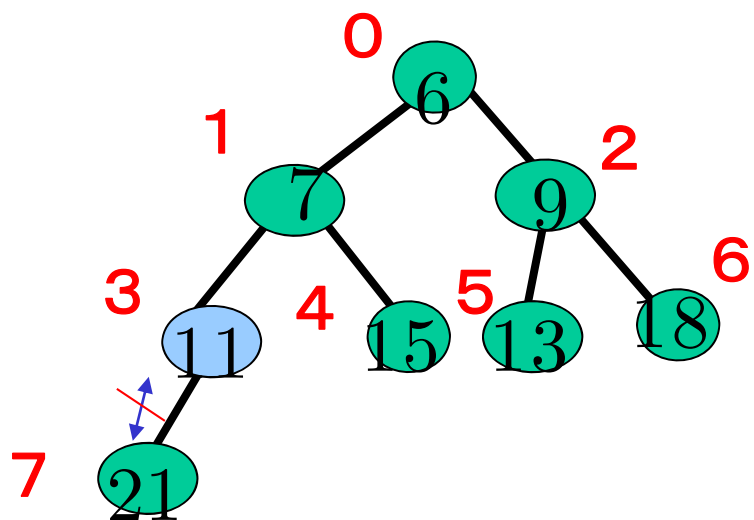
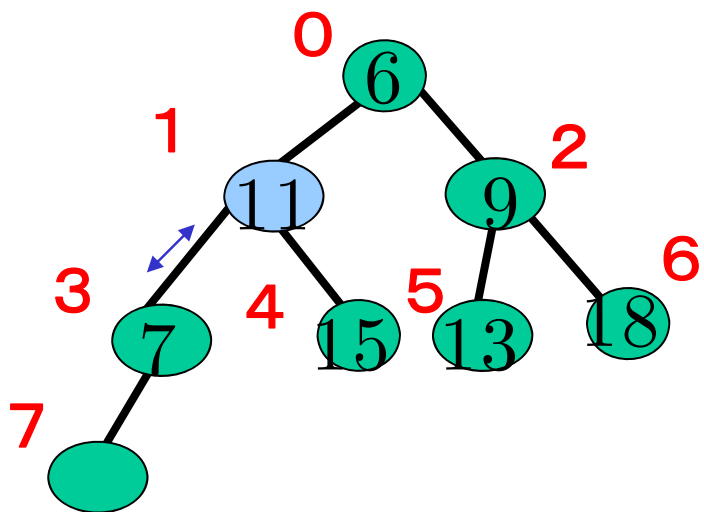
しかし、これだけだと、ヒープ条件を満たさない可能性がある  
があるので、葉のほうに向かって条件がみたされるまで  
交換を繰り返す。(ダウンヒープ)  
交換は、値の小さい子供の方と交換する。  
これをヒープ条件が満たされるまで繰り返す。



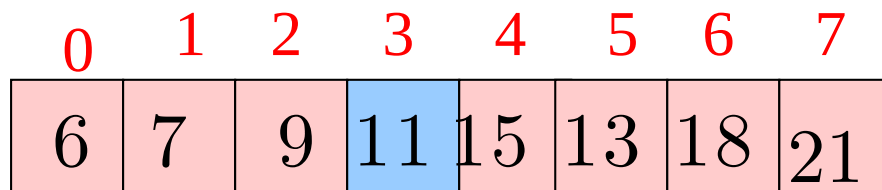
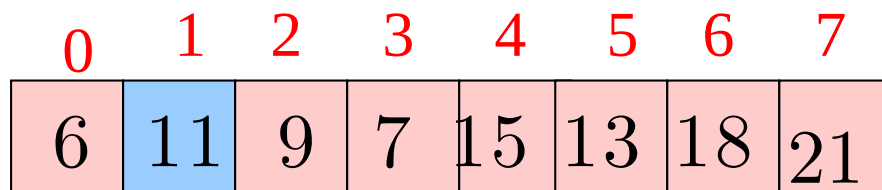
# 削除の例







削除終了



# ヒープソートの動き前半 (ヒープへ値挿入)

配列

|   |   |   |   |   |   |    |   |   |
|---|---|---|---|---|---|----|---|---|
|   | 0 | 1 | 2 | 3 | 4 | 5  | 6 | 7 |
| A | 5 | 3 | 8 | 1 | 4 | 13 | 9 | 2 |

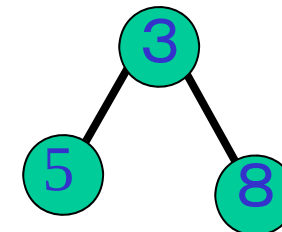
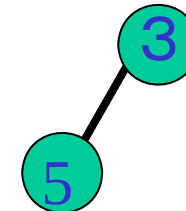
|   |   |   |   |   |    |   |   |
|---|---|---|---|---|----|---|---|
| 5 | 3 | 8 | 1 | 4 | 13 | 9 | 2 |
|---|---|---|---|---|----|---|---|

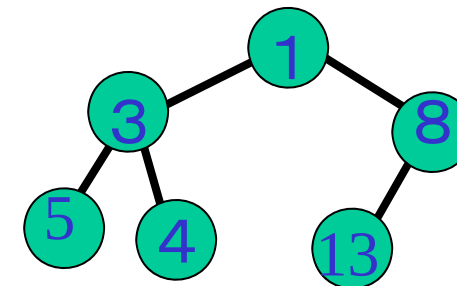
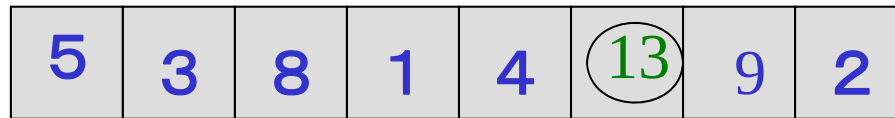
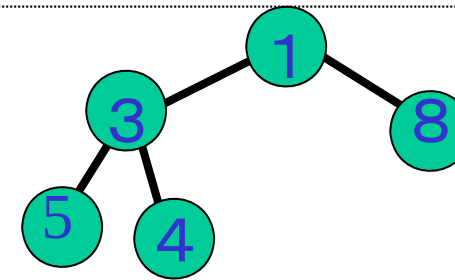
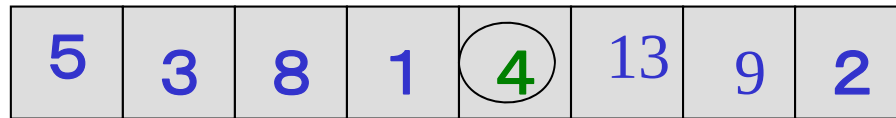
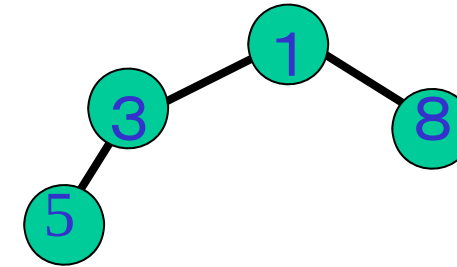
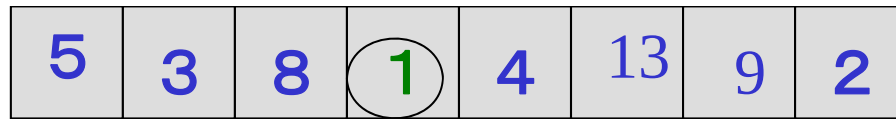
|   |   |   |   |   |    |   |   |
|---|---|---|---|---|----|---|---|
| 5 | 3 | 8 | 1 | 4 | 13 | 9 | 2 |
|---|---|---|---|---|----|---|---|

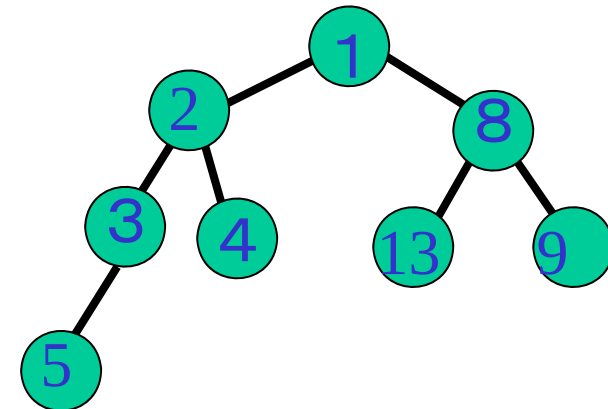
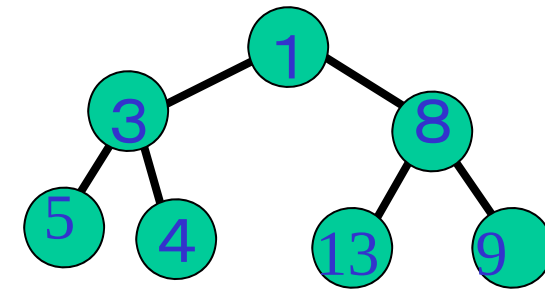
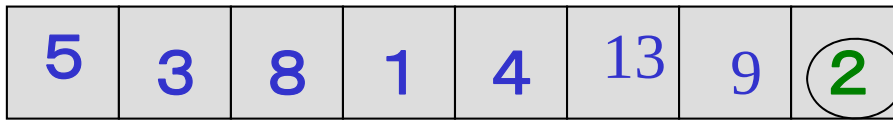
ヒープ

$\phi$

5

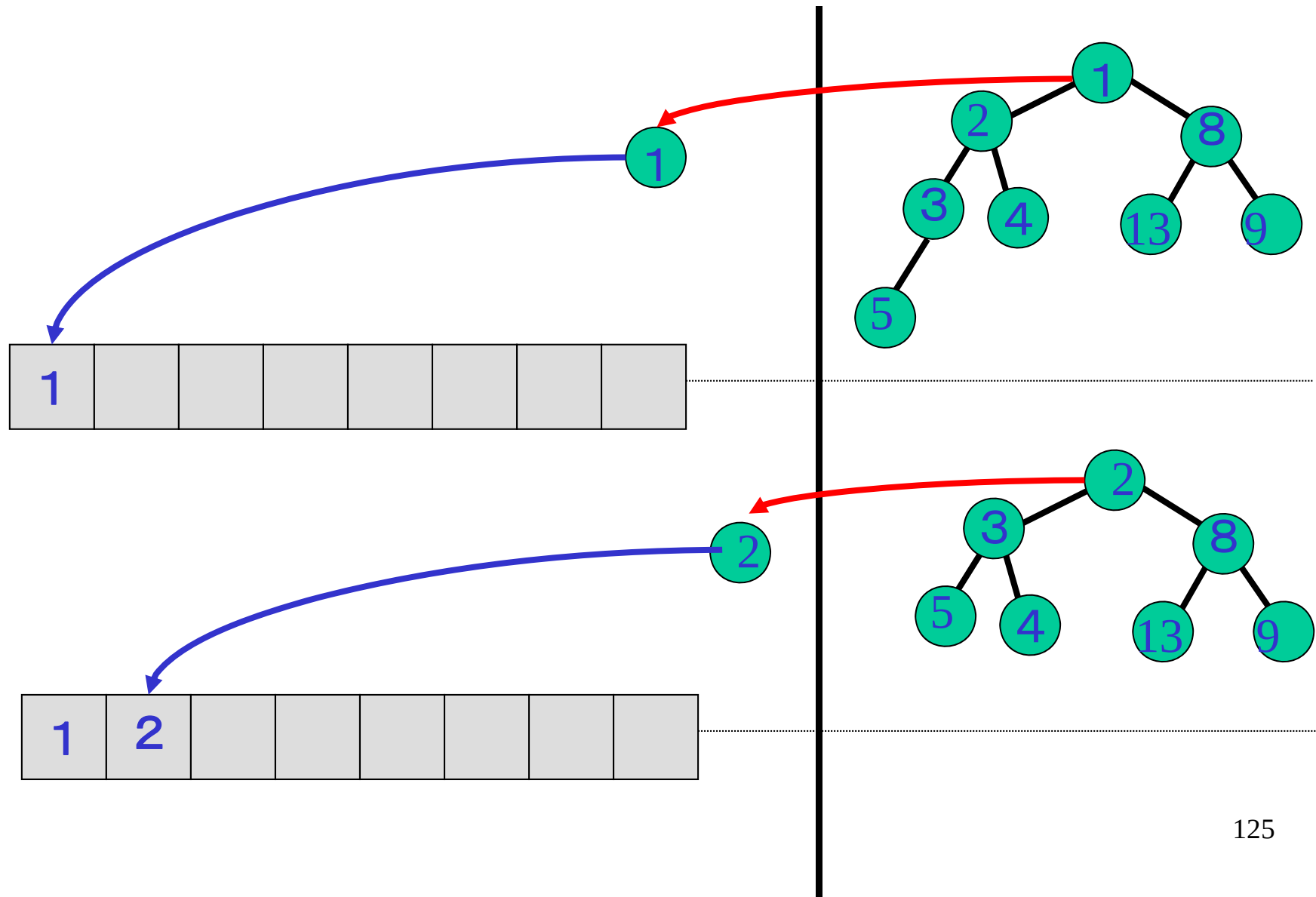


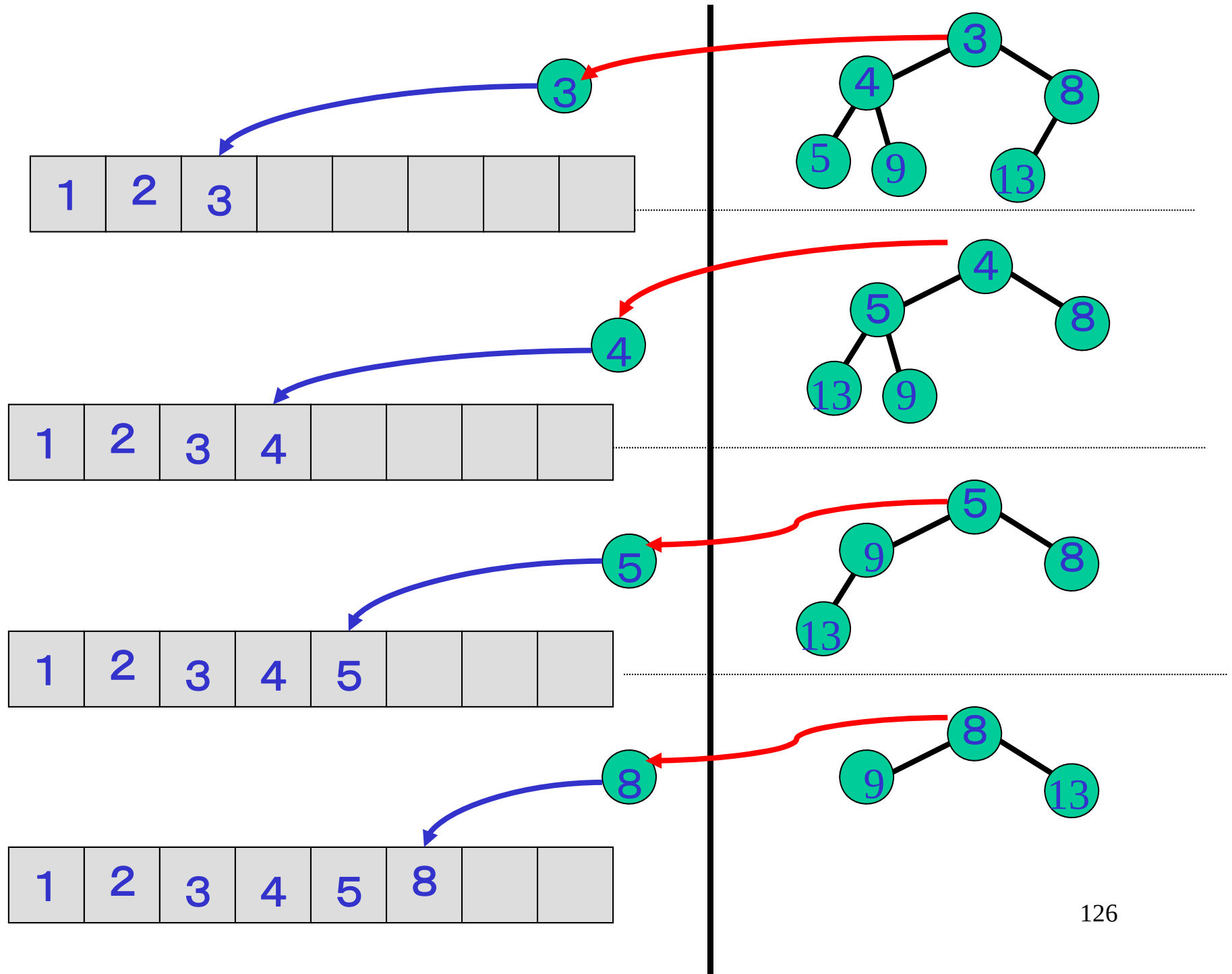




# ヒープソートの動き後半 (ヒープからの最小値削除)

ヒープ







# 練習

次の配列を、ヒープソートでソートするとき、  
ヒープの動作と配列の動きをシミュレートせよ。

|    |    |    |   |   |   |    |   |
|----|----|----|---|---|---|----|---|
| 11 | 25 | 21 | 1 | 8 | 3 | 16 | 5 |
|----|----|----|---|---|---|----|---|



# ヒープソートの実現

```
1. void heap_sort()
2. {
3.     int i;      /* カウンタ */
4.     make_heap(); /* 空のヒープの作成 */
5.     /* ヒープ挿入 */
6.     for(i=0; i<n; i++) {
7.         insert_heap(A[i]);
8.     }
9.
10.    /* ヒープからのデータの取り出し */
11.    for(i=0; i<n; i++) {
12.        A[i] = get_min();
13.    }
14.    return ;
15. }
```

細かい部分  
は省略します。

### 命題HP3 (ヒープソートの正当性)

ヒープソートにより、配列はソートされる。

証明

ヒープの削除 (`get_min()`) により、  
繰り返し最小値を求められることに注意すれば、  
帰納法により証明できる。

*QED*<sub>130</sub>

# ヒープソートの計算量

ヒープへのデータ挿入

操作insert\_heap(A) 1回あたり、時間量  $O(\log n)$

n回行うので、時間量  $O(n \log n)$

整列(最小値の削除の反復)

操作get\_min() 1回あたり、時間量  $O(\log n)$

n回行うので、時間量  $O(n \log n)$

---

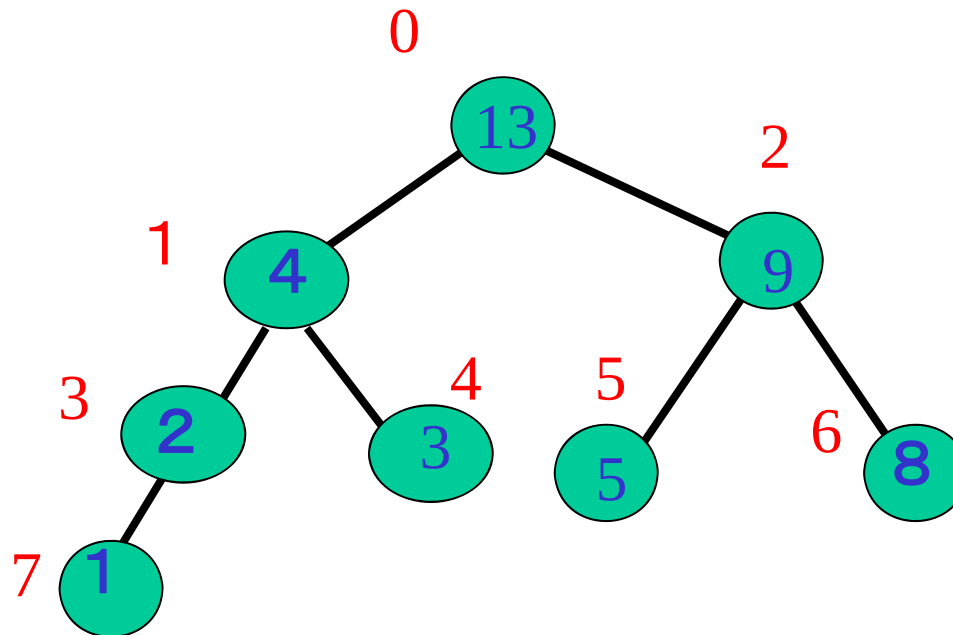
最悪時間量  $O(n \log n)$  のアルゴリズム

# 単一配列でのヒープソート

# 単一配列に変更する方針

方針

- 根が最大値になるように変更する。
- 先頭から順にヒープに挿入し、データ全体をヒープ化する。
- 最大値を取り出して、最後のデータにする。



# ヒープ条件の変更

(ヒープ条件)

全ての節点において、親の値が子の値より大きい

# ヒープ化 (ヒープソート前半)

配列

0 1 2 3 4 5 6 7

A

|   |   |   |   |   |    |   |   |
|---|---|---|---|---|----|---|---|
| 5 | 3 | 8 | 1 | 4 | 13 | 9 | 2 |
|---|---|---|---|---|----|---|---|

|   |   |   |   |   |    |   |   |
|---|---|---|---|---|----|---|---|
| 5 | 3 | 8 | 1 | 4 | 13 | 9 | 2 |
|---|---|---|---|---|----|---|---|

ヒープ

未ヒープ

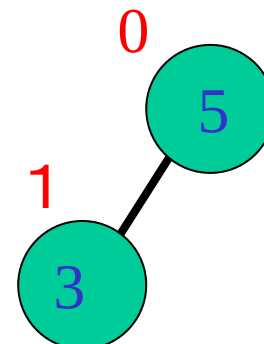
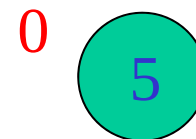
|   |   |   |   |   |    |   |   |
|---|---|---|---|---|----|---|---|
| 5 | 3 | 8 | 1 | 4 | 13 | 9 | 2 |
|---|---|---|---|---|----|---|---|

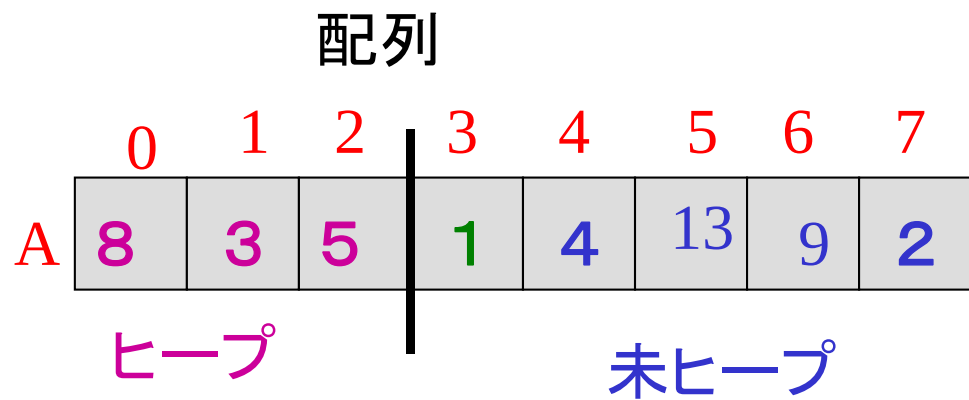
ヒープ

未ヒープ

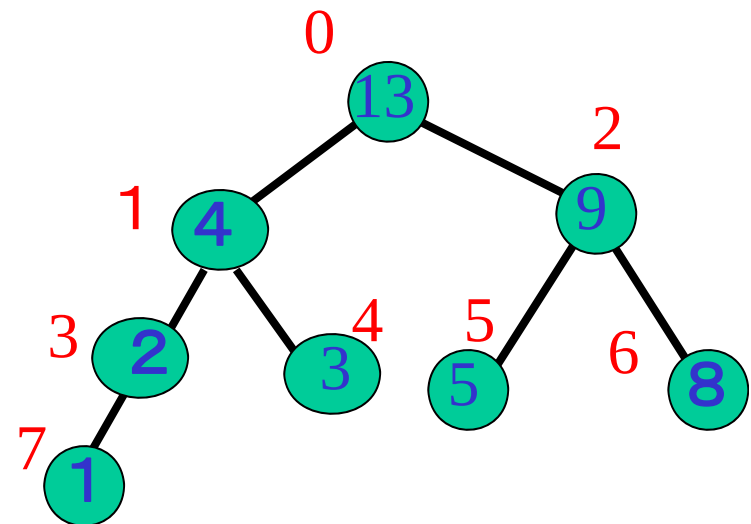
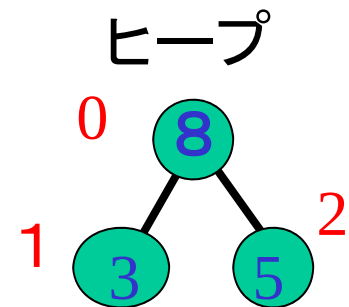
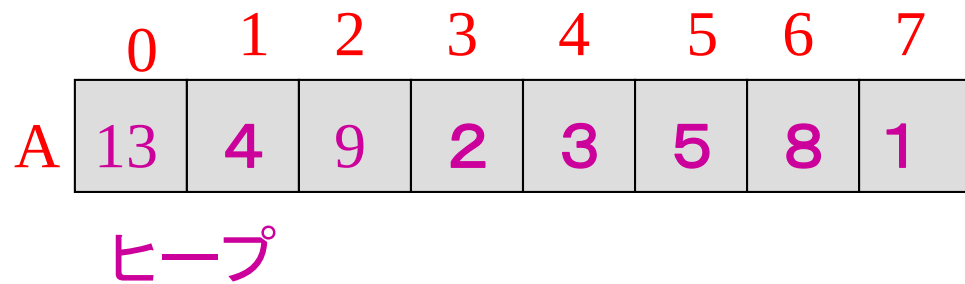
ヒープ

$\emptyset$



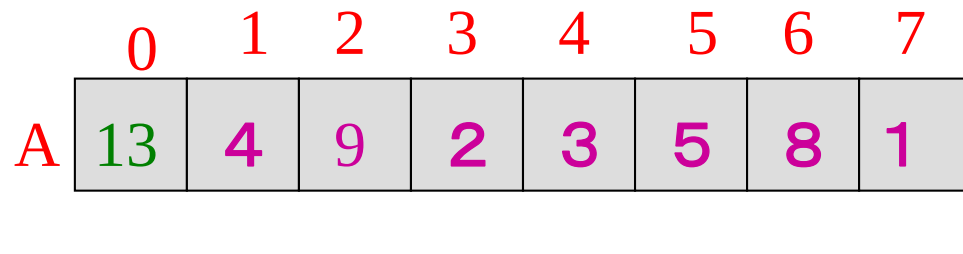


ちょっと省略



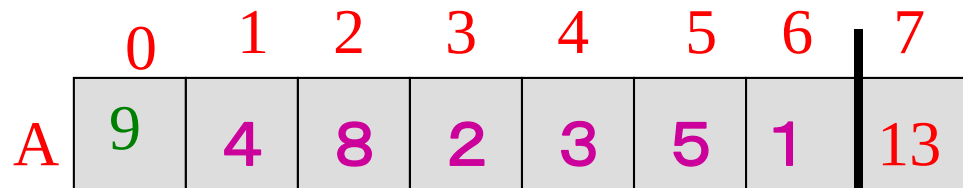


# 最大値の削除とソート



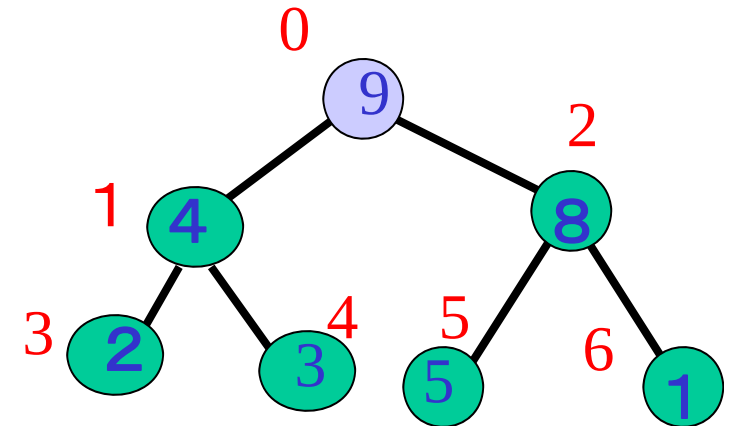
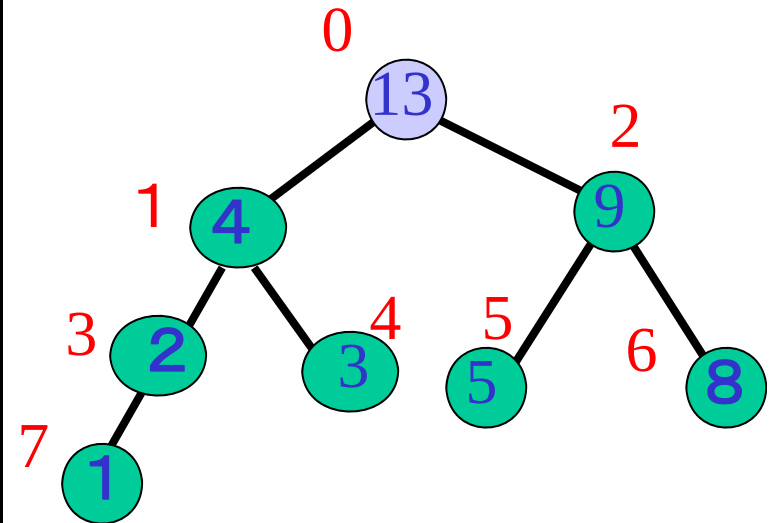
ヒープ

最大要素を、ヒープから削除し  
後ろに挿入

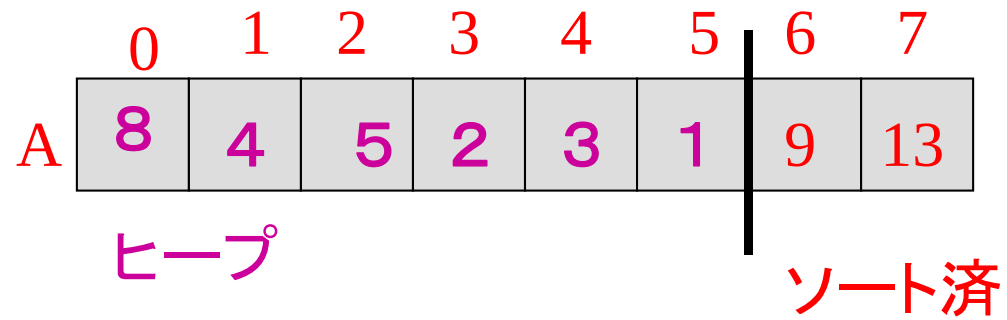


ヒープ

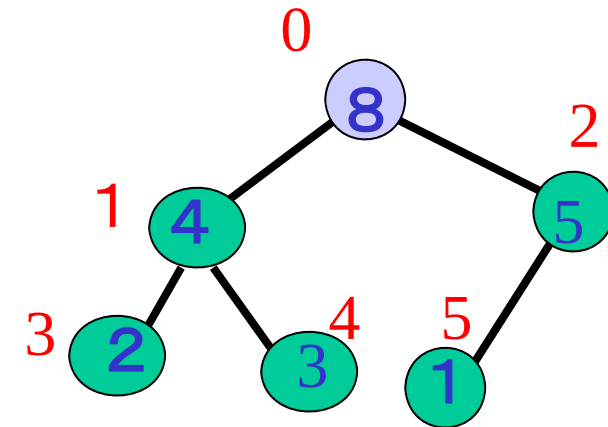
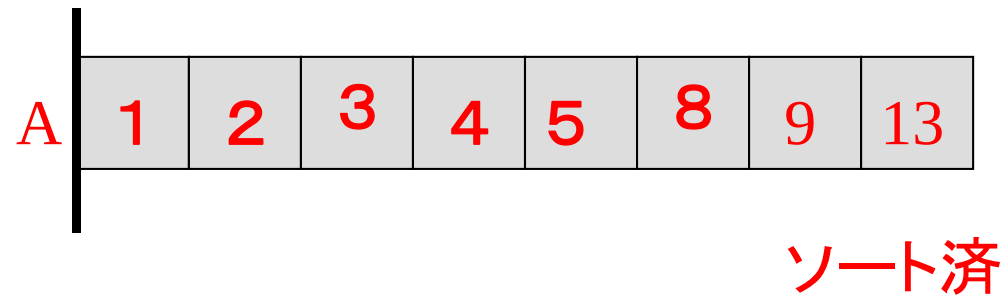
ソート済



# ヒープソート2の動き後半2



ちょっと省略



$\emptyset$

# 練習

次の配列を、単一配列ヒープソートを用いてソートするとき、配列の動きをシミュレートせよ。

|    |    |    |   |   |   |    |   |
|----|----|----|---|---|---|----|---|
| 11 | 25 | 21 | 1 | 8 | 3 | 16 | 5 |
|----|----|----|---|---|---|----|---|

# 単一配列ヒープの実現

```
1. void heap_sort()
2. {
3.     int i;      /* カウンタ */
4.     make_heap(); /* 空のヒープの作成 */
5.     /* ヒープ化 */
6.     for(i=0; i<n; i++) {
7.         insert_heap(A[i]);
8.     }
9.
10.    /* ヒープからのデータの取り出し */
11.    for(i=n-1; i>=0; i--) {
12.        A[i] = get_max();
13.    }
14.    return ;
15.}
```

仔細省略、  
ヒープの動作  
も変更する  
必要がある。

# 単一配列ヒープソートの計算量

## ヒープ化

操作insert\_heap(A) 1回あたり、時間量  $O(\log n)$

n回行うので、時間量  $O(n \log n)$

## 整列(最小値の削除の反復)

操作get\_max() 1回あたり、時間量  $O(\log n)$

n回行うので、時間量  $O(n \log n)$

---

ヒープ化、整列は、1回ずつ行われるので、

最悪時間量  $O(n \log n)$  のアルゴリズム

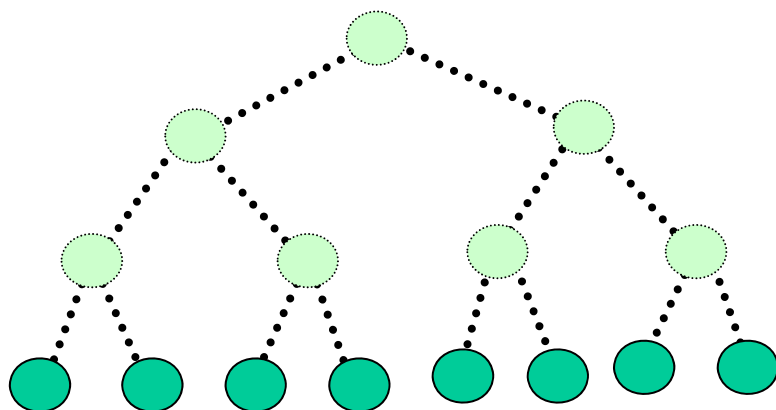
# ヒープ化の高速化

# ヒープ化の高速化におけるアイデア

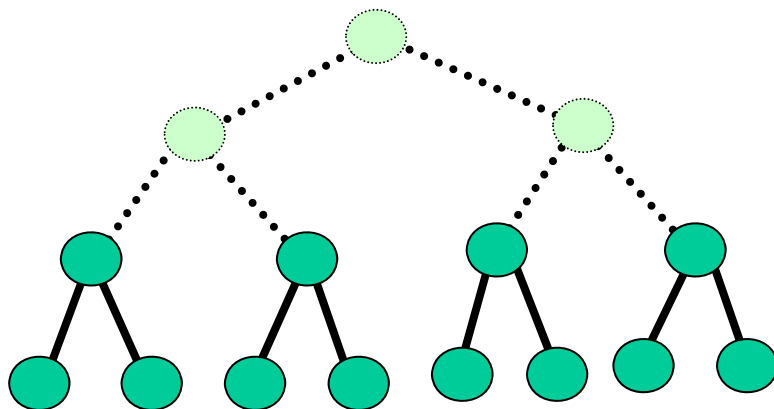
## 方針

- ヒープをボトムアップに生成する。
- 各接点では、2つの部分木を結合しながら、ダウンヒープで修正を繰り返す。

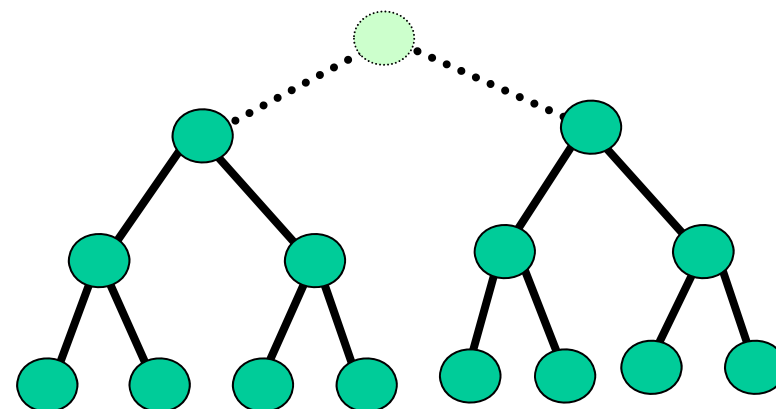
# イメージ



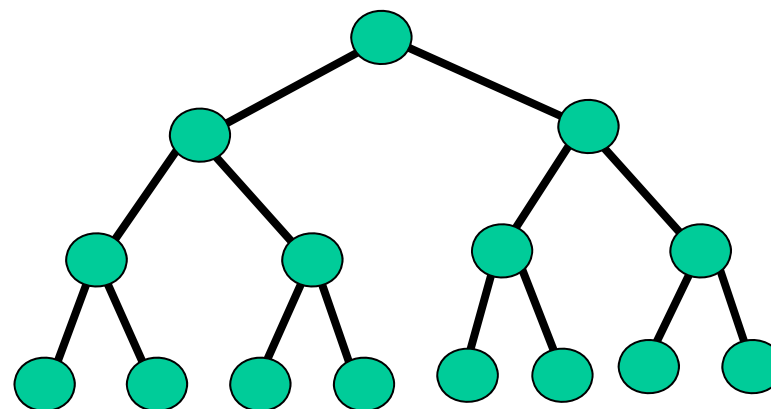
1点だけからなる 約 $n/2$ 個のヒープがあると考えるもよい。



3点のヒープが約 $n/4$ 個



7点のヒープが約 $n/8=2$ 個



$2^h - 1 = 15$  点のヒープが1個

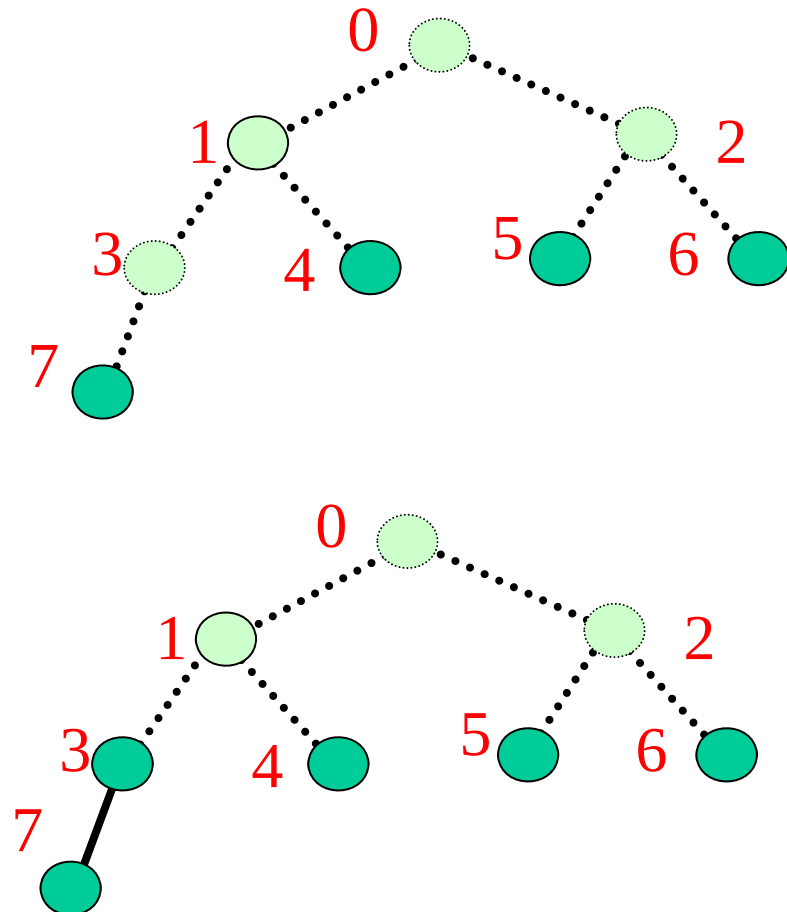


# 高速ヒープ化の動き

配列

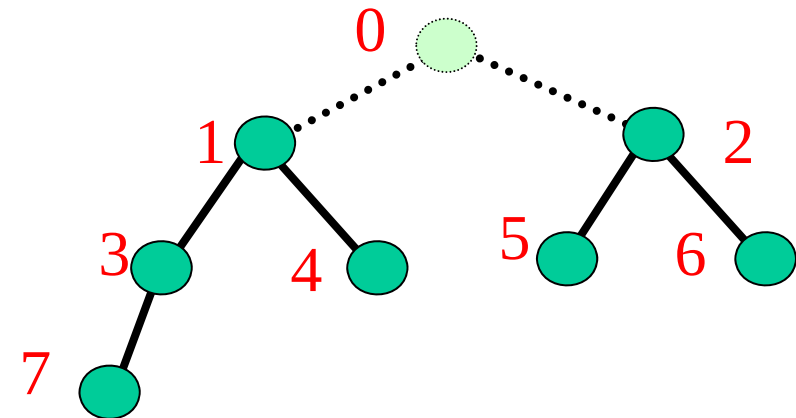
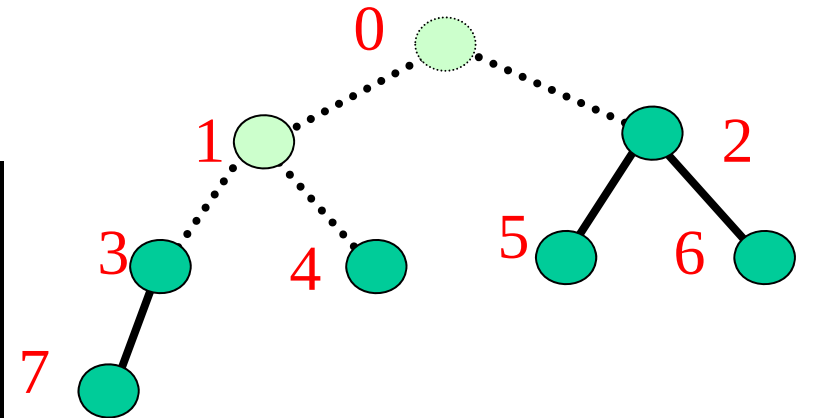
|   |   |   |   |   |    |   |   |
|---|---|---|---|---|----|---|---|
| 0 | 1 | 2 | 3 | 4 | 5  | 6 | 7 |
| 5 | 3 | 8 | 1 | 4 | 13 | 9 | 2 |

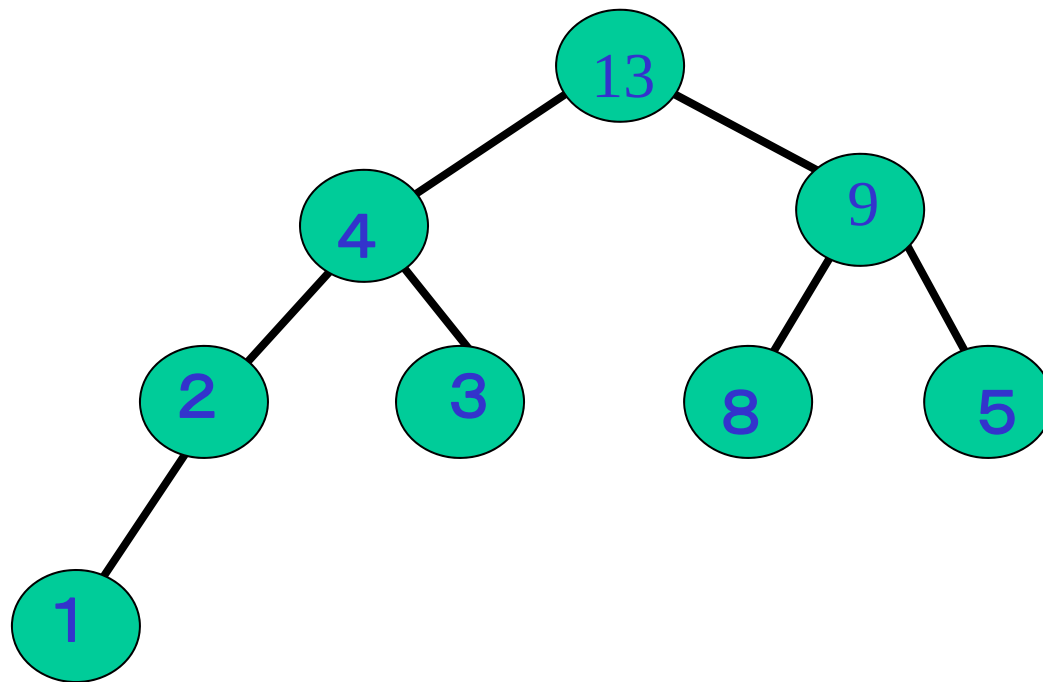
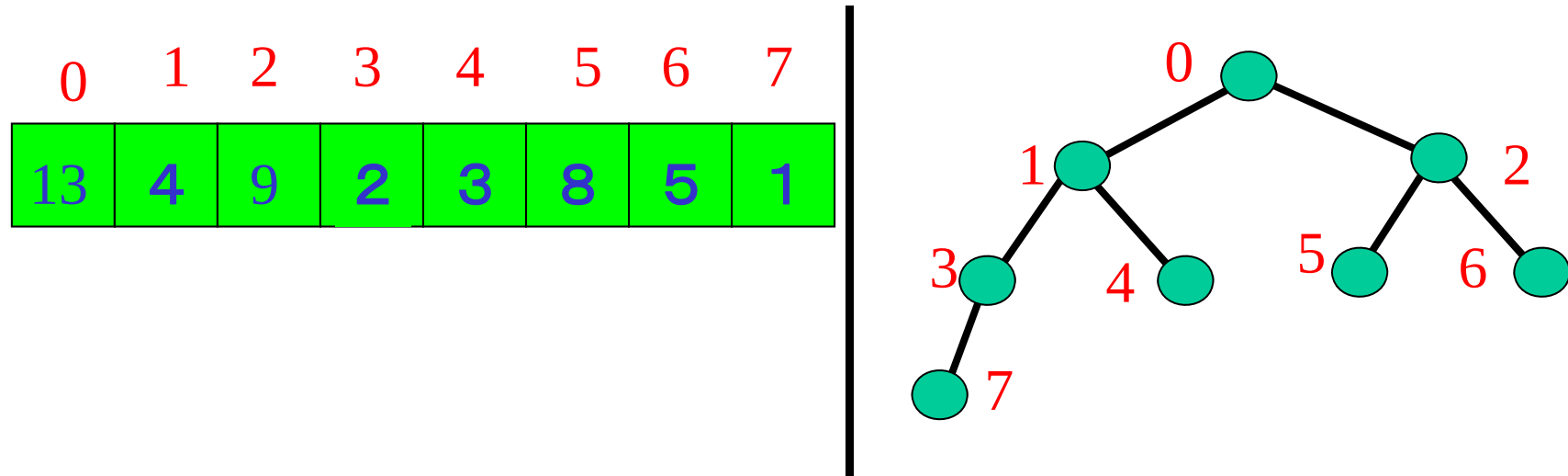
|   |   |   |   |   |    |   |   |
|---|---|---|---|---|----|---|---|
| 0 | 1 | 2 | 3 | 4 | 5  | 6 | 7 |
| 5 | 3 | 8 | 2 | 4 | 13 | 9 | 1 |



|   |   |    |   |   |   |   |   |
|---|---|----|---|---|---|---|---|
| 0 | 1 | 2  | 3 | 4 | 5 | 6 | 7 |
| 5 | 3 | 13 | 2 | 4 | 8 | 9 | 1 |

|   |   |    |   |   |   |   |   |
|---|---|----|---|---|---|---|---|
| 0 | 1 | 2  | 3 | 4 | 5 | 6 | 7 |
| 5 | 4 | 13 | 2 | 3 | 8 | 9 | 1 |





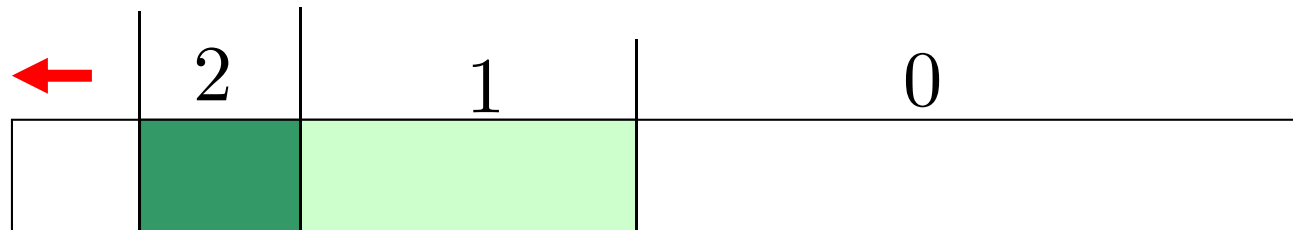
```
1. void heap_sort()
2. {
3.     int i;    /* カウンタ */
4.     make_heap(); /* 空のヒープの作成 */
5.     /* ヒープ化 */
6.     for(i=n/2;i>=0;i--){
7.         down_heap(A[i]);
8.     }
9.
10.    /*ヒープからのデータの取り出し*/
11.    for(i=n-1;i<=0;i--){
12.        A[i]=get_max();
13.    }
14.    return ;
15.}
```

## 命題HP4 (高速ヒープ化の最悪時間計算量)

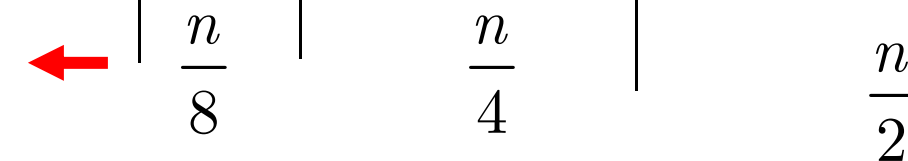
高速ヒープ化の最悪計算時間計算量は、 $O(n)$ である。

証明

交換回数



添え字範囲



このことより、ヒープ化に必要な最悪計算時間量を

$$T_h(n)$$

と書くと次式が成り立つ。

$$\begin{aligned} T_h(n) &\leq \frac{n}{4} \times 1 + \frac{n}{8} \times 2 + \frac{n}{16} \times 3 \dots \\ &= \frac{1}{2} \sum_{i=1}^{\log_2 n - 1} \frac{i}{2^i} \end{aligned}$$

$$2T_h(n) \leq \frac{n}{2} \times 1 + \frac{n}{4} \times 2 + \frac{n}{8} \times 3 \dots$$

$$\rightarrow T_h(n) \leq \frac{n}{4} \times 1 + \frac{n}{8} \times 2 + \frac{n}{16} \times 3 \dots$$


---

$$T_h(n) \leq \frac{n}{2} \times 1 + \frac{n}{4} \times 1 + \frac{n}{8} \times 1 \dots + 1$$

$$\leq n \frac{\frac{1}{2} \left( 1 - \frac{1}{2^{\log_2 n}} \right)}{1 - \frac{1}{2}} \leq n$$

# ヒープソートの計算量

ヒープ化の部分  $O(n)$

最大値発見と移動の部分

操作 `delete_max(A)` 1回あたり、時間量  $O(\log n)$   
n回行うので、時間量  $O(n \log n)$

---

結局

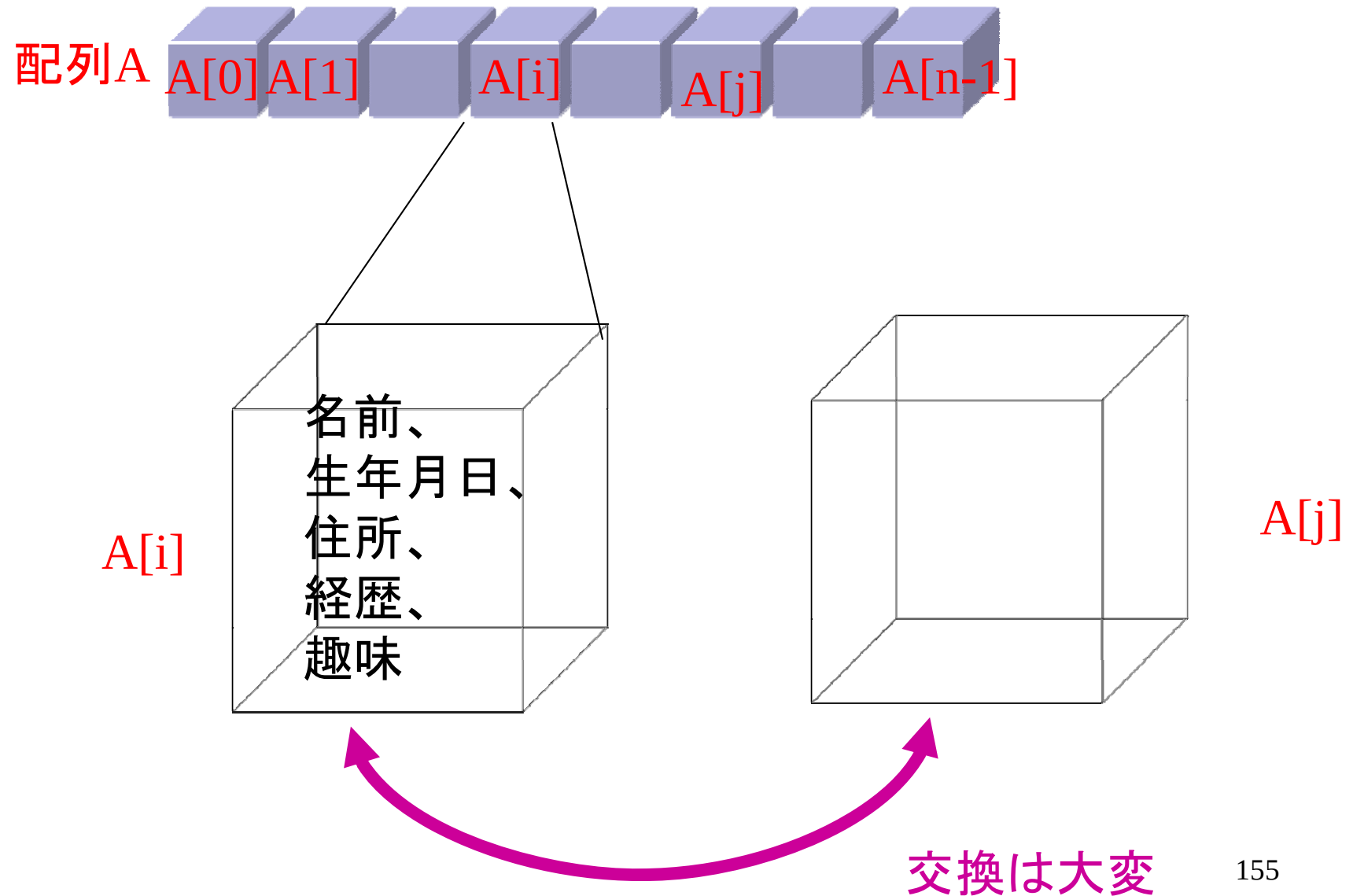
最悪時間計算量  $O(n \log n)$  のアルゴリズム



## 3-4: 比較によらないソート

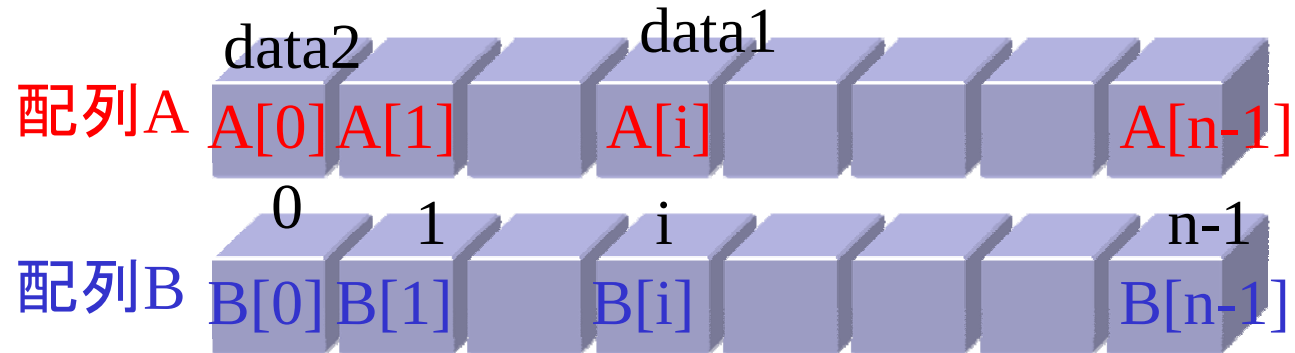
# 準備：容量の大きいデータの処理

ちょっと寄り道  
(一個一個が大きいデータ进行处理する工夫)

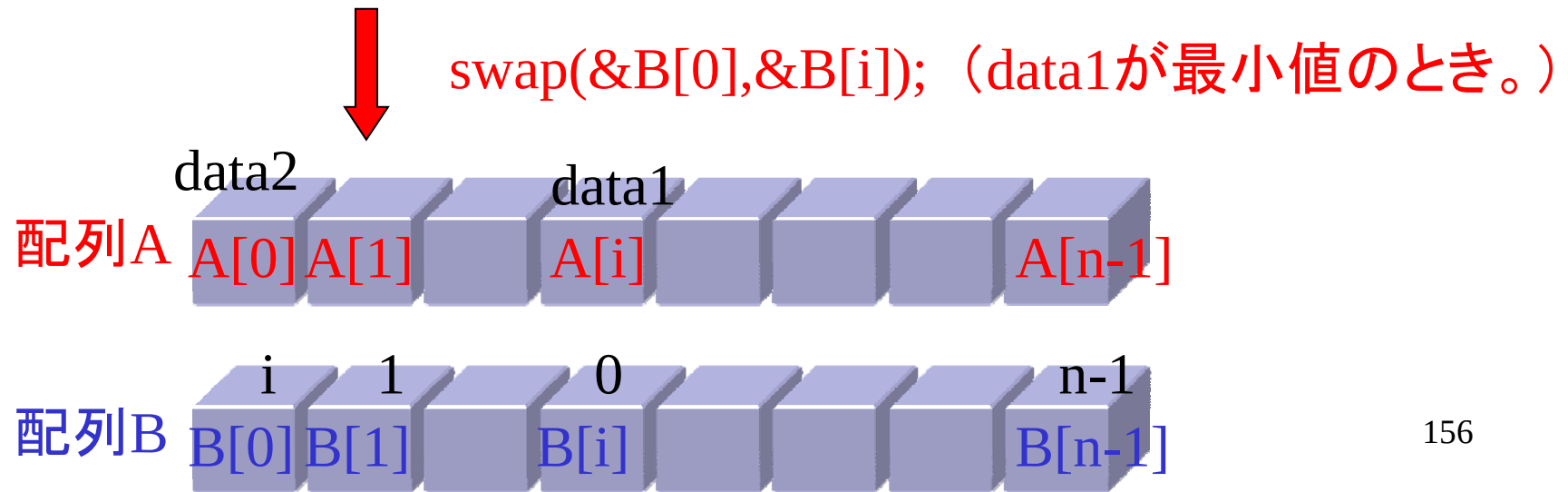


## 大きいデータを処理する工夫2

工夫: 大きいデータは動かさずに、  
たどり方だけがわかればいい。

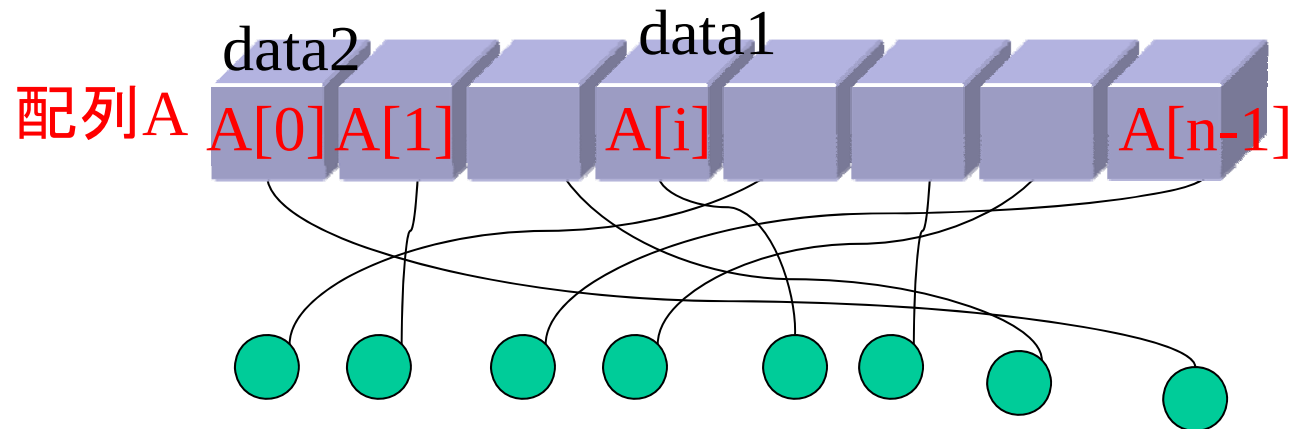
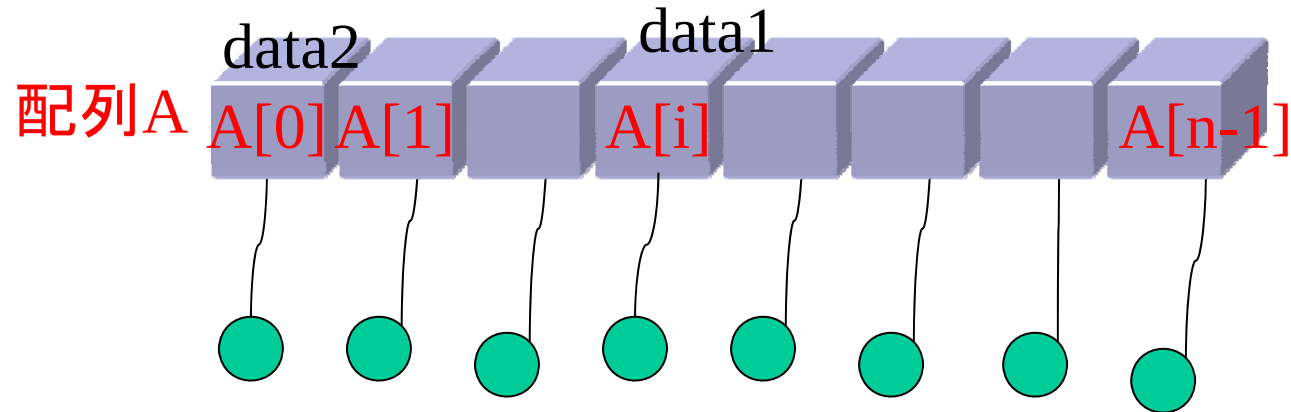


添字の配列Bだけを操作して、配列Aは動かさない。



## 大きいデータを処理する工夫3

### イメージ



ソート順は、下の情報からえられる。  
(配列の添字の利用だけでなく、ポインタでも同様のことができる。)

# 実現

細かい部分は省略します。

```
1. void selection_sort(struct data A[]){
2.     int B[N]; /*配列Aの要素を指す*/
3.     int i,j;   /*カウンタ*/
4.     for(i=0;i<N;i++){
5.         B[i]=i;
6.     }
7.     for(i=0;i<N;i++){
8.         min=i;
9.         for(j=i+1;j<N;j++){
10.            if(A[B[min]].item>A[B[j]].item){
11.                min=j;
12.            }
13.            swap(&B[i],&B[min]);
14.        }
15.    return;
16.}
```

# 比較によらないソート

## ● バケットソート

データが上限のある整数のような場合に用いる。  
データの種類が定数種類しかない場合には、  
関数で整数に写像してから用いてもよい。  
(ハッシュ関数)

## ● 基数ソート

大きい桁の数に対して、  
桁毎にバケットソートをしてソートする。

# バケットソート



# バケットソートの方針

とりあえず、簡単のために、  
データは、

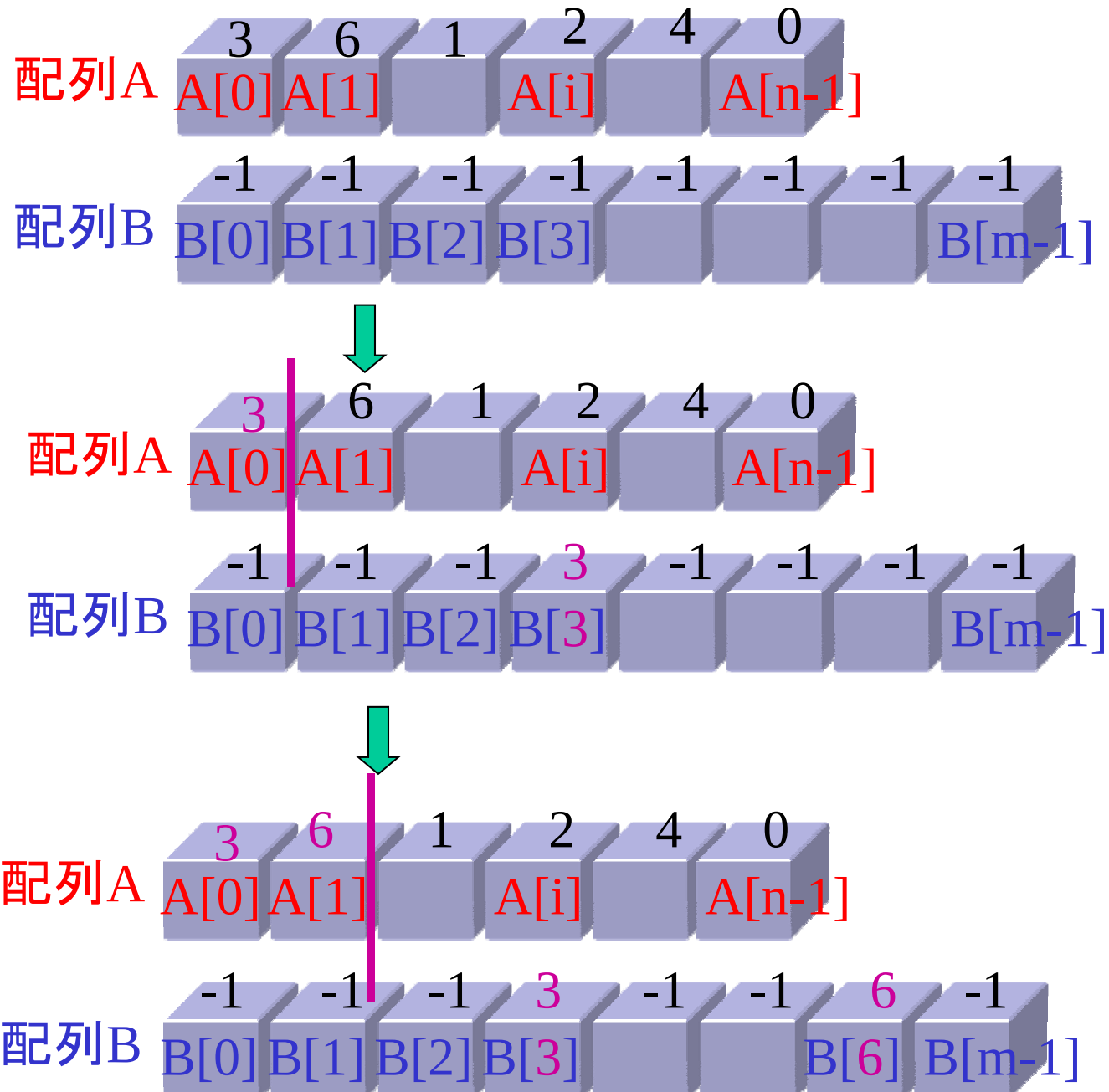
1. 重複がなく、
2. 0から $m-1$ の整数

という性質を満足としましょう。  
(例えば、学籍番号の下2桁とか。)

## 方針

- $m$ 個のバケット(配列)を用意して、データを振り分けていく。
- データそのものを配列の添字として使う。

# バケットソートの動き1

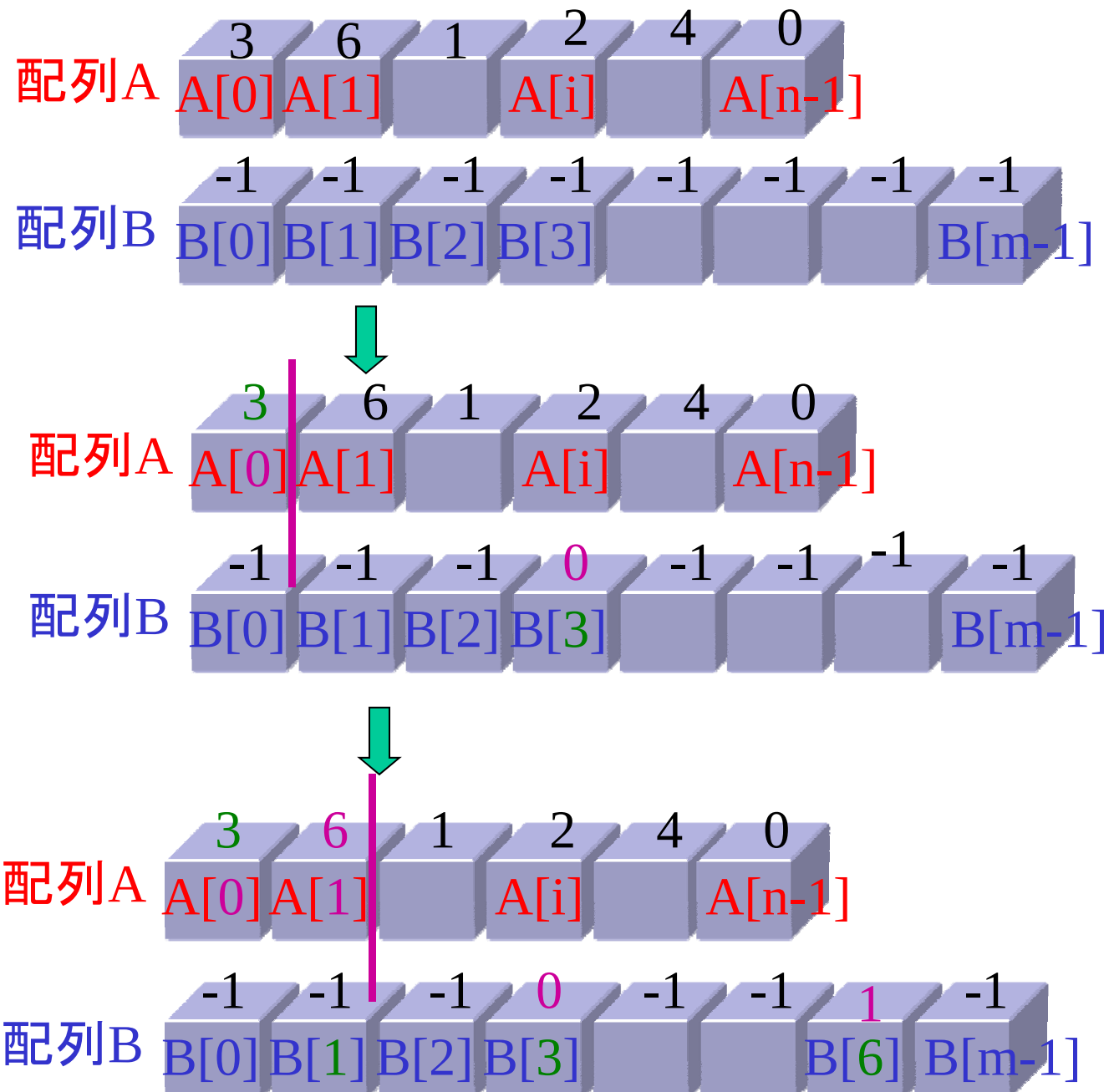


# バケットソートの実現

```
/* 概略です。細かい部分は省略
   入力データの配列の型がintであることに注意 */
void bucket_sort(int A[],int B[])
{
    int i;        /*カウンタ*/

    for(i=0;i<n;i++)
    {
        B[A[i]]=A[i];
    }
    return;
}
```

## バケットソートの動き2(添字を用いた場合)



# バケットソートの実現2

```
/* 配列の添字を用いて、間接的にソート*/  
void bucket_sort(int A[],int B[])  
{  
    int i;        /*カウンタ*/  
  
    for(i=0;i<n;i++)  
    {  
        B[A[i]]=i;  
    }  
    return;  
}
```

i番目のデータの参照は、A[B[i]]で行う。

# バケットソートの計算量

配列1回のアクセスには、定数時間で実行できる。

繰り返し回数は、明らかにデータ数 $n$ 回です。

また、

配列Bの準備や、走査のために、 $O(m)$  の時間量必要です。

最悪時間量  $O(m + n)$  のアルゴリズムです。

# 基数ソート

# 基数ソート

## 方針

- 大きい桁の数に対して、  
桁毎にバケットソートをしてソートする。
- 下位桁から順にバケットソートする。



# 基数ソートの動き(3桁)

|    |     |
|----|-----|
|    | A   |
| 0  | 221 |
| 1  | 650 |
| 2  | 23  |
| 3  | 2   |
| 4  | 106 |
| 5  | 226 |
| 6  | 250 |
| 7  | 126 |
| 8  | 372 |
| 9  | 47  |
| 10 | 215 |
| 11 | 33  |

0桁で  
ソート



|    |     |
|----|-----|
|    | A   |
| 0  | 650 |
| 1  | 250 |
| 2  | 221 |
| 3  | 2   |
| 4  | 372 |
| 5  | 23  |
| 6  | 33  |
| 7  | 215 |
| 8  | 106 |
| 9  | 226 |
| 10 | 126 |
| 11 | 47  |

1桁で  
ソート



|    |     |
|----|-----|
|    | A   |
| 0  | 2   |
| 1  | 106 |
| 2  | 215 |
| 3  | 221 |
| 4  | 23  |
| 5  | 226 |
| 6  | 126 |
| 7  | 33  |
| 8  | 47  |
| 9  | 650 |
| 10 | 250 |
| 11 | 372 |

2桁で  
ソート



|    |     |
|----|-----|
|    | A   |
| 0  | 2   |
| 1  | 23  |
| 2  | 33  |
| 3  | 47  |
| 4  | 106 |
| 5  | 126 |
| 6  | 215 |
| 7  | 221 |
| 8  | 226 |
| 9  | 250 |
| 10 | 372 |
| 11 | 650 |

# 練習

次の配列に基数ソートを適用したときの動作を示せ。

A

|    |     |
|----|-----|
| 0  | 123 |
| 1  | 32  |
| 2  | 612 |
| 3  | 4   |
| 4  | 821 |
| 5  | 621 |
| 6  | 100 |
| 7  | 754 |
| 8  | 253 |
| 9  | 558 |
| 10 | 56  |
| 11 | 234 |

# 基数ソートの実現

```
/*細かい実現は省略*/  
void radix_sort(int A[])  
{  
    int i,j;        /*カウンタ*/  
  
    for(k=0;k<max_k;k++)  
    {  
        bucket_sort(A,k);  
        /*第k桁を基にして、バケットソートでソートして、  
        もとの配列に戻すように拡張する。*/  
    }  
  
    return;  
}
```

# 基数ソートの計算量

バケットソートを桁数分行えばよいので、  
k桁数を基数ソートするには、

最悪時間量  $O(k(m+n))$  のアルゴリズムです。

また、

$N$  種類のデータを区別するには、 $k = \log_m N$  桁必要です。

$N \simeq n$  のときには、結局

$O(m \log n + n \log n)$  の時間量を持つアルゴリズムです。

だから、バケットソートや基数ソートは、  
データ範囲 $m$ や、桁数 $k$ に注意しましょう。

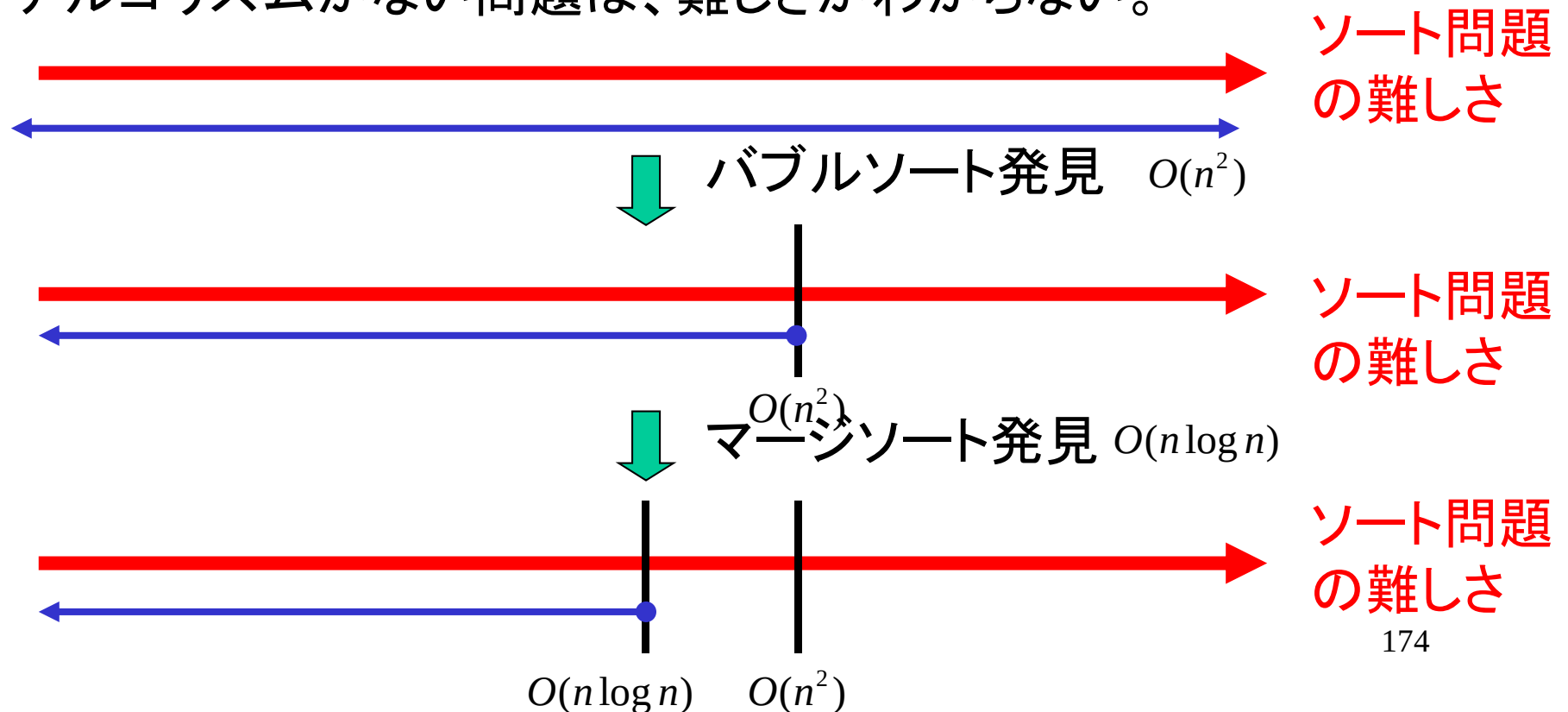
## 3-5:ソート問題の下界

# 問題とアルゴリズム

具体的なアルゴリズムを作ることは、  
問題の難しさ(問題固有の計算量)の上界を与えています。

## 最適なアルゴリズムの存在範囲

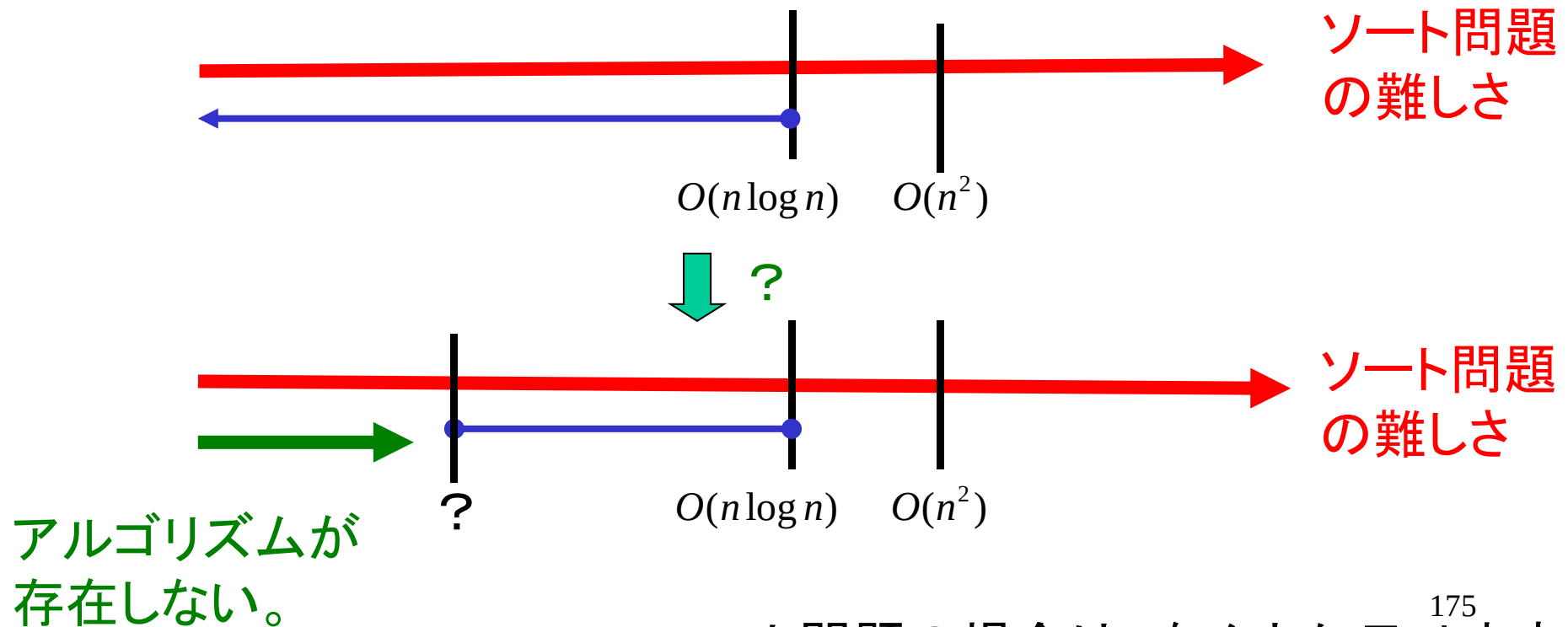
アルゴリズムがない問題は、難しさがわからない。



# 問題と下界

一方、問題の難しさの範囲を下の方から狭めるには、問題を解くアルゴリズムが無いことを示さないといけない。実は、1つのアルゴリズムを作ることより、アルゴリズムが存在しないことを示すことの方が難しい。

最適なアルゴリズムの存在範囲



ソート問題の場合は、なんとか示せます。

# アルゴリズムと決定木

## (比較によるソートの下界証明の準備)

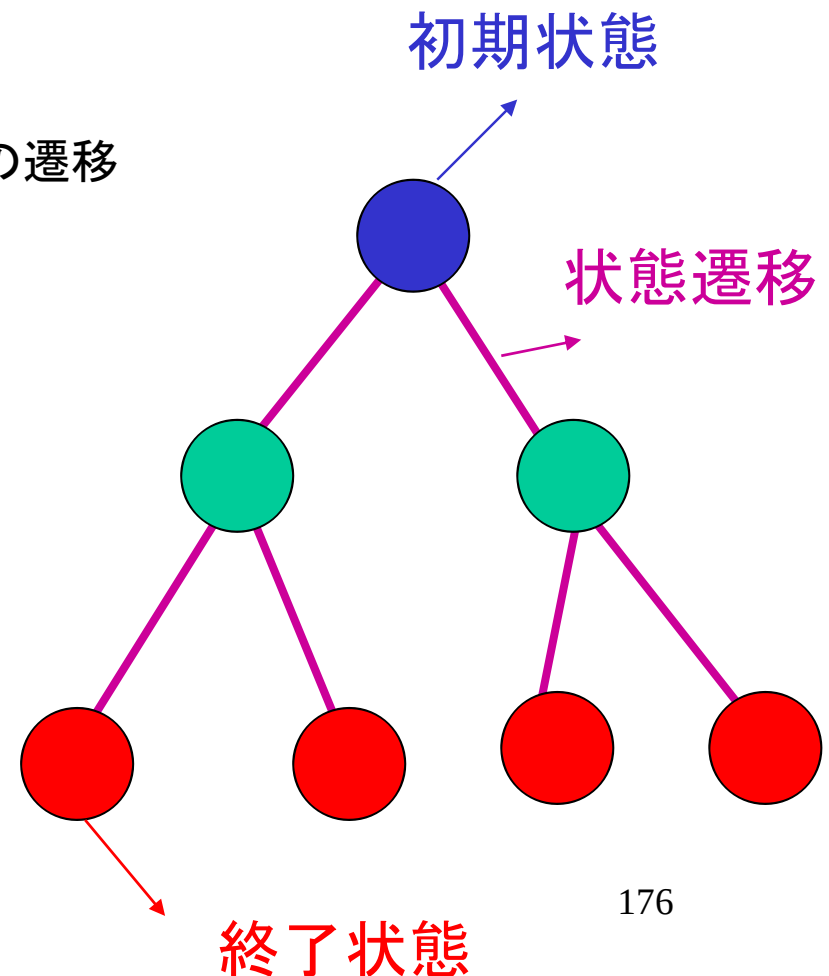
決定木の根: 初期状態

決定木の節点: それまでの計算の状態

決定木の枝 : アルゴリズムの進行による状態の遷移

決定木の葉: 終了状態

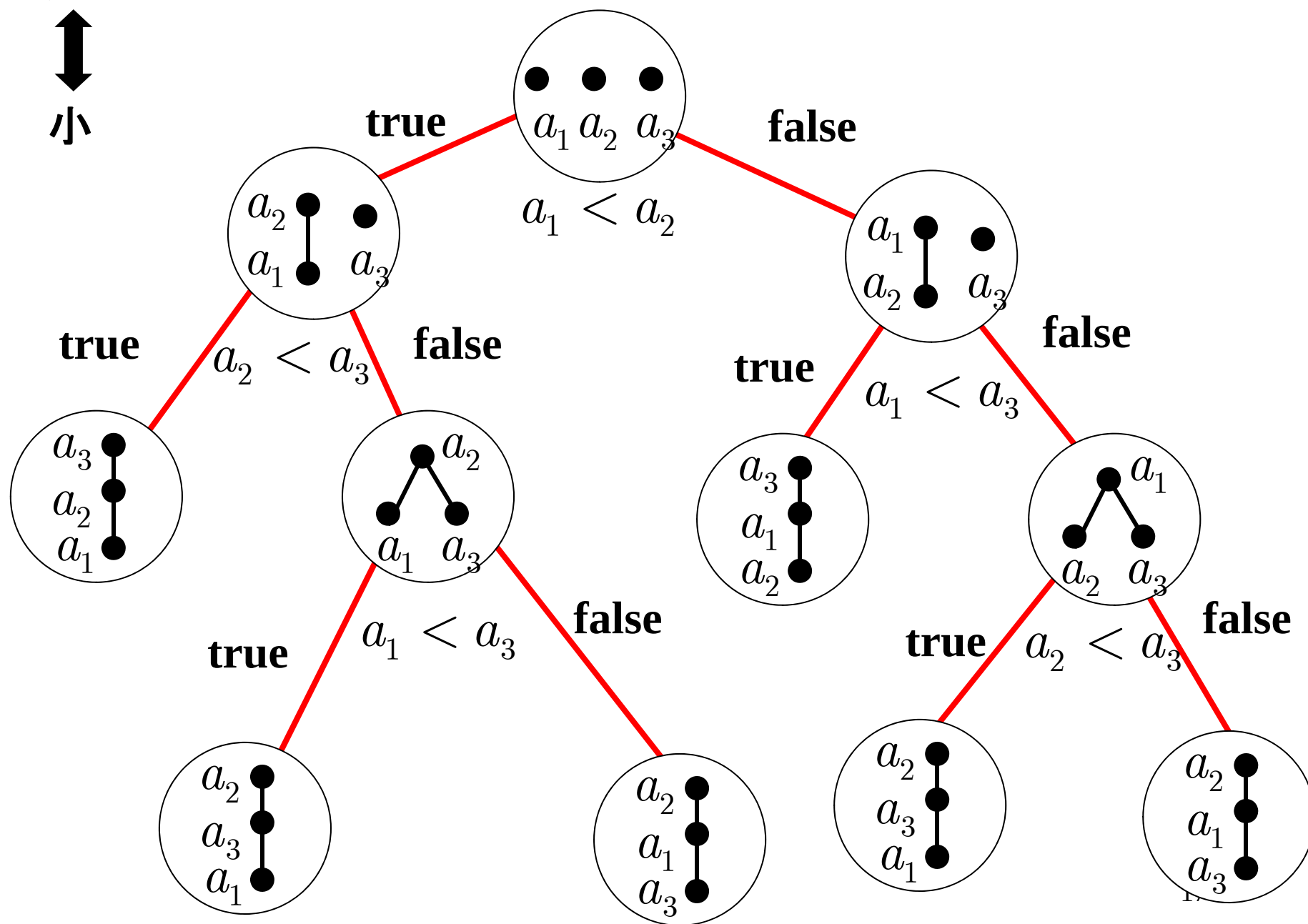
(ヒープのような)データ構造の木ではなくて、概念的、抽象的なもの。  
根がアルゴリズムの初期状態に対応し、  
葉がアルゴリズム終了状態に対応し、  
根からの道がアルゴリズムの実行順に対応し、  
根から葉までの道の長さが時間量に対応する。





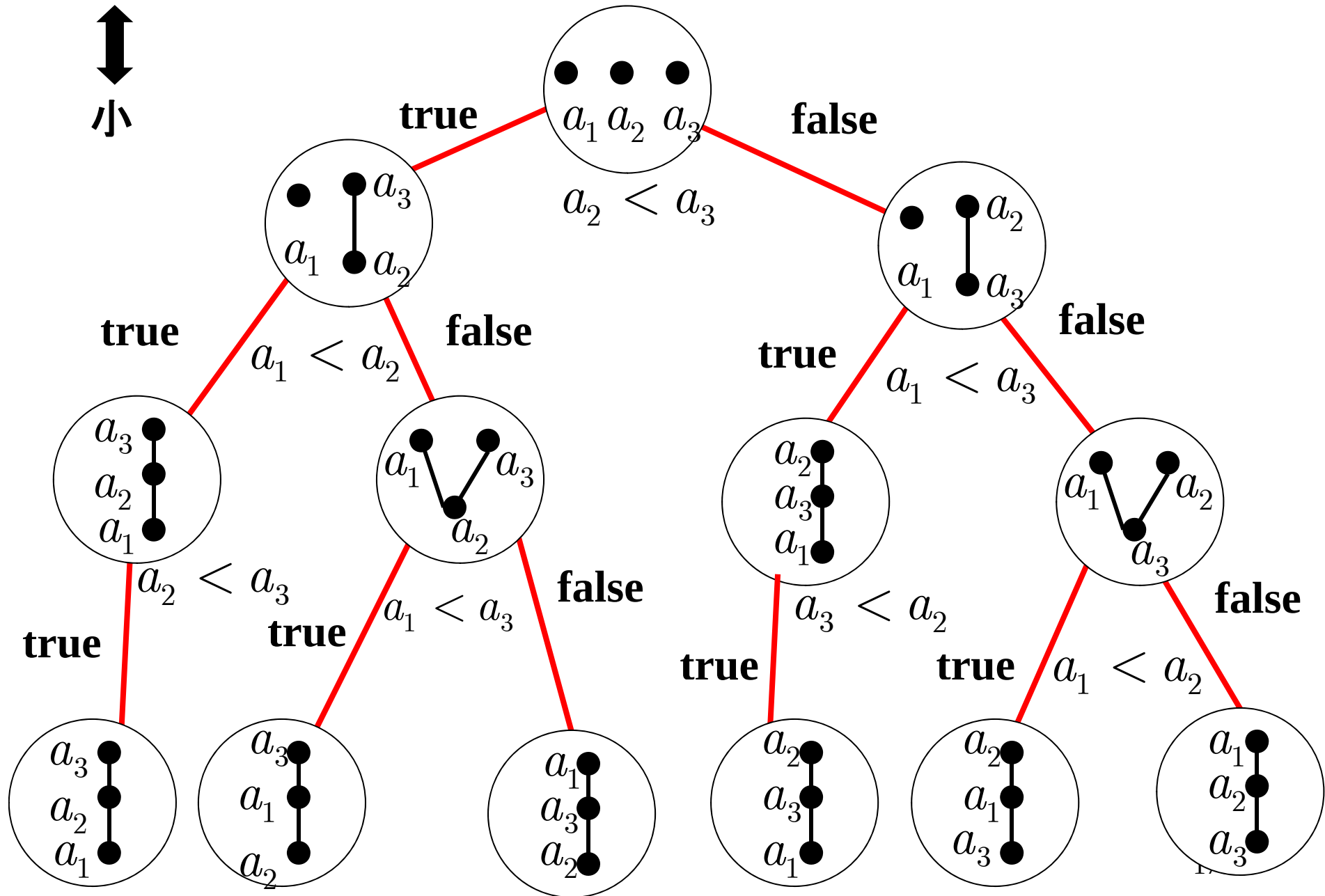
# 決定木の例 (挿入ソート)

大  
↑  
↓  
小



# 決定木の例(バブルソート)

大  
↑  
↓  
小



# 練習

(1) 3要素の選択ソートのアルゴリズムに対応する決定木を作成せよ。

(2) 4要素の決定木を作成せよ。  
(どんなアルゴリズムを用いても良い。)

# ソート問題の下界

どんな入力でもきちんとソートするには、  
決定木に $n!$ 個以上の葉がなければならない。  
それで、アルゴリズムの比較回数は、  
決定木の高さで定まる。

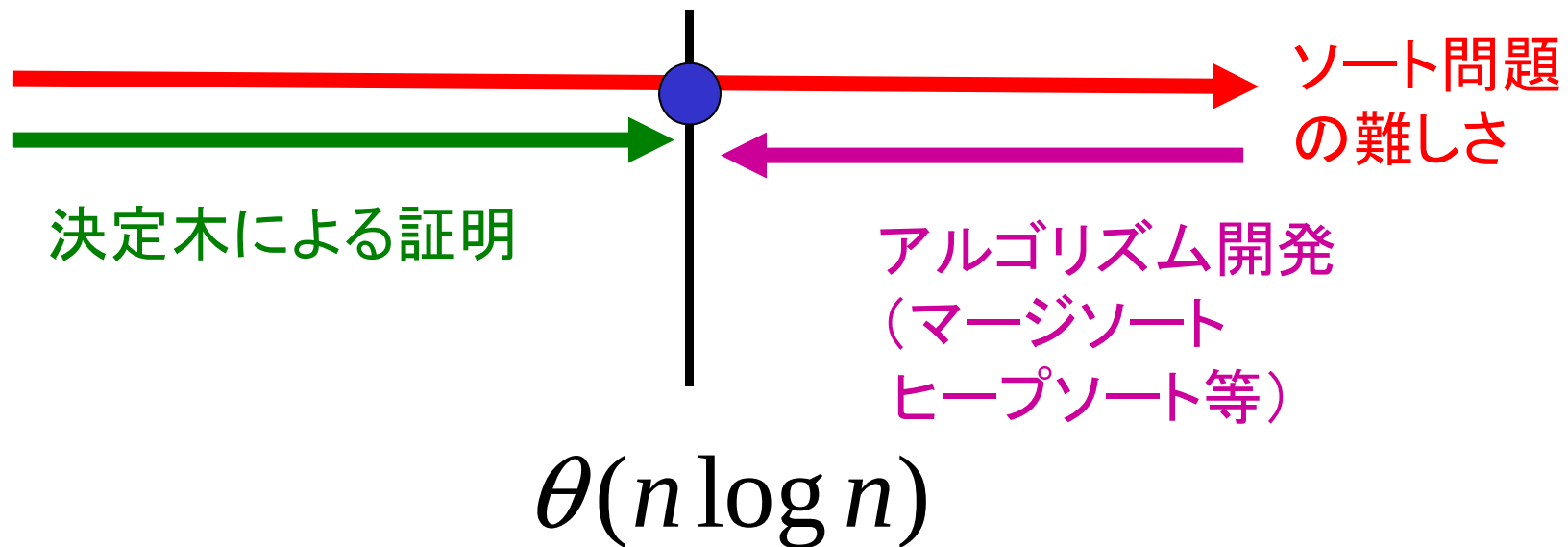
最悪時間が良いアルゴリズムは高さが低く、  
悪いアルゴリズムは高さが高い。

高さが $h$ の決定木では、高々 $2^h$ 個の葉しかない。

よって、 $h \geq \log_2 n! \geq n \log n$

よって、ソートアルゴリズムでは少なくとも  
 $\Omega(n \log n)$  の時間量が必要である。

# ソート問題の難しさ



こんなことを考えるのが、  
計算量理論の分野です。