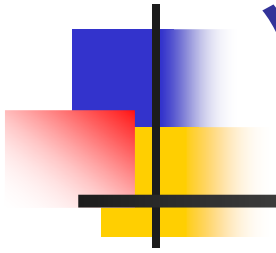


ソート(並べ替え)アルゴリズム

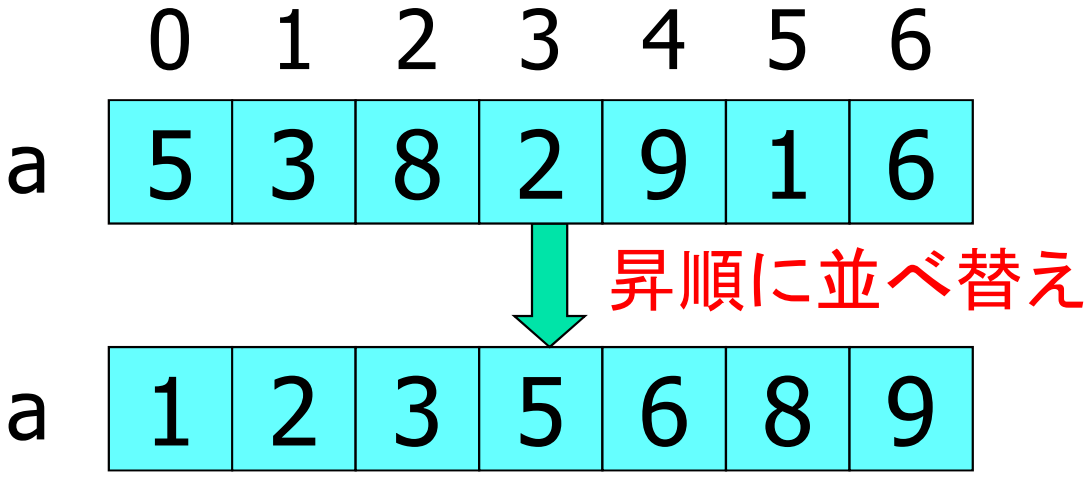
C言語による実装

埼玉大学工学部電気電子システム工学科
伊藤 和人

並べ替えとは

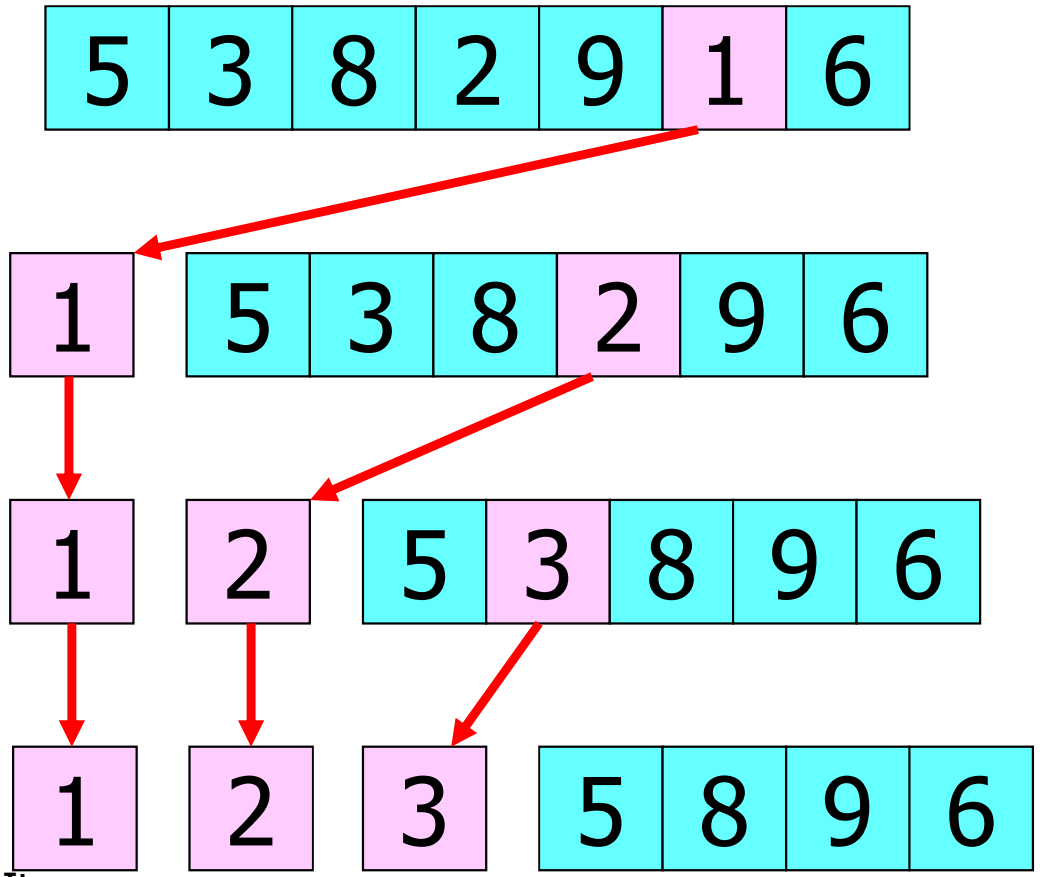
- 並べ替え=ソート(sort)
- 複数個のデータを大きい順(降順)あるいは小さい順(昇順)に整列する
- 以降では昇順の並べ替えのみ考慮

例　int a[7]



基本ソートアルゴリズム

- アイデア:
  先頭に最も小さな値、残りのデータのうち
  最も小さな値が2番目、以下同様

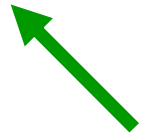


基本ソートのプログラム

例

```
void sort( void )
{
   int i, j;
   for( i=0 ; i<N-1 ; i++ )
      for( j=i+1 ; j<N ; j++ )
         if( a[i] > a[j] ) swap( i, j );
}
void swap( int i, int j )
{
    int t = a[i]; a[i] = a[j]; a[j] = t;
}
```

要素数Nの配列aを
小さい順にソート



配列aの実際の型に合わせて変更する

基本ソートプログラム実行時間

- 2重ループの内側の実行回数

$$\sum_{i=0}^{N-2}\sum_{j=i+1}^{N-1}1=\sum_{i=0}^{N-2}(N-1-i)$$

$$=(N-1)^2-\frac{(N-1)(N-2)}{2}$$

$$=\frac{N(N-1)}{2}$$

比較実行回数 $=\dfrac{N(N-1)}{2}$

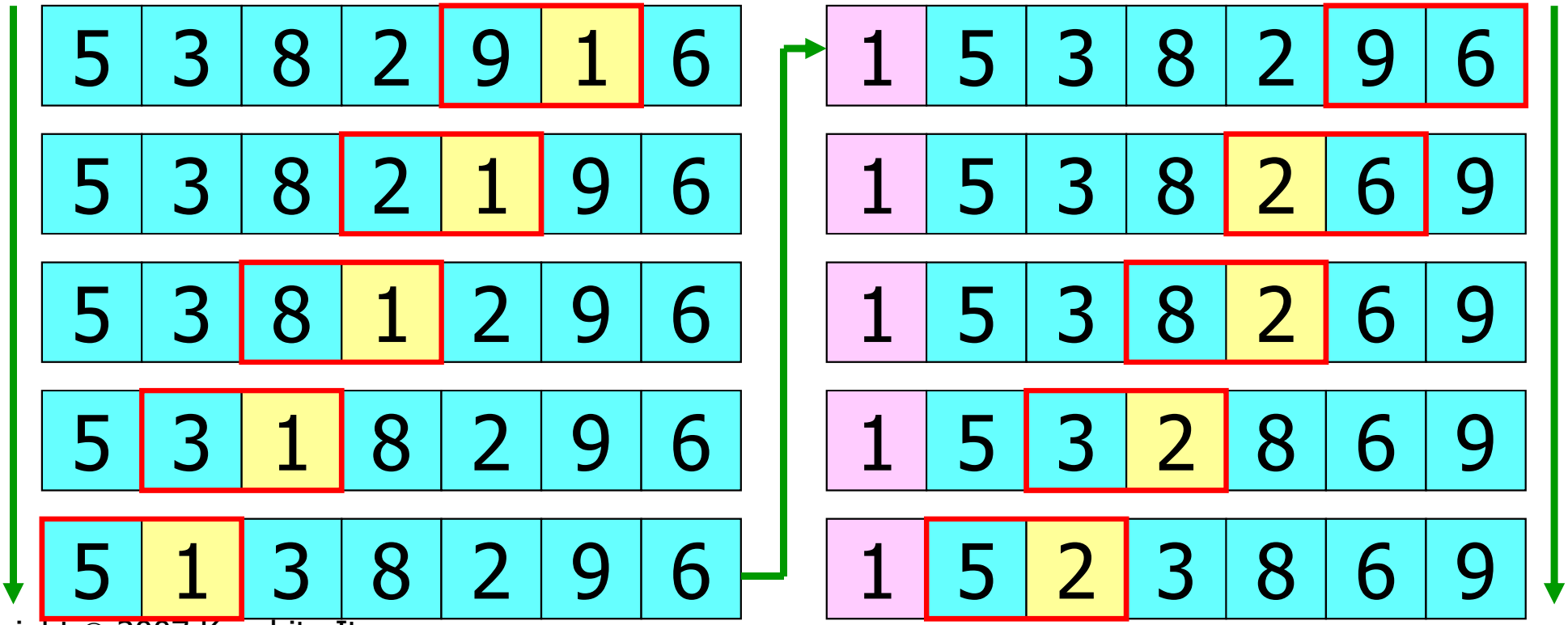
交換実行回数 $=\dfrac{N(N-1)}{4}$

実行時間は $O(N^2)$

バブルソート

泡(バブル)が立ち上るように
並べ替えが進行

- アイデア:
  小さいデータは前、大きいデータは後ろ
  ⇒末尾から順に調べ、隣り合うデータの
  大小が逆転していれば交換

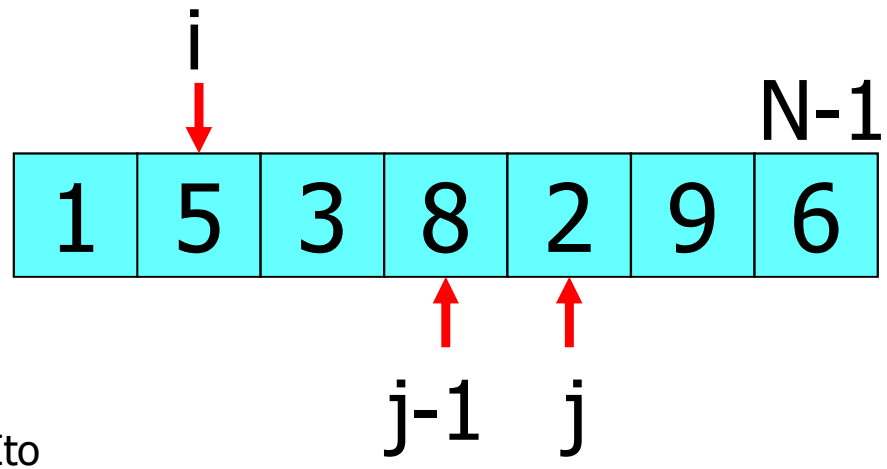


バブルソートのプログラム

例

```
void sort( void )
{
   int i, j;
   for( i=0 ; i<N-1 ; i++ )
      for( j=N-1 ; j>i ; j-- )
         if( a[j-1] > a[j] )
            swap( j-1, j );
}
```

要素数Nの配列aを
小さい順にソート



バブルソート実行時間

- 2重ループの内側の実行回数

$$\sum_{i=0}^{N-2}\sum_{j=i+1}^{N-1}1=\frac{N(N-1)}{2}$$

比較実行回数 $=\frac{N(N-1)}{2}$

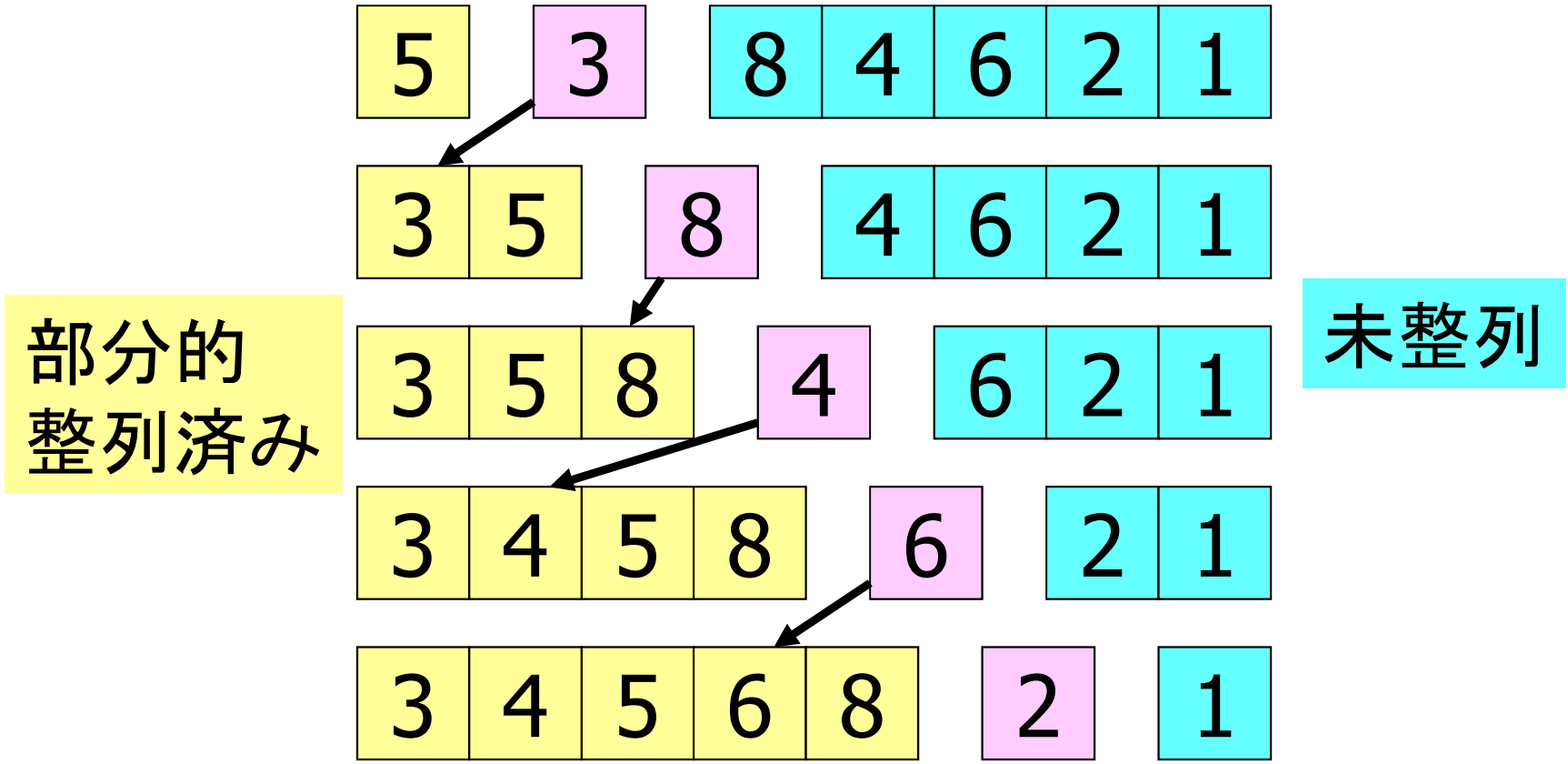
交換実行回数 $=\frac{N(N-1)}{4}$

実行時間は
$O(N^2)$

基本アルゴリズムと全く同一の実行時間

挿入ソートアルゴリズム

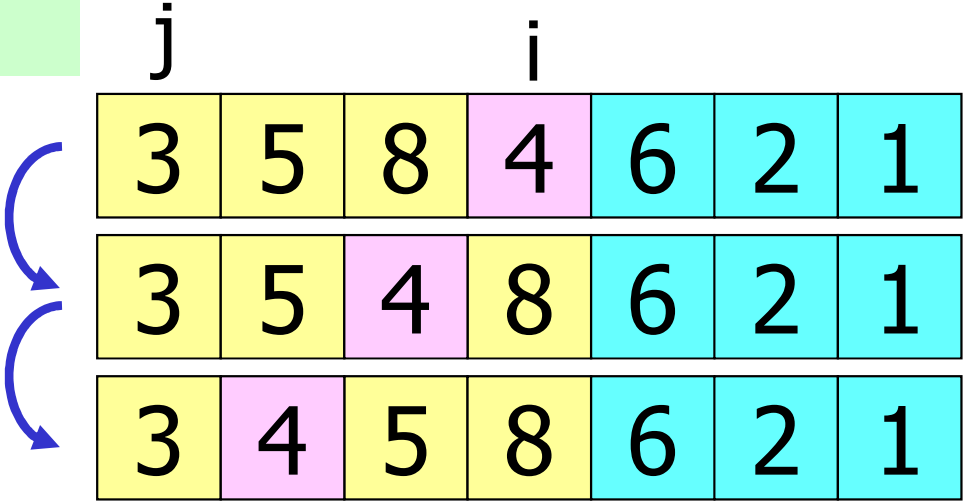
- アイデア:
  部分的に整列済みのデータの適切な位置に、新たなデータを挿入する



挿入ソートのプログラム

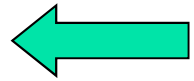
例
```
void sort( void )
{
    int i, j, k;
    for( i=1 ; i<N ; i++ ){
        for( j=i-1 ; j>=0 ; j-- )
            if( a[j] < a[i] ) break;
        for( k=i ; k>j+1 ; k-- )
            swap( k, k-1 );
    }
}
```

要素数Nの配列aを
小さい順にソート



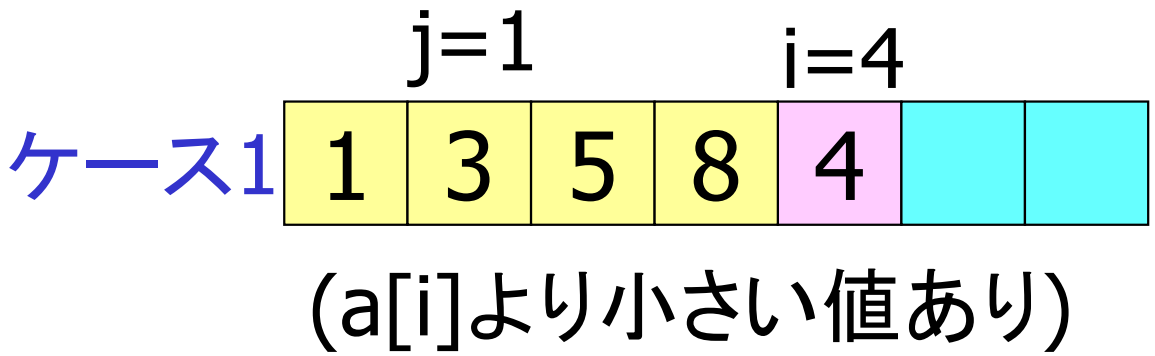
挿入ソート: 挿入位置を調べる

```
for( j=i-1 ; j>=0 ; j-- )
    if( a[j] < a[i] ) break;
for( k=i ; k>j+1 ; k-- )
    swap( k, k-1 );
```

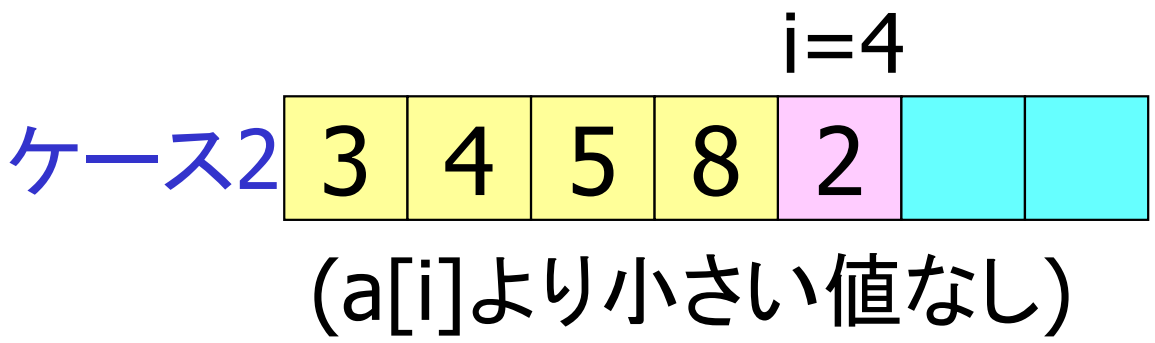


このfor文に注目

位置iより左にあり、
a[i]より小さい値を探す



if文の条件が成立して
for文終了(このときj=1)



for文の終了条件により
終了(このときj=-1)

適切な位置へ挿入

```
for( j=i-1 ; j>=0 ; j-- )
    if( a[j] < a[i] ) break;
for( k=i ; k>j+1 ; k-- )
    swap( k, k-1 );
```

挿入処理

ケース1(j=1)

i=4

1	3	5	8	4		

実際に行う挿入処理(k>2)

```
swap( 4, 3 );
swap( 3, 2 );
```

順に実行

ケース2(j=-1)

i=4

3	4	5	8	2		

実際に行う挿入処理(k>0)

```
swap( 4, 3 );
swap( 3, 2 );
swap( 2, 1 );
swap( 1, 0 );
```

順に実行

挿入ソートプログラム実行時間

- 比較: 2重ループの内側の最多実行回数

$$\sum_{i=1}^{N-1}\sum_{j=0}^{i-1} 1 = \sum_{i=1}^{N-1} i = \frac{N(N-1)}{2}$$

- 交換: 2重ループの内側の最多実行回数

$$\sum_{i=1}^{N-1}\sum_{k=1}^{i} 1 = \sum_{i=1}^{N-1} i = \frac{N(N-1)}{2}$$

平均比較実行回数 $= \frac{N(N-1)}{4}$

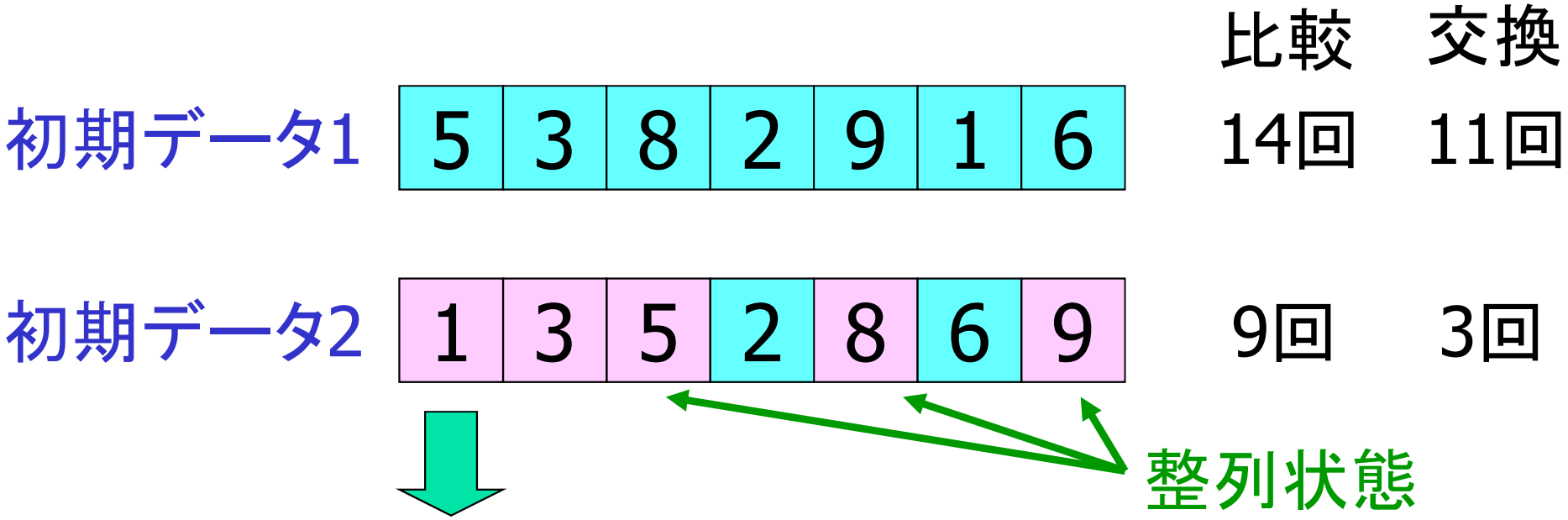
平均交換実行回数 $= \frac{N(N-1)}{4}$

実行時間は
$O(N^2)$

他のアルゴリズム
より少ない

挿入ソートの特徴

- 元のデータがほとんど整列していると、比較の回数および交換回数が減少



少ない手数で大雑把な整列を行い
その後に厳密な挿入ソートを行う

概略挿入ソート

例

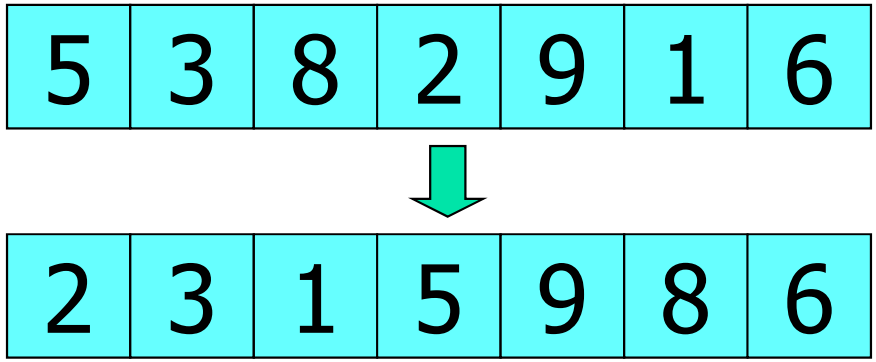
```
void sort_1( void )
{
   int i, j, k;
   for( i=w ; i<N ; i++ ){
      for( j=i-w ; j>=0 ; j-=w )
         if( a[j] < a[i] ) break;
      for( k=i ; k>j+w ; k -=w )
         swap( k, k-w );
   }
}
```

挿入ソートに間隔wを導入

概略+厳密挿入ソート

- w=3として概略挿入ソート実行

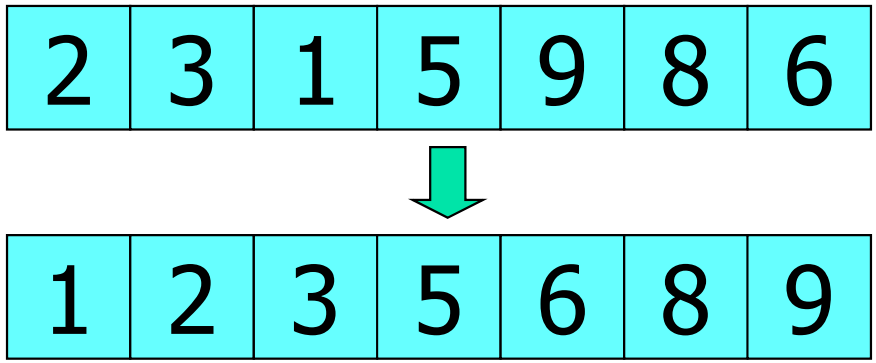
初期データ



比較　交換

4回　2回

- w=1として概略ソート、つまり厳密ソート実行



比較　交換

10回　5回

挿入ソートを直接行う場合に比べて交換回数が低減

この手法をシェルソートという

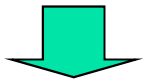
シェルソートプログラム実行時間

- 間隔wの選び方

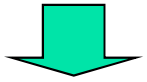
系列: 1, 4, 13, 40, 121, ...　⇒ $w_k=3w_{k-1}+1$

データ数Nより小さい最大のw_kを定め、
順次wの値を小さなものに入れ替えながら
繰り返し概略挿入ソートを実行

ほとんど整列済みのデータに
挿入ソートを1回実行すると　$O(N)$



wの値を変えながら挿入ソートを繰り返し



wの値に上の系列を使った場合　$O(N^{1.25})$

シェルソートプログラム

```
void sort( void )
{
   int i, j, k, w;
   for( w=1 ; w<N ; w=3*w+1 );
   for( w /= 3 ; w>0 ; w /= 3 )
      for( i=w ; i<N ; i++ ){
         for( j=i-w ; j>=0 ; j-=w )
            if( a[j] < a[i] ) break;
         for( k=i ; k>j+w ; k -=w )
            swap( k, k-w );
      }
}
```

次のw/=3と
セットでwの
初期値を求める

wの系列を逆に
たどりながら
概略挿入ソートを
繰り返す

ソート実行時間の理論限界

- データ数nに対して最短ソート時間の下限
  $O(n\log n)$ であることが知られている
- バブルソート、挿入ソート: $O(n^2)$
  シェルソート: $O(n^{1.25})$

理論限界に達していない

理論限界を達成するソートアルゴリズム

計算量を減らす工夫

- 基本的なアイデア

データ数nに対して$O(n^2)$の処理

kn^2

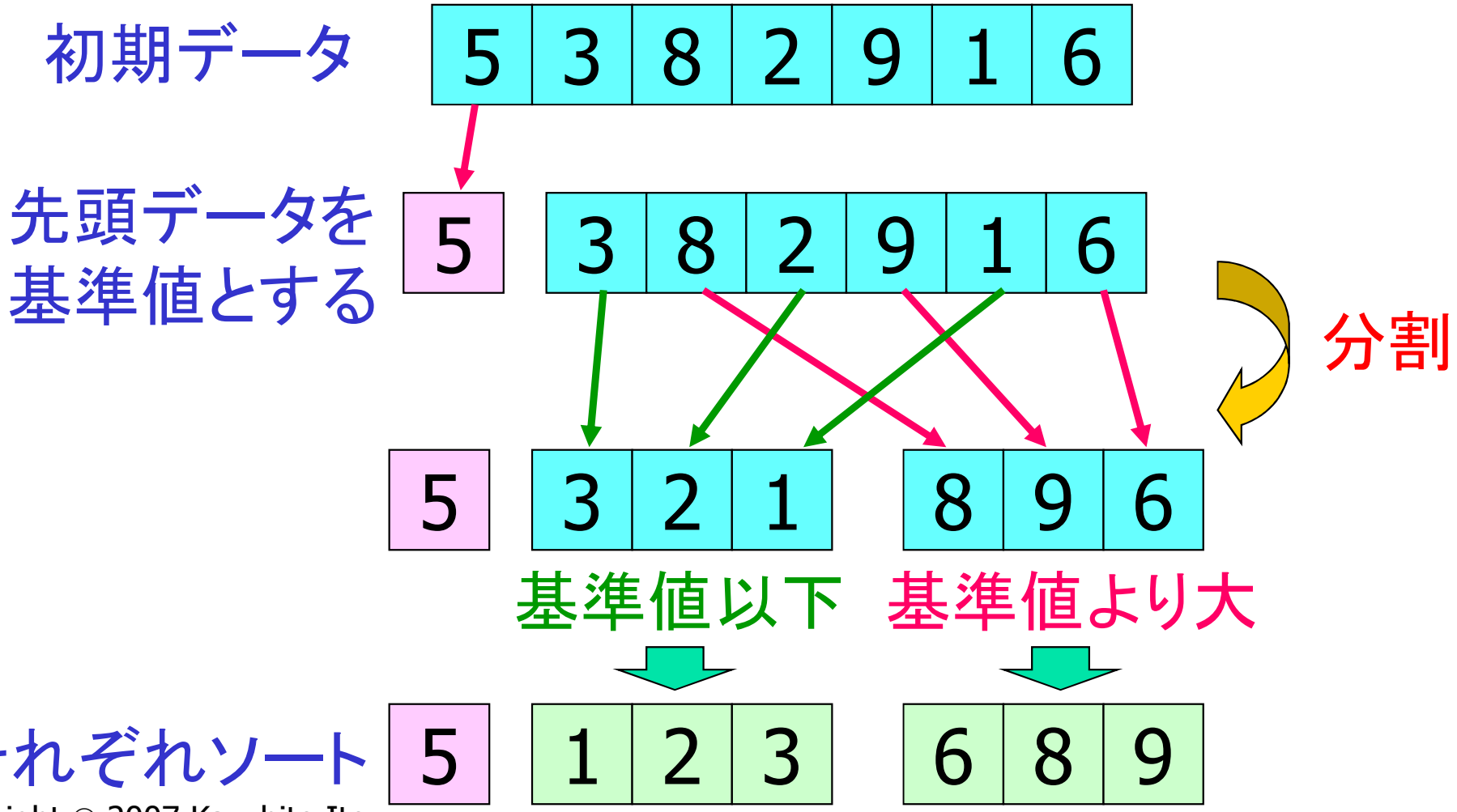
データを$n/2$個と$n/2$個に分割し、
それぞれに対して$O((n/2)^2)$の処理

$$k\frac{n}{4}^2 + k\frac{n}{4}^2 = k\left(\frac{n^2}{2}\right)$$

ソートへの応用は?

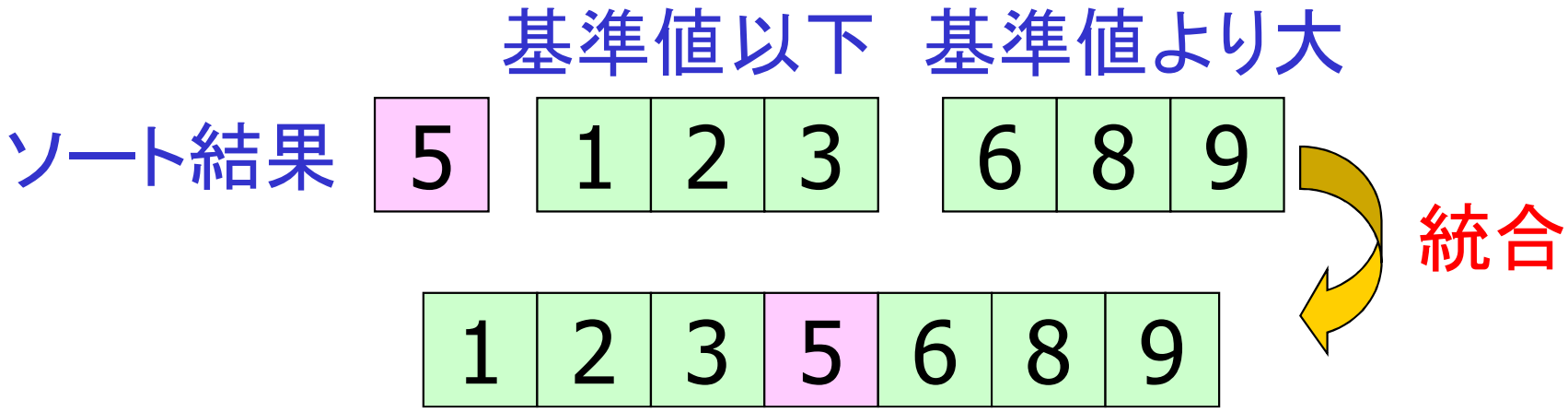
計算量を減らすソート

- 基準値を選び、基準値より小さいデータと基準値より大きいデータに分けてソート



計算量を減らすソート(その2)

- それぞれのソート結果を統合する



- 基準値以下のデータと基準値より大きいデータにうまく分割できると高速化
- 分割したデータをさらに分割してソート
  ⇒　さらに高速化

このアルゴリズムをクイックソートという

再帰的なクイックソート

```
void quick_sort( int a[], int num )
{
    int la[N-1], ha[N-1];
    int i, j, lnum=0, hnum=0;
    int mid = a[0];
    for( i=1 ; i<num ; i++ )
        if( a[i] <= mid ) la[lnum++] = a[i];
        else              ha[hnum++] = a[i];
    if( lnum > 0 ) quick_sort( la, lnum );
    if( hnum > 0 ) quick_sort( ha, hnum );
    for( i=0 ; i<lnum ; i++ ) a[i] = la[i];
    a[i++] = mid;
    for( j=0 ; j<hnum ; j++, i++ ) a[i] = ha[j];
}
```

小さい(lower)データと
大きい(higher)データ

分割

それぞれ
ソート

統合

クイックソートのメモリ節約

- 前ページのプログラムでは、quick_sortの呼出しごとに要素数N-1の配列を2個作成
- しかし、全体では半分の要素しか必要ない
  ⇒ メモリの無駄
- 元データの配列をうまく利用する

メモリを節約するデータ分割

メモリを節約したクイックソート

```
void quick_sort( int a[], int low, int high )
{
   int mid=a[low], i=low+1, j=high;
   for( ; ; ){
      while( i<j && a[i] <= mid ) i++;
      while( a[j] > mid ) j--;
      if( i>j || a[i] == a[j] ) break;
      swap( i, j );
   }
   swap( low, j );
   if( low<j-1 ) quick_sort( a, low, j-1 );
   if( j+1<high) quick_sort( a, j+1, high );
}
```

終了条件なし
⇒無限ループ

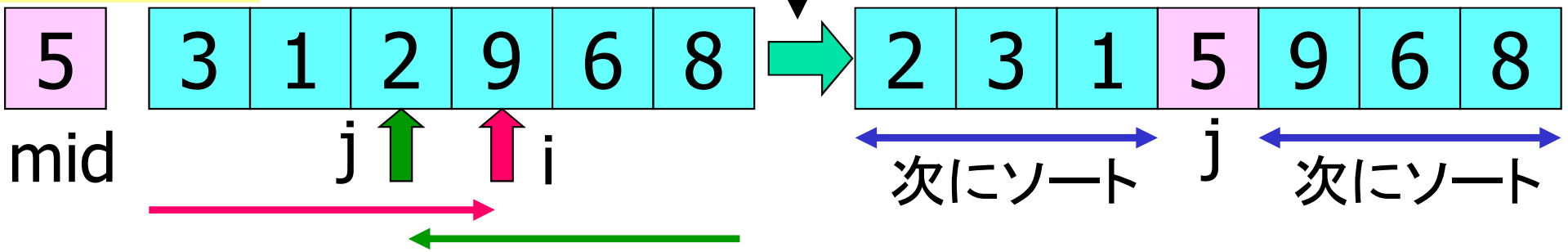
ループを
抜け出す

配列の分割

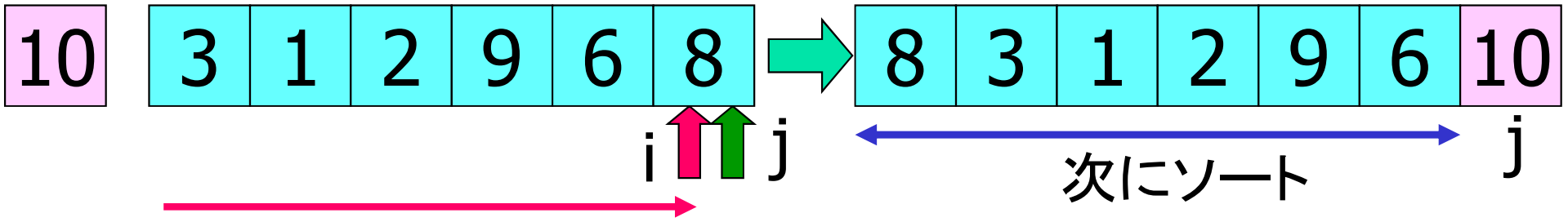
メモリを節約したクイックソート

for文の終了条件

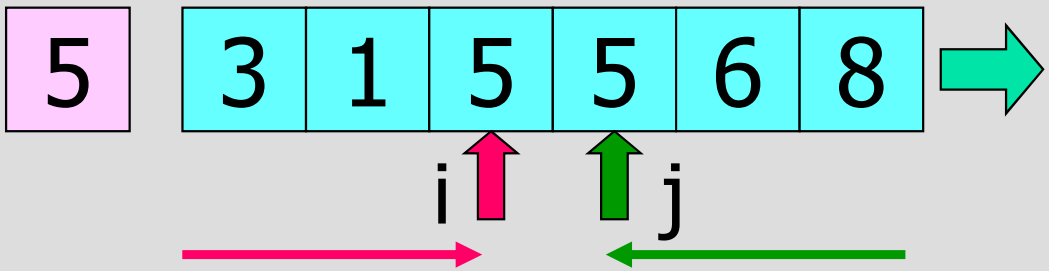
i>j の場合



a[i]==a[j] の場合(i=j。For文終了時にはi<jにならない)



a[i]==a[j] (i<j) の場合

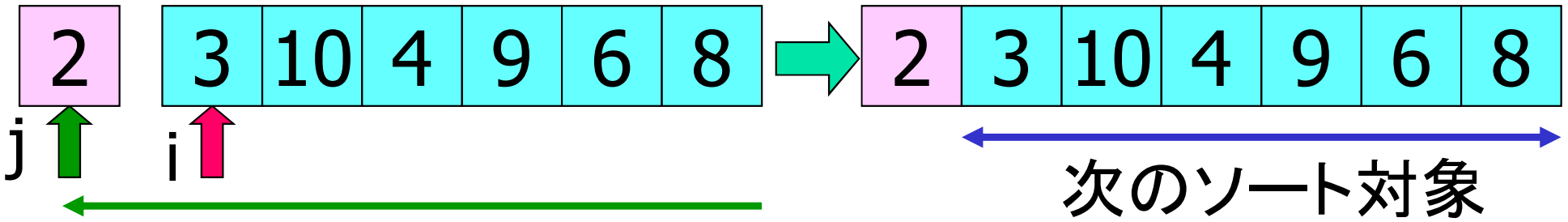


iはさらに右へ移動(大きく)できるので、for文終了時にはi<jでない

メモリを節約したクイックソート

特殊な場合：midがソート対象配列中の最小値

for文の終了時の状態



ソート対象配列を2個の配列に分割できない

ソート済みの配列にクイックソートを適用すると
アルゴリズム終了までに長い時間がかかる

クイックソートの実行時間

- ソートの理論限界を達成

  $O(n\log n)$

- ランダムなデータに対して高速
- ほとんど整列しているデータに適用すると実行時間は$O(n^2)$に悪化

- C言語ではクイックソートの関数が予め用意されている

  qsort関数