データ構造・アルゴリズム論 解説資料集 2

整列	2
分類法による整列	4
選択法による整列	6
挿入法による整列	8
交換法による整列	9
分割統治法による整列	10
グラフの基礎	12
グラフの深さ優先探索	14
グラフの幅優先探索	16
グラフの深さ優先探索と幅優先探索	18

山田 俊行

https://www.cs.info.mie-u.ac.jp/~toshi/lectures/algorithm/

2023 年 12 月

整列 (教科書 3 章)

データを整列する各種のアルゴリズムの，処理手順と計算量について学ぶ.

●整列の基礎

整列 (sorting)

データ列を指定された順序で並べ替えること

キー (key)

比較のために直接参照するデータの項目

整列法の分類

方法	名前
分類法	バケットソート，基数ソート
選択法	単純選択ソート，ヒープソート
挿入法	単純挿入ソート
交換法	バブルソート
分割統治法	マージソート，クイックソート

●整列アルゴリズム

バケットソート (教科書 p. 29 コード 3.1)

(1) m 個のキュー (バケット) を生成
(2) 入力を走査し，各データをキー値の番号のキューに挿入
(3) 番号順にキューからデータを全て取り出す

基数ソート (教科書 p. 30)

k 番目から 1 番目に向けて，各位置での値をキーとして，
バケットソート (などの安定な整列) を反復

(単純) 選択ソート (教科書 p. 31 コード 3.2)

未整列部分の最小値を整列済みの列に追加することを反復

ヒープソート (教科書 p. 43 コード 3.9)

(1) 空のヒープにデータを並び順に挿入
(2) ヒープから最小値を取り出すことを反復

(単純) 挿入ソート (教科書 p. 32 コード 3.3)

未整列のデータ一つを，整列部分の適切な位置に挿入することを反復

バブルソート (教科書 p. 33 コード 3.4)

データを走査しながら，隣り合う 2 値が逆順なら交換するのを，交換がなくなるまで反復

マージソート (教科書 p. 36 コード 3.5)

(1) 未整列データの 2 分割を反復

(2) 整列済み部分列の統合 (マージ) を反復

クイックソート (教科書 p. 39 コード 3.6)

(1) データ群から基準値を選び，値の大小による 2 分割を反復

(2) 分割済みの列の連結を反復

●整列アルゴリズムの時間計算量

整列法	最悪の時間計算量	
バケットソート	$\mathrm{O}(m+n)$	m はバケット数
基数ソート	$\mathrm{O}(k(m+n))$	m はバケット数，k は組の要素数
単純選択ソート	$\mathrm{O}(n^2)$	
ヒープソート	$\mathrm{O}(n\log n)$	
単純挿入ソート	$\mathrm{O}(n^2)$	
バブルソート	$\mathrm{O}(n^2)$	
マージソート	$\mathrm{O}(n\log n)$	
クイックソート	$\mathrm{O}(n^2)$	平均の時間計算量は $\mathrm{O}(n\log n)$

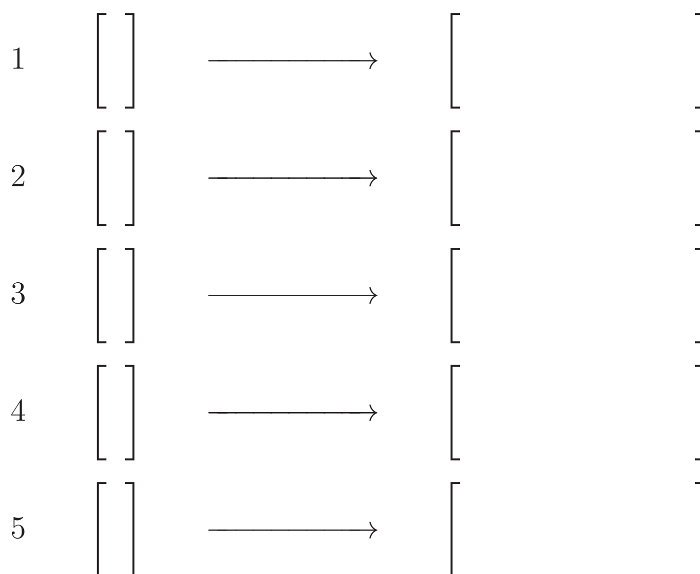
分類法による整列 (教科書 3.1 節)

分類法に基づく整列アルゴリズムとその計算量を，具体例を通して理解する．

●バケットソート

動作例

$$\begin{array}{lcccccc} & [1] & [2] & [3] & [4] & [5] & [6] \\ \text{入力} & \begin{pmatrix} 4 \\ \mathrm{A} \end{pmatrix} & \begin{pmatrix} 2 \\ \mathrm{A} \end{pmatrix} & \begin{pmatrix} 4 \\ \mathrm{B} \end{pmatrix} & \begin{pmatrix} 1 \\ \mathrm{B} \end{pmatrix} & \begin{pmatrix} 2 \\ \mathrm{B} \end{pmatrix} & \begin{pmatrix} 5 \\ \mathrm{C} \end{pmatrix} \end{array}$$



出力 ( ) ( ) ( ) ( ) ( ) ( )

計算量 (最悪の実行時間)

(1) create を 回
(2) insert を 回
(3) empty を 回，top と delete を各 回
キューの各基本操作 O( )
整列全体 O( )

●基数ソート

動作例

入力 $\begin{pmatrix}1\\7\\1\end{pmatrix}$ $\begin{pmatrix}2\\5\\8\end{pmatrix}$ $\begin{pmatrix}3\\2\\8\end{pmatrix}$ $\begin{pmatrix}1\\2\\3\end{pmatrix}$ $\begin{pmatrix}2\\5\\4\end{pmatrix}$

$\begin{pmatrix}\\ \\ \end{pmatrix}$ $\begin{pmatrix}\\ \\ \end{pmatrix}$ $\begin{pmatrix}\\ \\ \end{pmatrix}$ $\begin{pmatrix}\\ \\ \end{pmatrix}$ $\begin{pmatrix}\\ \\ \end{pmatrix}$

$\begin{pmatrix}\\ \\ \end{pmatrix}$ $\begin{pmatrix}\\ \\ \end{pmatrix}$ $\begin{pmatrix}\\ \\ \end{pmatrix}$ $\begin{pmatrix}\\ \\ \end{pmatrix}$ $\begin{pmatrix}\\ \\ \end{pmatrix}$

出力 $\begin{pmatrix}\\ \\ \end{pmatrix}$ $\begin{pmatrix}\\ \\ \end{pmatrix}$ $\begin{pmatrix}\\ \\ \end{pmatrix}$ $\begin{pmatrix}\\ \\ \end{pmatrix}$ $\begin{pmatrix}\\ \\ \end{pmatrix}$

計算量 (最悪の実行時間)

バケットソートを　　回

O(　　　　　　)　　k $\cdots$ 組の要素数

m $\cdots$ バケット数

n $\cdots$ データ数

選択法による整列 (教科書 3.2, 2.4 節)

選択法に基づく整列アルゴリズムとその計算量を，具体例を通して理解する．

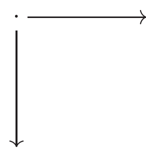
● (単純) 選択ソート

動作例

7	2	5	1	3	6
	2	5		3	6
1		5	7	3	6
1	2		7		6
1	2	3			6
1	2	3	5		

プログラム (教科書 p.31 コード 3.2)

2 重ループ



計算量 (実行時間)

O(　　)　　　比較回数

●ヒープソート

プログラム (教科書 p.43 コード 3.9)

(1) 空のヒープに，データを並び順に挿入

(2) ヒープからの，最小値の取り出しを反復

計算量 (最悪の実行時間)

(1)	create	O( )	回
	insert	O( )	回
(2)	findmin	O( )	回
	deletemin	O( )	回

整列全体 O( )

挿入法による整列 (教科書 3.2 節)

挿入法に基づく整列アルゴリズムとその計算量を，具体例を通して理解する．

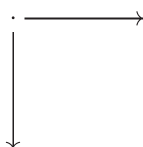
● (単純) 挿入ソート

動作例

7	2	5	1	3	6
		5	1	3	6
			1	3	6
				3	6
1	2				6
1	2	3	5		

プログラム (教科書 p.32 コード 3.3)

2 重ループ



時間計算量

最悪 ···

O( )　　比較回数

交換法による整列 (教科書 3.2 節)

交換法に基づく整列アルゴリズムとその計算量を，具体例を通して理解する．

●バブルソート

動作例

7	2	5	1	3	6
		5			
			1		
				3	
					6
2	5	1	3	6	7
			3		
				6	
2	1	3	5	6	7
		3	5		
1	2	3	5	6	7

時間計算量

最悪 ···

O( )

分割統治法による整列 (教科書 3.3, 3.4 節)

分割統治法に基づく二つの整列アルゴリズムを，呼び出し木を使って理解する．

●マージソートの動作例

配列の変化

$$[5, 6, 3, 7, 4, 1]$$

$$[5, 6, 3][7, 4, 1]$$

$$[5][6, 3][7, 4, 1]$$

$$[5][6][3][7, 4, 1]$$

$$[5][3, 6][7, 4, 1]$$

$$[3, 5, 6][7, 4, 1]$$

$$[3, 5, 6][7][4, 1]$$

$$[3, 5, 6][7][4][1]$$

$$[3, 5, 6][7][1, 4]$$

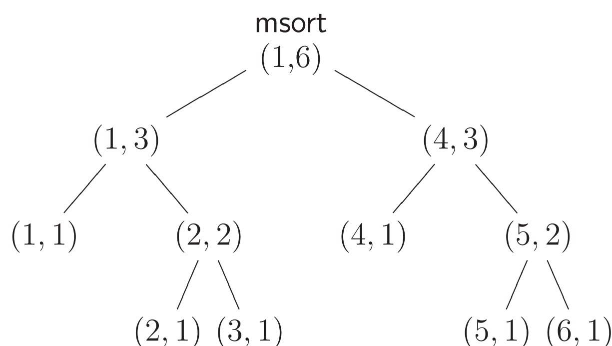
$$[3, 5, 6][1, 4, 7]$$

$$[1, 3, 4, 5, 6, 7]$$

再帰手続き (教科書 p.36 コード 3.5)

```
msort(p, n) {
    // p : 最左位置    n : 要素数
    if  (n ≤ 1)  return
    h ← n div 2
    msort(p,        h       )
    msort(p + h,  n − h)
    merge(p,        n       )
}
```

呼び出し木 (列長 $n = 6$)



●クイックソートの動作例

配列の変化

$$[5, 6, 3, 7, 4, 1]$$

$$[1, 4, 3][7, 6, 5]$$

$$[1][4, 3][7, 6, 5]$$

$$[1][3][4][7, 6, 5]$$

$$[1][3, 4][7, 6, 5]$$

$$[1, 3, 4][7, 6, 5]$$

$$[1, 3, 4][5, 6][7]$$

$$[1, 3, 4][5][6][7]$$

$$[1, 3, 4][5, 6][7]$$

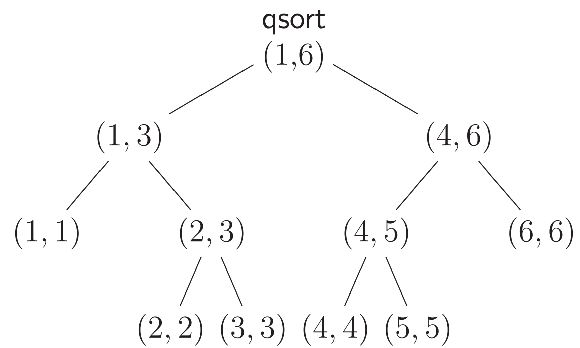
$$[1, 3, 4][5, 6, 7]$$

$$[1, 3, 4, 5, 6, 7]$$

再帰手続き (教科書 p.39 コード 3.6)

```
qsort(p, q) {
    // p : 最左位置　q : 最右位置
    if (p ≥ q) return
    (j, i) ← partition(p, q)
    qsort(p, j)
    qsort(i, q)
}
```

呼び出し木 (左の例のとき)



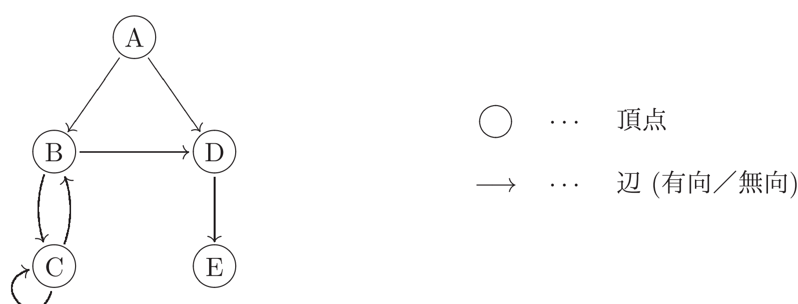
グラフの基礎 (教科書 1.4, 7.1 節)

グラフを使うと，列や木よりも自由に，データ (集合要素) の間の関係を表せる．

●グラフのデータ構造

グラフ (graph)

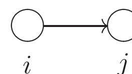
集合の要素間の (2 項) 関係を，点と点のつながりで表すもの



隣接行列 (adjacency matrix)

頂点間の辺の有無を，行列成分の 0, 1 (偽, 真) で表すデータ構造

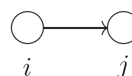
		A	B	C	D	E
		0	1	2	3	4
A	0	0	1	0	1	0
B	1	0	0	1	1	0
C	2					
D	3					
E	4					



隣接リスト (adjacency lists)

頂点から出る辺すべてを列で表すデータ構造

A	0	
B	1	
C	2	
D	3	
E	4	



無向グラフの有向グラフによる表現

- 無向辺を双方向の有向辺で表す
  (無向辺 $\{i, j\}$ を有向辺 $(i, j), (j, i)$ で表現)
- 無向辺を番号が小さい頂点から大きい頂点への有向辺で代表させる
  (無向辺 $\{i, j\}$ を有向辺 $(\min\{i, j\}, \max\{i, j\})$ で表現)

●グラフの基本操作の計算量

	隣接行列	隣接リスト
辺 (i, j) の有無の判定		
頂点 i から出る辺の有無の判定		
領域量		

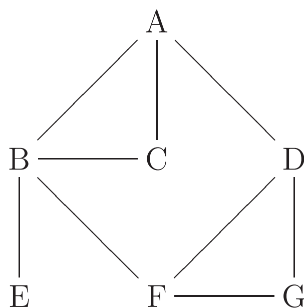
グラフの深さ優先探索　(教科書7.2節)

グラフの深さ優先探索のアルゴリズムを理解する．
頂点や辺を隣接関係に沿って順に処理しながら，条件に合う頂点や辺を見つけられる．

●深さ優先探索 (depth-first search, DFS) のアルゴリズム

```
 1:  dfs(v) {
 2:      v を処理し，Visited に v を挿入
 3:      foreach (w ∈ (G で v に隣接する頂点全体)) {
 4:          (v, w) を処理
 5:          if (not w ∈ Visited) {
 6:              // (v, w) は深さ優先スパニング木の辺
 7:              dfs(w)
 8:          }
 9:      }
10:  }
11:  main( ) {
12:      G を読み，Visited を生成          // G はグラフ，Visited は頂点集合
13:      while (not v ∈ Visited となる v がある ) {
14:          dfs(v)
15:      }
16:  }
```

●深さ優先探索の実行例



●グラフの探索プログラム

Python 言語によるプログラム例と実行結果を以下に示す.

```
$ cat dfs.py                              # プログラムファイルの内容を表示

def dfs(Adjacency, Visited, v):
    print(v)                              # 訪問頂点の処理
    Visited.add(v)
    for w in Adjacency[v]:
        # print((v,w))                    # 各辺の処理
        if w not in Visited:
            print((v,w))                  # スパニング木の各辺の処理
            dfs(Adjacency, Visited, w)

def main():
    Adjacency = {                         # 隣接リストを表す辞書
        'A': ['B','C','D'],               #     A       H
        'B': ['A','C','E','F'],           #   ／|＼     |
        'C': ['A','B'],                   # B—C   D    I
        'D': ['A','F','G'],               # |＼  ／|
        'E': ['B'],                       # E   F—G
        'F': ['B','D','G'],
        'G': ['D','F'],
        'H': ['I'],
        'I': ['H'],
        }
    Visited = set()                       # 頂点の空集合の生成
    for v in Adjacency.keys():
        if v not in Visited:
            dfs(Adjacency, Visited, v)

main()
```

このプログラムの実行結果 (各頂点の表示とスパニング森の各辺の表示) を以下に示す.

```
$ python3 dfs.py | tr -d "'" | fmt        # 実行結果を短く加工して表示

A (A, B) B (B, C) C (B, E) E (B, F) F (F, D) D (D, G) G H (H, I) I
```

リスト・集合・辞書 などの基本データ構造があらかじめ用意されている Python 言語を使うと，データ構造やアルゴリズムの実習が手軽にできる. 上記のプログラムでは，隣接リスト `Adjacency` を，辞書 (キーは頂点，値は頂点リスト) として実現した.

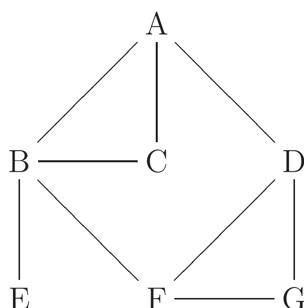
グラフの幅優先探索 (教科書7.2節)

グラフの幅優先探索は，深さ優先探索と同様に，頂点や辺を隣接関係に沿って系統的に走査するアルゴリズムである．幅優先探索では，開始頂点に近い頂点を優先して探索するため，始点から各頂点への最短の (通る辺数が最小の) 経路を求められる．

●幅優先探索 (breadth-first search, BFS) のアルゴリズム

```
 1: bfs(u) {
 2:     u を処理し，Visited に u を挿入
 3:     Waiting を生成            // Waiting はキューを表す頂点列
 4:     Waiting の末尾に u を追加
 5:     while (not Waiting が空) {
 6:         Waiting の先頭から v を取り出す
 7:         foreach (w ∈ (G で v に隣接する頂点全体)) {
 8:             if (not w ∈ Visited) {
 9:                 // (v, w) は幅優先スパニング木の辺
10:                 w を処理し，Visited に w を挿入
11:                 Waiting の末尾に w を追加
12:             }
13:         }
14:     }
15: }
16: main( ) {
17:     G を読み，Visited を生成            // G はグラフ，Visited は頂点集合
18:     while (not v ∈ Visited となる v がある) {
19:         bfs(u)
20:     }
21: }
```

●幅優先探索の実行例



●幅優先探索のプログラム

Python 言語によるプログラム例と実行結果 (各頂点の表示とスパニング森の各辺の表示) を以下に示す.

```
$ cat bfs.py                          # プログラムファイルの内容を表示

def bfs(Adjacency, Visited, u):
    print(u)                          # 訪問頂点の処理
    Visited.add(u)
    Waiting = [u]                     # 頂点を要素とするキューの リストによる実装
    while Waiting:
        v = Waiting.pop(0)            # 頂点の dequeue
        for w in Adjacency[v]:
            # print((v,w))            # 各辺の処理
            if w not in Visited:
                print((v,w))          # スパニング木の各辺の処理
                print(w)              # 訪問頂点の処理
                Visited.add(w)
                Waiting.append(w)     # 頂点の enqueue

def main():
    Adjacency = {                     # 隣接リストを表す辞書
        'A': ['B','C','D'],           #     A      H
        'B': ['A','C','E','F'],       #   ／|＼     |
        'C': ['A','B'],               # B―C  D    I
        'D': ['A','F','G'],           # |＼  ／|
        'E': ['B'],                   # E   F―G
        'F': ['B','D','G'],
        'G': ['D','F'],
        'H': ['I'],
        'I': ['H'],
        }
    Visited = set()                   # 頂点の空集合の生成
    for v in Adjacency.keys():
        if v not in Visited:
            bfs(Adjacency, Visited, v)

main()

$ python3 bfs.py | tr -d "'" | fmt    # 実行結果を短く加工して表示

A (A, B) B (A, C) C (A, D) D (B, E) E (B, F) F (D, G) G H (H, I) I
```

内側のループ本体の実行回数は，無向辺の数の 2 倍 (つまり双方向の有向辺の数) である．これを確かめるには，各スパニング木の辺を表示する `print((v,w))` の行を削除した後，`for` ループ本体の冒頭の `#` を消すことで `print((v,w))` を有効にしてから，プログラムを実行すればよい．

グラフの 深さ優先探索 と 幅優先探索 (教科書 7.2 節)

グラフの深さ優先探索は，再帰を使うと簡潔に書けるが，反復のアルゴリズムによっても記述できる．この反復アルゴリズムのデータ構造を一つだけを変えると，幅優先探索のアルゴリズムが得られる．

●深さ優先探索 (depth-first search, DFS)

```
 1: search(s) {            // s は探索開始の頂点
 2:     Waitingを生成            // Waiting はスタックを表す辺の列，末尾が最上段
 3:     Waiting に (s, s) を追加
 4:     Accepted に s を追加
 5:     while (not Waiting が空) {
 6:         (u, v) を Waiting の        から取り出す          // pop
 7:         // (u, v) は深さ優先スパニング木の辺
 8:         v を処理
 9:         foreach (w ∈ (G で v に隣接する頂点全体)) {
10:             if (not w ∈ Accepted) {
11:                 Waiting の末尾に (v, w) を追加          // push
12:                 Accepted に w を追加
13:             }
14:         }
15:     }
16: }
```

●幅優先探索 (breadth-first search, BFS)

```
 1: search(s) {            // s は探索開始の頂点
 2:     Waiting を生成            // Waiting はキューを表す辺の列
 3:     Waiting に (s, s) を追加
 4:     Accepted に s を追加
 5:     while (not Waiting が空) {
 6:         (u, v) を Waiting の        から取り出す          // dequeue
 7:         // (u, v) は幅優先スパニング木の辺
 8:         v を処理
 9:         foreach (w ∈ (G で v に隣接する頂点全体)) {
10:             if (not w ∈ Accepted) {
11:                 Waiting の末尾に (v, w) を追加          // enqueue
12:                 Accepted に w を追加
13:             }
14:         }
15:     }
16: }
```

●探索の呼び出し

```
1: main( ) {
2:     G を読み，Accepted を生成          // G はグラフ，Accepted は頂点集合
3:     while (not v ∈ Accepted となる v がある) {
4:         search(v)
5:     }
6: }
```

●グラフ探索のプログラム

Python 言語によるプログラム例と実行結果を以下に示す.

```
$ cat search.py                       # プログラムファイルの内容を表示

def dfs(Adjacency, Accepted, start):
    Waiting = [(None, start)]         # 辺を要素とするスタックの リストによる実装
    Accepted.add(start)               # 探索を開始
    while Waiting:
        (u,v) = Waiting.pop()         # 辺の pop
        if u is not None: print((u,v))  # スパニング木の辺
        print(v)                      # 頂点の訪問
        for w in reversed(Adjacency[v]):
            if w not in Accepted:
                Waiting.append((v,w))   # 辺の push
                Accepted.add(w)

def bfs(Adjacency, Accepted, start):
    Waiting = [(None, start)]         # 辺を要素とするキューの リストによる実装
    Accepted.add(start)               # 探索を開始
    while Waiting:
        (u,v) = Waiting.pop(0)        # 辺の dequeue
        if u is not None: print((u,v))  # スパニング木の辺
        print(v)                      # 頂点の訪問
        for w in Adjacency[v]:
            if w not in Accepted:
                Waiting.append((v,w))   # 辺の enqueue
                Accepted.add(w)

def main(search):
    Adjacency = {                     # 隣接リストを表す辞書
        1: [2, 3, 4],                 #     1       8
        2: [1, 3, 5, 6],              #   ／|＼     |
        3: [1, 2],                    # 2—3   4    9
        4: [1, 6, 7],                 # |＼   ／|
        5: [2],                       # 5   6—7
        6: [2, 4, 7],
        7: [4, 6],
        8: [9],
        9: [8],
        }
    Accepted = set()                  # 探索処理を開始済みの頂点の集合
    for v in sorted(Adjacency.keys()):
        if v not in Accepted:
            search(Adjacency, Accepted, v)

print('DFS:')
main(dfs)
print('')
print('BFS:')
main(bfs)

$ python3 search.py | fmt             # 実行結果を短く加工して表示

DFS: 1 (1, 2) 2 (2, 5) 5 (2, 6) 6 (6, 7) 7 (1, 3) 3 (1, 4) 4 8 (8, 9) 9

BFS: 1 (1, 2) 2 (1, 3) 3 (1, 4) 4 (2, 5) 5 (2, 6) 6 (4, 7) 7 8 (8, 9) 9
```