

p.2 下から4行目

誤  $r \leftarrow m; n \leftarrow r;$  正  $m \leftarrow n; n \leftarrow r;$

p.6 表1. 1

誤 正

n	線形探索	2分探索
1	1ms	1ms
1000	1sec	7ms
1000000	17min	14ms
1000000000	12days	21ms

n	線形探索	2分探索
1	1ms	1ms
1000	1sec	10ms
1000000	17min	20ms
1000000000	12days	30ms

p.11 3行目

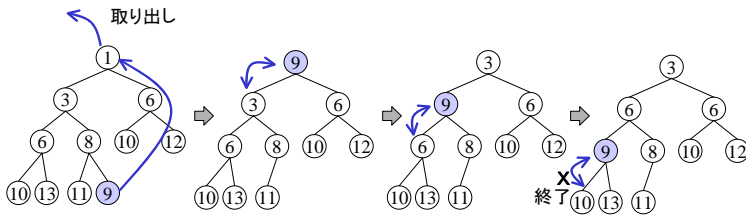
誤 長さの変わる操作を多く 正 長さの変わる操作が多く

p.16 問題3の最後

誤  $\rightarrow \text{enqueue}(25) \rightarrow \text{enqueue}$  正  $\rightarrow \text{enqueue}(25) \rightarrow \text{dequeue}$

p.20 図3.4

正



p.22 ヒープの擬似コードの下から5行目

誤 index 正 i

p.25 2分探索木の擬似コード中 delete(node, object){ 以下

正

```

delete(node, object){
  if (node = null)
    return null;
  if (object < node.object)
    node.left ← delete(node.left, object); //左の子を辿る
    return node;
  else if (object > node.object)
    node.right ← delete(node.right, object); //右の子を辿る
    return node;
  else // この node を削除する
    if (node.left = null and node.right = null) // 子がない
      return null;
    else if (node.left ≠ null and node.right = null) //左の子のみ
      return node.left;
    else if (node.left = null and node.right ≠ null) // 右の子のみ
      return node.right;
    else //子が2つ
      min ← deletemin(node.right, node); //右の子孫の最小値
      min.right ← node.right;
      min.left ← node.left;
      return min;
}
deletemin(node, parent){ //子孫のうちの最小値をとってくる
  if (node.left = null) // 右にしか子がない
    if (parent.left == node) // 自分自身のいた場所に右の子を配置する
      parent.left = node.right;

```

```

else
    parent.right = node.right;
    return node; //自分自身が最小値
parent ← node
node ← node.left;
while(node.left ≠ null) //左の子を辿る
    parent ← node;
    node ← node.left;
parent.left ← node.right;
return node;
}

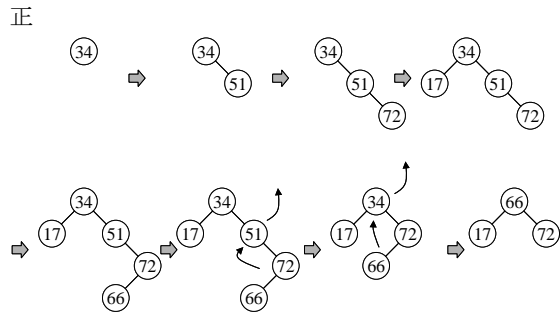
```

p.27 2行目

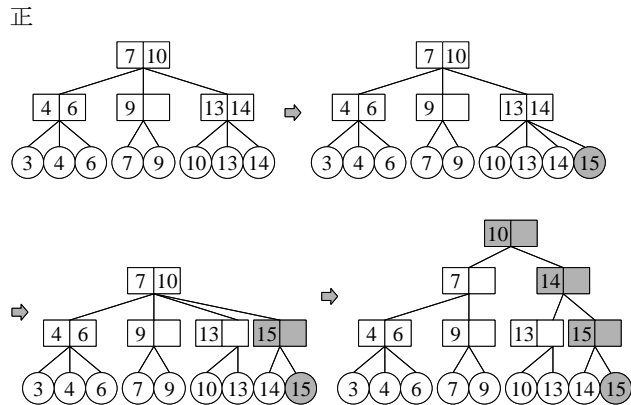
誤 
$$T(n) = \frac{1}{n} \sum_{i=0}^{n-1} \left[ \frac{1}{n} + \frac{i}{n} \{T(i)+1\} + \frac{n-i-1}{n} T\{(n-i-1)+1\} \right]$$

正 
$$T(n) = \frac{1}{n} \sum_{i=0}^{n-1} \left[ \frac{1}{n} + \frac{i}{n} \{T(i)+1\} + \frac{n-i-1}{n} \{T(n-i-1)+1\} \right]$$

p.28 図 3.11



p.30 図 3.13



p.31 下から 6 行目

```

誤 new_child ← child.insert(child, object); // child ← object を挿入
正 new_child ← insert(child, object); // child ← object を挿入

```

p.32 上から 15 行目

```

誤 result ← delete(child, object); // Child ← object を挿入
正 result ← delete(child, object); // Child から object を削除

```

p.36 最後の行

```

if (hash.data[index]=empty or deleted) //空き発見
のインデントを一つ深くする (全体を右へ 2 文字ずらす)

```

p.37 上から 8 行目

repeat  
を太字にする。

p.39 図 3.19 図内左下(bird の下あたり)の白字の「開番地法」を削除

p.49 クイックソートの擬似コード

正 (2行目を追加)

```

QuickSort(int[] array, int i, int j){
    if ( i>=j ) return array;
    pivot ←array の最初の 2 つの値のうち大きいもの
    array0 ←QuickSort(pivot より小さい要素を集めたもの);
    array1 ←QuickSort(pivot 以上の要素を集めたもの);
    return array0 の後に array1 をつなげたもの
}

```

p.50 図 4.3 2 段目

誤 6 2 8 4 7 4 9

正

6 2 8 4 7 3 9

p.50 図 4.4 下

誤  $O(n^2) < ? < O(\log n)$

正

$O(n^2) < ? < O(n \log n)$

p.51 下から 4 行目

誤  $a=1+T(1)$ とすれば

正

$a=c+T(1)$ とすれば

p.52 1 行目

誤  $\sum$

正

$\sum_{i=1}^{k-1}$

p.52 3 行目と 4 行目

誤  $\sum_{i < k/2}^{k-1}$

正

$\sum_{i=k/2}^{k-1}$

p.52 下から 4 行目

誤  $T(k) \leq k \log k$

正  $T(k) \leq a k \log k$

p.53 マージソートの擬似コード

正 (2行目を追加)

```

MergeSort(array){
    if (array.length = 1) return array;
    array0 ←MergeSort(array の前半)
    array1 ←MergeSort(array の後半)
    return merge(array0, array1);
}

```

p.53 merge の擬似コードの中の 4~7 行目

正

```

if ((i < array0.length かつ (array0[i] < array1[j] または j >= array1.length))
    { result.add(array0[i]); i ← i+1; }
if ((j < array1.length かつ (array0[i] > array1[j] または i >= array0.length))
    { result.add(array1[j]); j ← j+1 }

```

p.54 下から 2 行目

誤 効率のよい作業を実現する.

正 効率のよい作業を実現する (まず最初に整列すべき要素の列を機械的に半分に分割して 2 つのファイルにしておく).

p.55 図 4.6 のキャプション

誤 本ファイルを利用した

正

2 本のファイルを利用した

p.59

ページ右下の黒い四角（解答の終了を示すマーク）を図4.9の右下へ移動する

p.66 3行目以降全体

正：  
ダイクストラのアルゴリズムの計算量は、外側のループ(a)が n 回まわり、内側の処理(b, c)に O(n)かかるので、合計で O(n<sup>2</sup>)である。しかし、辺の数 e が節点の数 n と同程度の場合には、節点からでている辺の集合をリストで管理し、(b)の処理を優先度付待ち行列を用いて効率化することで、全体の計算量を O((e+n) log n)に抑えることができる。この場合に擬似コードは以下のようになる。

```

ダイクストラ(有向グラフ(V,E), 辺のコスト d[], 出発点 s){
  for(v in V) // 初期化。直接移動のコストを C にセットする。
    C[v] ← d[s, v];
  ヒープ S に V のすべての要素 v を加える。順序は C[v] を基準とする。
  while( S が空でない)
    w ← S から C[w] が最小の頂点 w を取り出す
    for ( v ← w から出ている辺の行き先)
      if ( C[w] + d[w, v] < C[v] )
        C[v] ← C[w] + d[w, v] //コストの更新
        ヒープにおける v の位置を C[v] に従って繰り上げる
        (逆転がなくなるまで上に上げていく。)
  return C;
}

```

ヒープへの要素の追加、位置の更新・最小値の取り出し、にはそれぞれ O(log n)かかる。最小値を取り出す操作は n 回、位置の更新は合計で最大 e 回実行されるので、全体では O((n+e) log n)となる。一般的に辺の数は頂点の数と同程度のオーダーである疎なグラフであることが多いので、全体の計算量は O(n log n)となり、このアルゴリズムが有効である。しかし、すべての頂点間が結ばれた完全グラフ(e=n<sup>2</sup>)のような場合には O(n<sup>2</sup> log n)となり、逆に効率が悪くなるので注意が必要である。

p.67 8行目

誤 フロイド(有向グラフ(V,E), 辺のコスト d[], 出発点 s){  
正 フロイド(有向グラフ(V,E), 辺のコスト d[]){

p.68 図 5.4 一番右下の表の 3 段目 b の右隣の 6 を青の太字にする

p.69 図 5.5 左下  
誤 クラス 正 グラフ

p.70 2行目  
誤 dfs(v, visited) 正 dfs(u, visited)

p.71 図 5.7  
「帰りがけ」「深さ優先探索」の文字と、そこから出ている青い矢印を消す  
「深さ優先探索 (帰りがけに番号をふる)」の文字を図中右側の深さ優先の極大森の上 (a と c のノードの上) にもってくる。

p.72 図 5.8  
右側の探索木で、 f から c へ向かう点線矢印を加え、「交差」の文字をつける。また、 e から f の間の双方向矢印は e から f への単方向矢印とする。

p.78 下から 2 行目  
誤 T ← s; 正 T ← { }

p.79 3 行目  
誤 T に辺 { u, N[w] } を加える 正 T に辺 { w, N[w] } を加える

p.79 図 6.3 の上の 2 行にわたる「プリムのアルゴリズムの計算量は、」以下の文は、 p.79 の図 6.3 の後にもってくる。  
解答終了の四角印は、図 6.3 の後、移動した 1 文の前にくるようにする

p.79 図 6.3 の右上、⑧から出ている矢印は右上を向く

p.80 11 行目

誤 while(|T|<|V|){ 正 while(U が空でない){

p.81 図 6.5

最初のグラフで一番上のノードは黒でなく白とする。2 番目のグラフでコスト 1 の辺で結ばれたノードは両方とも黒にする。

p.87 表 7.1 右下

誤 aabaab 正 aabaab  
121555 121455

p.87 図 7.2 下半分

a と b の間を指している i は、ぴったり a を指すように、微妙に左にずらす  
c と a の間を指している j は、ぴったり c を指すように、微妙に左にずらす

p.89 3 行目

誤 O(m/n) 正 O(n/m)

p.90 2 行目

誤 max(skip[text[i+j]]+j-m+j+1, 1) 正 max(skip[text[i+j]]+j-m+1, 1)

p.90 8 行目

誤 while(j>=0 and text[i+j]=pattern[i]); 正 while(j>=0 and text[i+j]=pattern[j]);

p.91 図 7.4

パターンの文字列が abcabc になっているのを abcdbc に直す。

p.96 9 行目

誤 bcaba 正 bcba

p.96 擬似コード内

2 行目の末尾に以下の文を追加 (表の範囲外はすべて 0 とする)

3 行目~8 行目 インデントを一段浅くする (2 文字左にずらす)

p.97 表 8.1

正

				Y			
		b	d	c	a	b	a
	a	0	0	0	1	1	1
	b	1	1	1	1	2	2
	v	1	1	2	2	2	2
X	b	1	1	2	2	3	3
	d	1	2	2	2	3	3
	a	1	2	2	3	3	4
	b	1	2	2	3	4	4

p.111 skip の表

誤

ささうささん

1 2 3 4 5 3

正

ささうささん

1 2 1 4 5 3