理解度の確認；解説

（1　章）

問 1.1　以下に財布を表すクラス Purse の実装例を示す．指定された財布 q の中身を足し合わせるメンバ関数 void add(Purse& q) では，19 行目で q の中身を空にしている．もし q を値渡しした場合は，呼び出し元の変数（この例では 32 行目の p2）に影響を与えることはできないことに注意してほしい．

```
1  #include <iostream>
2  using namespace std;
3
4  class Purse {
5  private:
6    int h;     // １００円硬貨の枚数
7    int f;     // ５０円硬貨の枚数
8    int t;     // １０円硬貨の枚数
9  public:
10   // コンストラクタ
11   Purse(int _h, int _f, int _t) { h = _h; f = _f; t = _f; }
12   // 合計金額を返す関数
13   int total() const { return 100*h + 50*f + 10*t; }
14   // 指定された財布 q の中身を足し合わせる関数
15   void add(Purse& q) {
16     h += q.h;
17     f += q.f;
18     t += q.t;
19     q.h = q.f = q.t = 0;     // 財布 q の中身を空に
20   }
21   // 各硬貨の枚数をストリームに出力するためのフレンド関数
22   friend ostream& operator<< (ostream& stream, const Purse& p) {
23     return stream << "100*" << p.h << "+50*" << p.f << "+10*" << p.t
24                   << "=" << p.total();
25   }
26 };
27
28 int main(void) {
29   Purse p1(1, 2, 3);     // ２３０円の財布
30   Purse p2(4, 5, 6);     // ７１０円の財布
31   cout << p1 << ", " << p2 << endl;
32   p1.add(p2);
33   cout << p1 << ", " << p2 << endl;
34   return 0;
35 }
```

（2　章）

問 2.1　クラステンプレート List における以下のメンバ関数

- リストの n 番目の要素を取得するメンバ関数 getAt(int n)
- リストの n 番目に要素 e を追加するメンバ関数 addAt(int n, int e)
- リストの n 番目の要素を削除するメンバ関数 removeAt(int n)
- リストの末尾要素を削除するメンバ関数 removeLast()
- リストの長さを返すメンバ関数 size()

の定義例を以下に示す．これらのメンバ関数以外の定義は，リスト **2.3** と同じであるため省略している．また getAt, removeAt では n 番目の要素が存在しない場合には assert によりエラーメッセージを出力してプログラムの実行を停止している．同様に addAt ではリストの長さが n より短い場合に，removeLast ではリストが空の場合にエラーを出力して停止する．

```
1  template <class T>
2  class List {
3      ⋮
4  public:
5      ⋮
6    T getAt(int n) {
7      Cell *p = head;
8      while (n > 0) {
9        p = p->next;
10       assert(p != NULL);
11       n--;
12     }
13     return p->data;
14   }
15
16   void addAt(int n, T e) {
17     if (n == 0) {
18       addFirst(e);                // 先頭に追加する場合のみ例外的に処理
19       return;
20     }
21     // 挿入位置の１つ前の要素を探す
22     Cell *prev = head;
23     while (n > 1) {
24       prev = prev->next;
25       assert(prev != NULL);
26       n--;
27     }
28     // 挿入する
29     prev->next = new Cell(e, prev->next);
30   }
31
32   T removeAt(int n) {
33     assert(!isEmpty());           // リストは空ではないこと
34     if (n == 0)
35       return removeFirst();       // 先頭要素を削除する場合のみ例外的に処理
36     // 削除する要素の１つ手前の要素を探す
37     Cell *prev = head;
38     Cell *curr = head->next;
39     while (n > 1) {
40       assert(curr != NULL);
41       prev = curr;
42       curr = curr->next;
43       n -= 1;
44     }
45     // 要素を削除する
```

```
46      assert(curr != NULL);
47      prev->next = curr->next;
48      T data = curr->data;
49      delete curr;
50      return data;
51    }
52
53    T removeLast() {
54      assert(!isEmpty());           // リストは空ではないこと
55      if (head->next == NULL)
56        return removeFirst();       // 先頭要素しか存在しない場合は例外的に処理
57      // 末尾要素の１つ前の要素を探す
58      Cell *p = head;
59      while (p->next->next != NULL) p = p->next;
60      // 末尾要素を削除
61      T data = p->next->data;
62      delete p->next;
63      p->next = NULL;
64      return data;
65    }
66
67    int size() {
68      int length = 0;
69      for (Cell *p = head; p != NULL; p = p->next)
70        length++;
71      return length;
72    }
73  };
```

問 2.2 末尾要素へのポインタ付き片方向リストの定義例を以下に示す．クラス List のメンバ変数として先頭要素へのポインタ head のみでなく，末尾要素を指し示すポインタ tail を新たに追加している（16 行目）．これに伴い，要素の追加や削除などで末尾要素が変化する場合には，それに応じて tail を適宜更新する必要がある．例えば，リストの先頭に要素を追加する場合であっても，空のリストに追加する場合は，追加した要素は末尾要素でもあるため tail を更新する必要がある（26 行目）．

リストが先頭要素へのポインタしか持たない場合，末尾に新しい要素を追加するには先頭から末尾まで順に辿る必要があったが，末尾要素へのポインタがあれば一定時間で高速に末尾に要素を追加することが可能になる（33,34 行目）．

```
1   #include <iostream>
2   #include <cassert>
3   using namespace std;
4
5   template <class T>
6   class List {
7   private:
8     class Cell {
9     public:
10      T data;
11      Cell *next;
12      Cell(T d, Cell *n=NULL) { data = d; next = n; }
13    };
14
15    Cell *head;                          // リストの先頭要素へのポインタ
16    Cell *tail;                          // リストの末尾要素へのポインタ
17
18  public:
19     List() { head = tail = NULL; }
20    ~List() { while (!isEmpty()) removeFirst(); }
```

```
21
22    bool isEmpty() const { return head == NULL; }
23
24    void addFirst(T data) {
25      head = new Cell(data, head);
26      if (tail == NULL) tail = head;     // リストが空であったならば tail を更新
27    }
28
29    void addLast(T data) {
30      if (head == NULL)
31        addFirst(data);
32      else {
33        tail->next = new Cell(data);     // 末尾要素の次要素として追加
34        tail = tail->next;               // tail を更新
35      }
36    }
37
38    T removeFirst() {
39      assert(!isEmpty());
40      Cell *old_head = head;
41      T data = old_head->data;
42      head = head->next;
43      delete old_head;
44      if (head == NULL) tail = NULL;     // リストが空になったならば, tail も NULL に
45      return data;
46    }
47
48    T getAt(int n) {
49      Cell *p = head;
50      while (n > 0) {
51        p = p->next;
52        assert(p != NULL);
53        n--;
54      }
55      return p->data;
56    }
57
58    void addAt(int n, T e) {
59      if (n == 0) {
60        addFirst(e);                 // 先頭に追加する場合のみ例外的に処理
61        return;
62      }
63      // 挿入位置の１つ前の要素を探す
64      Cell *prev = head;
65      while (n > 1) {
66        prev = prev->next;
67        assert(prev != NULL);
68        n--;
69      }
70      // 挿入する
71      prev->next = new Cell(e, prev->next);
72      // 末尾要素の次要素として追加したならば, tail を更新する
73      if (prev == tail) tail = prev->next;
74    }
75
76    T removeAt(int n) {
77      assert(!isEmpty());            // リストは空ではないこと
78      if (n == 0)
79        return removeFirst();        // 先頭要素を削除する場合のみ例外的に処理
80      // 削除する要素の１つ手前の要素を探す
81      Cell *prev = head;
82      Cell *curr = head->next;
83      while (n > 1) {
84        assert(curr != NULL);
```

```
85        prev = curr;
86        curr = curr->next;
87        n -= 1;
88      }
89      // 要素を削除する
90      assert(curr != NULL);
91      prev->next = curr->next;
92      T data = curr->data;
93      delete curr;
94      // 末尾要素を削除したならば，tail を更新する
95      if (prev->next == NULL) tail = prev;
96      return data;
97    }
98
99    T removeLast() {
100     assert(!isEmpty());            // リストは空ではないこと
101     if (head->next == NULL)
102       return removeFirst();        // 先頭要素しか存在しない場合は例外的に処理
103     // 末尾要素の 1 つ前の要素を探す
104     Cell *p = head;
105     while (p->next->next != NULL) p = p->next;
106     // 末尾要素を削除
107     T data = p->next->data;
108     delete p->next;
109     p->next = NULL;
110     // tail を更新する
111     tail = p;
112     return data;
113   }
114
115   int size() {
116     int length = 0;
117     for (Cell *p = head; p != NULL; p = p->next)
118       length++;
119     return length;
120   }
121
122   friend ostream& operator << (ostream& stream , const List& list) {
123     stream << "[";
124     if (list.head != NULL) {
125       stream << list.head->data;
126       for (Cell *p = list.head->next; p != NULL; p = p->next)
127     stream << "," << p->data;
128     }
129     return stream << "]";
130   }
131 };
```

問 2.3 末尾要素へのポインタ付き双方向リストの定義例を以下に示す．リストの要素を表す内部クラス Cell に，次の要素へのポインタ next だけでなく，前の要素へのポインタ prev を持たせている（11 行目）．これらのポインタは Cell のコンストラクタにおいて初期化される．コンストラクタの引数 d, p, n（13 行目）は，それぞれ要素が保持するデータ data，前要素へのポインタ prev，次要素へのポインタ next の初期値である．例えばリストの先頭に要素を追加する場合，新しい要素は new Cell(data, NULL, head) のように（26 行目），旧先頭要素を次要素（第 3 引数）として指定することで生成できる．同様に末尾に追加する場合は，new Cell(data, tail) のように（36 行目），旧末尾要素を前要素（第 2 引数）に指定することで生成できる．またリストから要素を削除した場合には，その前後の要素の next, prev ポインタを適宜更新する必要がある（例えば 97–99 行目など）．

リストに末尾要素へのポインタと，リストの各要素に前後要素へのポインタを持たせることで，リストの末尾要素の追加と削除ではリストを先頭から順に辿る必要がなくなり，一定時間で処理を実行することが可能になる（36,37,112–118 行目）.

```
1  #include <iostream>
2  #include <cassert>
3  using namespace std;
4
5  template <class T>
6  class List {
7  private:
8    class Cell {
9    public:
10     T data;
11     Cell *prev;                        // 前要素へのポインタ
12     Cell *next;                        // 次要素へのポインタ
13     Cell(T d, Cell *p=NULL, Cell *n=NULL) { data = d; prev = p; next = n; }
14   };
15
16   Cell *head;                          // リストの先頭要素へのポインタ
17   Cell *tail;                          // リストの末尾要素へのポインタ
18
19 public:
20   List() { head = tail = NULL; }
21   ~List() { while (!isEmpty()) removeFirst(); }
22
23   bool isEmpty() { return head == NULL; }
24
25   void addFirst(T data) {
26     head = new Cell(data, NULL, head);
27     if (tail == NULL) tail = head;     // リストが空であったならば tail を更新
28     if (head->next != NULL)
29       head->next->prev = head;         // 旧先頭要素の前要素として head を登録
30   }
31
32   void addLast(T data) {
33     if (head == NULL)
34       addFirst(data);
35     else {
36       tail->next = new Cell(data, tail);  // 末尾要素の次要素として追加
37       tail = tail->next;                  // tail を更新
38     }
39   }
40
41   T removeFirst() {
42     assert(!isEmpty());
43     Cell *old_head = head;
44     T data = old_head->data;
45     head = head->next;
46     delete old_head;
47     if (head == NULL)
48       tail = NULL;                     // リストが空になったならば，tail も NULL に
49     else
50       head->prev = NULL;               // 先頭要素の前要素はない
51     return data;
52   }
53
54   T getAt(int n) {
55     Cell *p = head;
56     while (n > 0) {
57       p = p->next;
58       assert(p != NULL);
59       n--;
```

```
60      }
61      return p->data;
62    }
63
64    void addAt(int n, T e) {
65      if (n == 0) {
66        addFirst(e);                // 先頭に追加する場合のみ例外的に処理
67        return;
68      }
69      // 挿入位置の１つ前の要素を探す
70      Cell *prev = head;
71      while (n > 1) {
72        prev = prev->next;
73        assert(prev != NULL);
74        n--;
75      }
76      // 挿入する
77      prev->next = new Cell(e, prev, prev->next);
78      // 末尾要素の次要素として追加したならば，tail を更新する
79      if (prev == tail) tail = prev->next;
80    }
81
82    T removeAt(int n) {
83      assert(!isEmpty());           // リストは空ではないこと
84      if (n == 0)
85        return removeFirst();       // 先頭要素を削除する場合のみ例外的に処理
86      // 削除する要素の１つ手前の要素を探す
87      Cell *prev = head;
88      Cell *curr = head->next;
89      while (n > 1) {
90        assert(curr != NULL);
91        prev = curr;
92        curr = curr->next;
93        n -= 1;
94      }
95      // 要素を削除する
96      assert(curr != NULL);
97      prev->next = curr->next;
98      if (curr->next != NULL)
99        curr->next->prev = prev;
100     T data = curr->data;
101     delete curr;
102     // 末尾要素を削除したならば，tail を更新する
103     if (prev->next == NULL) tail = prev;
104     return data;
105   }
106
107   T removeLast() {
108     assert(!isEmpty());           // リストは空ではないこと
109     if (head->next == NULL)
110       return removeFirst();       // 先頭要素しか存在しない場合は例外的に処理
111     // p は末尾要素の１つ前の要素
112     Cell *p = tail->prev;
113     // 末尾要素を削除
114     T data = p->next->data;
115     delete p->next;
116     p->next = NULL;
117     // tail を更新する
118     tail = p;
119     return data;
120   }
121
122   int size() {
123     int length = 0;
```

```
124     for (Cell *p = head; p != NULL; p = p->next)
125       length++;
126     return length;
127   }
128
129   friend ostream& operator << (ostream& stream , const List& list) {
130     stream << "[";
131     if (list.head != NULL) {
132       stream << list.head->data;
133       for (Cell *p = list.head->next; p != NULL; p = p->next)
134     stream << "," << p->data;
135     }
136     return stream << "]";
137   }
138 };
```

問 2.4 固定長配列をリングバッファと見なして作成したキューの実装例を以下に示す．コンストラクタの引数で指定された大きさの固定長配列を確保し（16 行目），それをリングバッファとして用いている．head は先頭要素の配列上の添字を表し，tailは末尾要素の次要素の添字を表す（9,10 行目）．cap は固定長配列の大きさ（11 行目），sz はキューが保持している要素数を表している（12 行目）．25 行目の tail = (tail + 1) % cap; は，添字が配列の大きさを超えた場合に 0 に戻すことを意味している（% は剰余演算子）．32 行目の head の更新式も同様である．

```
1  #include <iostream>
2  #include <cassert>
3  using namespace std;
4
5  template <class T>
6  class FixedRingQueue {
7  private:
8    T   *data;                          // 要素を保存するための固定長配列
9    int  head;                          // キューの先頭要素の添字
10   int  tail;                          // キューの末尾要素の次要素の添字
11   int  cap;                           // キューが保持可能な最大要素数
12   int  sz;                            // キューが保持している要素数
13 public:
14   FixedRingQueue(int _cap) {
15     cap  = _cap;
16     data = new T[cap];                // 引数で指定された大きさの固定長配列を確保
17     head = tail = sz = 0;
18   }
19
20   ~FixedRingQueue() { delete data; }  // 確保した固定長配列を破棄
21
22   void enqueue(T n) {
23     assert(sz < cap);                 // キューが満杯でないこと
24     data[tail] = n;                   // 末尾要素として登録
25     tail = (tail + 1) % cap;          // 末尾要素の次要素の添字更新
26     sz++;
27   }
28
29   T dequeue() {
30     assert(sz > 0);                   // キューが空でないこと
31     T n = data[head];                 // 先頭要素を取得
32     head = (head + 1) % cap;          // 先頭要素の添字更新
33     sz--;
34     return n;
35   }
36
```

```
37    int  size()     { return sz;        }
38    int  capacity() { return cap;       }
39    bool isFull()   { return sz == cap; }
40    bool isEmpty()  { return sz == 0;   }
41
42    friend ostream& operator << (ostream& stream , const FixedRingQueue& q)
43    {
44      stream << "[";
45      if (q.sz > 0) {
46        int h = q.head;
47        while (true) {
48          stream << q.data[h];
49          h = (h + 1) % q.cap;
50          if (h == q.tail) break;
51          stream << " ";
52        }
53      }
54      return stream << "]";
55    }
56  };
57
58  int main(void) {
59    FixedRingQueue<int> q(5);     // サイズ5のキューを作成
60
61    // 動作確認のためのループ
62    for (int i=0; i < 6; i++) {
63      // キューが満杯になるまで要素をエンキュー
64      int n = 1;
65      while (!q.isFull()) {
66        q.enqueue(n);
67        cout << "enqueue(" << n << ") " << q << endl;
68        n++;
69      }
70      // キューの要素数が1つになるまでデキュー
71      while (q.size() > 1)
72        cout << "dequeue = " << q.dequeue() << " " << q << endl;
73    }
74    return 0;
75  }
```

問 2.5 キューの大きさが不足した場合に自動的に伸張するキューの定義例を以下に示す．ここでは問 **2.4** の解答例で示したクラス FixedRingQueue を利用してサイズが自動的に伸張するキューを定義している．16–23 行目において，もしキューが満杯ならば，サイズを約 2 倍に大きくしたキューを新たに作成し，すべての要素をコピーすることでキューの大きさが不足することを回避している．

```
1   template <class T>
2   class VarRingQueue {
3   private:
4     FixedRingQueue<T> *fixed_queue;                    // 固定長配列に基づくキュー
5
6   public:
7
8     VarRingQueue(int init_cap = 0) {
9       fixed_queue = new FixedRingQueue<T>(init_cap);   // 初期サイズでキューを作成
10    }
11
12    ~VarRingQueue() { delete fixed_queue; }            // 確保したキューを破棄
13
14    void enqueue(T n) {
15      // もしキューが満杯ならば，サイズを2倍にしたキューを作成し，すべての要素をコピーする
16      if (fixed_queue->isFull()) {
```

```
17        int new_cap = fixed_queue->capacity() * 2 + 1;
18        FixedRingQueue<T> *new_fixed_queue = new FixedRingQueue<T>(new_cap);
19        while (!fixed_queue->isEmpty())
20          new_fixed_queue->enqueue(fixed_queue->dequeue());
21        delete fixed_queue;
22        fixed_queue = new_fixed_queue;
23      }
24      fixed_queue->enqueue(n);
25    }
26
27    T    dequeue()  { return fixed_queue->dequeue();  }
28    int  size()     { return fixed_queue->size();     }
29    int  capacity() { return fixed_queue->capacity(); }
30    bool isEmpty()  { return fixed_queue->isEmpty();  }
31
32    friend ostream& operator << (ostream& stream , const VarRingQueue<T>& q)
33    {
34      return stream << *(q.fixed_queue);
35    }
36  };
37
38  int main(void) {
39    VarRingQueue<int> q;     // サイズが自動的に伸張するキューを作成
40
41    // 動作確認のためのループ
42    for (int i=0; i < 6; i++) {
43      // キューに5つの要素をエンキュー
44      for (int n=1; n <= 5; n++) {
45        q.enqueue(n);
46        cout << "enqueue(" << n << ") " << q << endl;
47      }
48      // キューの要素数が i 個になるまでデキュー
49      while (q.size() > i)
50        cout << "dequeue = " << q.dequeue() << " " << q << endl;
51    }
52
53    return 0;
54  }
```

問 2.6 木に含まれる頂点数および木の深さを，再帰呼び出しを利用せずに求めるメンバ関数 getNumNodes() と getDepth() の定義例を以下に示す．これらのメンバ関数以外の定義はリスト **2.7** と同じであるため省略している．

メンバ関数 getNumNodes() では，リスト **2.5** で定義したスタッククラス VecStack を用いて木の各頂点を前順でなぞりながら頂点数を数えている．

メンバ関数 getDepth() も同様にスタックを利用して各頂点を前順でなぞる．ただし，頂点の深さを調べるため，スタックには頂点だけでなく，その深さも合わせて追加している（21, 26, 28 行目）．このためにクラステンプレート pair を利用している．pair は，2 つの異なる型の値を保持する “組” を表現するためのクラスである．pair オブジェクトを作るときは，make_pair(this, 0)のようにする（21 行目）．これは，根 (this) とその深さ (根の深さは 0 である) を組として作成している．pair オブジェクトの第 1 要素（この例では頂点）は pair のメンバ変数 first に，第 2 要素（頂点の深さ）は second に入っている．例えば 24 行目では，スタックから取り出した組 p に対し，その深さ p.second が，それまでに見つけた最大深さ depth よりも深いならば，depth を更新している．また右の子 p.first->right が存在するならば，スタックに右の子とその深さ p.second + 1 の組を積んでいる（26 行目）．左の子 p.first->left についても同様である（28 行目）

```
1  template <class T> class Node {
2      :
3  public:
4      :
5    int getNumNodes() {
6      int num = 0;                                  // 木の頂点数を表す変数
7      VecStack<Node<T> *> stack;                    // 木を前順でなぞるためのスタック
8      stack.push(this);                             // 根をスタックに追加
9      while (!stack.isEmpty()) {
10       Node<T> *p = stack.pop();                   // スタックから頂点を取り出す
11       num++;                                      // pop するたびにカウントを増やす
12       if (p->right != NULL) stack.push(p->right);  // 右の子をスタックに追加
13       if (p->left  != NULL) stack.push(p->left );  // 左の子をスタックに追加
14     }
15     return num;
16   }
17
18   int getDepth() {
19     int dep = 0;                                  // 木の深さを表す変数
20     VecStack<pair<Node<T> *, int> > stack;        // 木を前順でなぞるためのスタック
21     stack.push(make_pair(this, 0));               // 根とその深さをスタックに追加
22     while (!stack.isEmpty()) {
23       pair<Node<T> *, int> p = stack.pop();       // スタックから頂点とその深さを取り出す
24       if (dep < p.second) dep = p.second;         // 最大深さを更新
25       if (p.first->right != NULL)                 // 右の子とその深さをスタックに追加
26         stack.push(make_pair(p.first->right, p.second + 1));
27       if (p.first->left  != NULL)                 // 左の子とその深さをスタックに追加
28         stack.push(make_pair(p.first->left , p.second + 1));
29     }
30     return dep;
31   }
32 };
```

問 2.7　木に含まれる頂点数および木の深さを，再帰呼び出しを用いて求めるメンバ関数 `rgetNumNodes()` と `rgetDepth()` の定義例を以下に示す．これらのメンバ関数以外の定義はリスト **2.7** と同じであるため省略している．

まず関数 `rgetNumNodes()` を説明しよう．8 行目の `left->rgetNumNodes()` は，左の子供を根とする部分木の頂点数を求めている．10 行目の `right->rgetNumNodes()`も同様に，右の子供を根とする部分木の頂点数を求めている．その後，頂点自身を加算（11 行目の `+ 1`）して，その頂点を根とする部分木に含まれる頂点数を求めている．

次に関数 `rgetDepth()` について説明しよう．この関数が属するオブジェクトを頂点 x と表記することにしよう．20 行目の `left->rgetDepth()` は，x の左の子供を根とする部分木の深さを表す．これに 1 を加算したものが，x を根とする木における左の子孫の深さ `l_dep` になる．24 行目では，同様に x を根とする木における右の子孫の深さ `r_dep` を求めている．そして，`l_dep` と `r_dep` のうち大きなものが，x を根としたときの木の深さとなる（25 行目）．最後の関数 `max` は，同じ型の 2 つの値を受け取り，最大の値を返す関数テンプレートである．

```
1  template <class T> class Node {
2      :
3  public:
4      :
5    int rgetNumNodes() {
6      int num = 0;                          // 子孫頂点の数をあわらす変数
7      if (left  != NULL)
8        num += left ->rgetNumNodes();       // 左の子孫頂点の数を加算
```

```
    if (right != NULL)
      num += right->rgetNumNodes();     // 右の子孫頂点の数を加算
    return num + 1;                     // この頂点自身を足して返す
  }

  int rgetDepth() {
    int l_dep = 0;              // この頂点を根としたとき，右の子孫の最大深さを表す変数
    int r_dep = 0;              // この頂点を根としたとき，右の子孫の最大深さを表す変数
    if (left  != NULL)
      // まず，左の子を根としたときの深さを求める（left->rgetDepth()）
      // 次に，この頂点を根とすると 1 を加算する必要がある
      l_dep = left ->rgetDepth() + 1;
    if (right != NULL)
      // まず，右の子を根としたときの深さを求める（right->rgetDepth()）
      // 次に，この頂点を根とすると 1 を加算する必要がある
      r_dep = right->rgetDepth() + 1;
    return max(l_dep, r_dep);           // 左右の子孫で最大の深さを返す
  }
};
```

（3　章）

問 3.1　a, b をそれぞれ 0 より大きな 1 以外の数とし，n を任意の正の数とする．このとき $\log_a n$ と $\log_b n$ の違いを考えてみよう．対数の底の変換公式 ($\log_a n = \log_b n / \log_b a$) を用いれば

$$\frac{\log_a n}{\log_b n} = \frac{\log_b n}{\log_b a \log_b n} = \frac{1}{\log_b a}$$

が成り立つ．したがって任意の 2 つの底 a, b に対し，$\log_a n$ と $\log_b n$ との間には定数倍 ($1/\log_b a$) の違いしかないことが分かる．計算量の漸近的評価において定数倍の違いは無視されるため，O 表記では底を省略することができる．

問 3.2　書籍配列に対し，任意のキーを指定して二分探索を行う関数テンプレート `binary_search` の実装例を以下に示す．

まず書籍クラス `Book` に，書籍のタイトルと著者名，価格のそれぞれ対して統一的な操作を提供する内部クラス `TitleSelector`, `AuthorSelector`, `PriceSelector`を定義している（25,34,43 行目）．例えば内部クラスのメンバ関数 `get()(const Book& b)` は，書籍オブジェクト `b` のタイトル，著者名，価格のそれぞれを返す関数である（27,36,45 行目）．またメンバ関数 `bool operator()(const Book& left, const Book& right)` は，2 つの書籍オブジェクトを比較して大小判定を行うための関数であり，書籍配列をソートするために用意してある（二分探索は整列済の配列を前提としているため）（28,37,46 行目）．

内部クラスが提供する統一的なメンバ関数を前提として二分探索のプログラムを実装することにより，任意のキーを指定して探索することが可能となる．これらの内部クラスはキーの選択器であるともいえる．

関数 `binary_search` について説明しよう（54,69 行目）．第 2 引数の `keySelector`には，書籍オブジェクトから探索対象のキーを取得する選択器オブジェクト（前述の内部クラスのオブジェクト）を指定し，第 3 引数の `key` には探索したいキーの値を指定する．その使用例は `main` 関数を参照してほしい．例えば 89 行目では，著者名をキーとして，著者 “Grant” が執筆した書籍の有無を調べている．ここで `Book::AuthorSelector()` は書籍オブジェクトが持つ著者を取得するための選択器オブジェクトを生成している．

94 行目の `sort(books.begin(), books.end(), Book::PriceSelector())` では，書籍配列を価格の昇順でソートしている．関数 `sort` は STL が提供する整列用の関数テンプレートであり，ヘッダファイル `algorithm` をインクルードすることにより利用可能になる．この例では，書籍配列の先頭 `books.begin()` から末尾 `books.end()` までをソートする．第 3 引数の `Book::PriceSelector()` は，2 つの書籍オブジェクト間の大小関係を判定するオブジェクトである．具体的には `sort` 関数内部から 46 行目のメンバ関数 `bool operator()(const Book& left, const Book& right)` が 2 つの書籍オブジェクト `left`, `right` の価格を比較するために呼び出される．

この実装例では，キーごとに異なる関数 `getAuthor()` や `getPrice()` ではなく，同じ形式の関数 `get()` を用いることで（その代わり，関数 `get()` を提供する内部クラス `AuthorSelector` や `PriceSelector` を関数 `binary_search` の呼び出し時に指定する），汎用的な二分探索のプログラムを実現している．

```
1  #include <iostream>
2  #include <vector>
3  #include <string>
```

```
4  #include <algorithm>
5  using namespace std;
6
7  class Book {
8  private:
9    string author;
10   string title;
11   int    price;
12 public:
13   Book(string a, string t, int p) { author = a; title = t; price = p; }
14
15   string getAuthor() { return author; }
16   string getTitle()  { return title;  }
17   int    getPrice()  { return price;  }
18
19   friend ostream& operator << (ostream& stream , const Book& b) {
20     return stream << "(" << b.author << ","
21            << b.title << "," << b.price << "yen)";
22   }
23
24   // タイトルを取得するためのクラス
25   class TitleSelector {
26   public:
27     string get(const Book& b) { return b.title; }
28     bool operator()(const Book& left, const Book& right) const {
29       return left.title < right.title;
30     }
31   };
32
33   // 著者を取得するためのクラス
34   class AuthorSelector {
35   public:
36     string get(const Book& b) { return b.author; }
37     bool operator()(const Book& left, const Book& right) const {
38       return left.author < right.author;
39     }
40   };
41
42   // 価格を取得するためのクラス
43   class PriceSelector {
44   public:
45     int get(const Book& b) { return b.price; }
46     bool operator()(const Book& left, const Book& right) const {
47       return left.price < right.price;
48     }
49   };
50 };
51
52 // 書籍配列に対して任意のキーで２分探索を行う関数テンプレート（探索範囲が指定可能）
53 template <class KeySelector, class KeyType>
54 int binary_search(vector<Book>& books, KeySelector keySelector, KeyType key,
55           int low, int high) {
56   if (low > high) return -1;
57
58   int mid = (low + high) / 2;
59   if (keySelector.get(books[mid]) == key)
60     return mid;
61   if (keySelector.get(books[mid]) <  key)
62     return binary_search(books, keySelector, key, mid + 1, high   );
63   else
64     return binary_search(books, keySelector, key, low    , mid - 1);
65 }
66
67 // 書籍配列に対して任意のキーで２分探索を行う関数テンプレート（探索範囲は配列全体）
```

```
68  template <class KeySelector, class KeyType>
69  int binary_search(vector<Book>& books, KeySelector keySelector, KeyType key)
70  {
71    return binary_search(books, keySelector, key, 0, books.size() - 1);
72  }
73
74  void init_books(vector<Book>& books) {
75    books.push_back(Book("Alice", "C++",  1000));
76    books.push_back(Book("Bob",   "C++",  1500));
77    books.push_back(Book("Carol", "C",    1250));
78    books.push_back(Book("Dave",  "Java", 1000));
79    books.push_back(Book("Eve",   "C++",  2000));
80    books.push_back(Book("Frank", "Ruby", 1500));
81    books.push_back(Book("Grant", "C#",   1000));
82  }
83
84  int main(void) {
85    vector<Book> books;
86    init_books(books);     // 書籍配列は著者名の昇順にソート済
87
88    // 著者名による２分探索の例
89    int idx = binary_search(books, Book::AuthorSelector(), "Grant");
90    if (idx >= 0) cout << books[idx]  << endl;
91    else          cout << "NOT FOUND" << endl;
92
93    // 書籍配列を価格の昇順にソート
94    sort(books.begin(), books.end(), Book::PriceSelector());
95
96    // 価格による２分探索の例
97    idx = binary_search(books, Book::PriceSelector(), 1250);
98    if (idx >= 0) cout << books[idx]  << endl;
99    else          cout << "NOT FOUND" << endl;
100
101   // 書籍配列をタイトルの昇順にソート
102   sort(books.begin(), books.end(), Book::TitleSelector());
103
104   // タイトルによる２分探索の例
105   idx = binary_search(books, Book::TitleSelector(), "COBOL");
106   if (idx >= 0) cout << books[idx]  << endl;
107   else          cout << "NOT FOUND" << endl;
108
109   return 0;
110 }
```

問 3.3　再帰呼び出しを用いずに二分探索を行う関数テンプレート binary_search の実装例を以下に示す．問 **3.2** の解答例と同様に，任意のキーを指定できるように一般化してある．以下のプログラムは **3.3** 節「再帰的探索」で示した 2 分探索のアルゴリズム（p.49）をそのまま反映したものになっている．

```
1   // 書籍配列に対して任意のキーで２分探索を行う関数テンプレート（探索範囲が指定可能）
2   template <class KeySelector, class KeyType>
3   int binary_search(vector<Book>& books, KeySelector keySelector, KeyType key,
4                     int low, int high) {
5     while (true) {
6       if (low > high) return -1;
7       int mid = (low + high) / 2;
8       if (keySelector.get(books[mid]) == key)
9         return mid;
10      if (keySelector.get(books[mid]) <  key)
11        low = mid + 1;
12      else
13        high = mid - 1;
```

```
14   }
15 }
16
17 // 書籍配列に対して任意のキーで２分探索を行う関数テンプレート（探索範囲は配列全体）
18 template <class KeySelector, class KeyType>
19 int binary_search(vector<Book>& books, KeySelector keySelector, KeyType key)
20 {
21   return binary_search(books, keySelector, key, 0, books.size() - 1);
22 }
```

問 3.4 ある型 T の配列を特定のキーに基づきクイックソートで整列する関数テンプレート quick_sort の実装例を以下に示す．リスト **3.6** との違いは以下のとおりである．

- 第 1 引数を vector<int>& a から vector<T>& a に変更し，任意の型 T の配列を受け取るようにしている．
- 第 2 引数 Comparator lessThan を追加している．この lessThan には，2 つの T 型のオブジェクトの大小関係を判定する関数もしくはオブジェクトを指定する．
- 9,11 行目において，2 つの T 型オブジェクトの大小関係を判定するため，第 2 引数で指定した lessThan を呼び出している．

その他の部分はリスト **3.6** と同一である．

この関数テンプレート quick_sort の使用例は main 関数を参照してほしい．この main 関数は，問 **3.2** の解答例で示した書籍クラス Book と書籍配列を初期化する関数 init_books を前提としている．main 関数では，まず関数 init_books により書籍配列を生成し，その並びを出力したあと，価格，著者名，タイトルのそれぞれでソートした結果を出力している．例えば価格でソートする場合は，quick_sort(books, Book::PriceSelector()) のようにする（49 行目）．第 2 引数の Book::PriceSelector() は，書籍クラス Book の内部クラスであり，このクラスは 2 つの書籍の価格を比較するメンバ関数 bool operator() (const Book& left, const Book& right) を持っている．この関数は，2 つの書籍の価格を比較するために 9,11 行目において呼び出される．

```
1  template <class T, class Comparator>
2  int partition(vector<T>& a, Comparator lessThan, int begin, int end) {
3    int pivot_idx = begin;                    // ピボットの位置（配列上の添字）
4    T   pivot     = a[pivot_idx];             // ピボットの値
5    int left_idx  = begin + 1;
6    int right_idx = end;
7    while (true) {
8      // 右部分列においてピボット以下の要素を探す
9      while (lessThan(pivot, a[right_idx])) right_idx--;
10     // 左部分列においてピボット以上の要素を探す
11     while (left_idx <= right_idx && lessThan(a[left_idx], pivot)) left_idx++;
12     if (left_idx >= right_idx) break;        // 左右の部分列が交差したら分割完了
13     swap(a[left_idx++], a[right_idx--]);     // 左右の要素を交換
14   }
15   // ピボットを左部分列の右端要素と交換
16   pivot_idx = left_idx - 1;
17   swap(a[begin], a[pivot_idx]);
18   return pivot_idx;                         // ピボットの位置を戻り値とする
19 }
20
21 template <class T, class Comparator>
22 void quick_sort(vector<T>& a, Comparator lessThan, int begin, int end) {
23   // 整列すべき要素数が 1 なら整列の必要なし
24   if (begin >= end) return;
25   // ピボットを中心に左右に分割
```

```
26    int pivot_idx = partition(a, lessThan, begin, end);
27    // 左部分列に対して再帰的に quick_sort を適用
28    quick_sort(a, lessThan, begin, pivot_idx - 1);
29    // 右部分列に対して再帰的に quick_sort を適用
30    quick_sort(a, lessThan, pivot_idx + 1, end);
31  }
32
33  template <class T, class Comparator>
34  void quick_sort(vector<T>& a, Comparator lessThan) {
35    quick_sort(a, lessThan, 0, a.size() - 1);            // 配列全体を整列する
36  }
37
38  // 使用例（クラス Book，書籍配列の初期化関数 init_books は省略）
39  int main(void) {
40    vector<Book> books;
41    init_books(books);
42
43    cout << "Original Book array:" << endl;
44    for (unsigned int i=0; i < books.size(); i++)
45      cout << books[i] << endl;
46    cout << endl;
47
48    cout << "Sorted by price:" << endl;
49    quick_sort(books, Book::PriceSelector());
50    for (unsigned int i=0; i < books.size(); i++)
51      cout << books[i] << endl;
52    cout << endl;
53
54    cout << "Sorted by author:" << endl;
55    quick_sort(books, Book::AuthorSelector());
56    for (unsigned int i=0; i < books.size(); i++)
57      cout << books[i] << endl;
58    cout << endl;
59
60    cout << "Sorted by title:" << endl;
61    quick_sort(books, Book::TitleSelector());
62    for (unsigned int i=0; i < books.size(); i++)
63      cout << books[i] << endl;
64    cout << endl;
65
66    return 0;
67  }
```

上の実装例では，関数 quick_sort の第 2 引数に，2 つの書籍を比較するためのオブジェクトを渡しているが，関数を指定することもできる．例えば次の関数 bookLessThan は，2 つの書籍 left, right に対し，価格が異なるならば left がより安いかどうかを返し，価格が等しい場合は，left の著者名が辞書式順序でより小さいかどうかを返している．この関数を利用してソートするには quick_sort(books, bookLessThan) のようにする．これにより価格が等しい場合は著者名順でソートすることが可能になる．

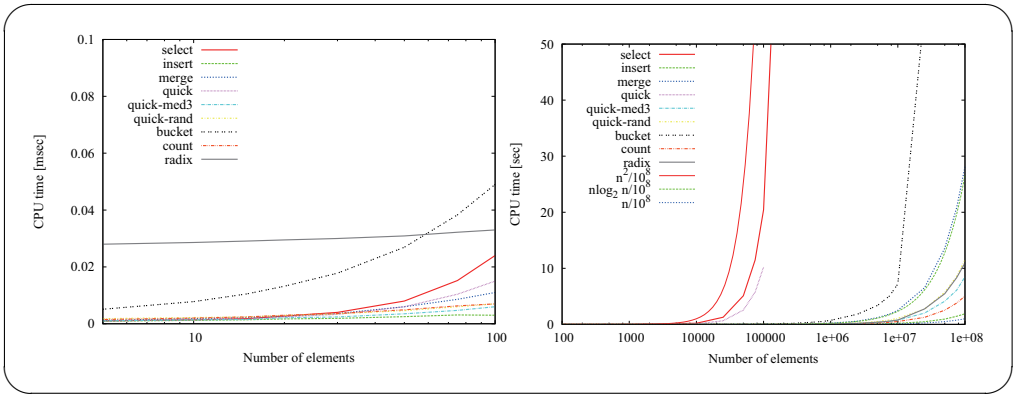
```
1  bool bookLessThan(Book& left, Book& right) {
2    if (left.getPrice() != right.getPrice())
3      return left.getPrice() < right.getPrice();
4    else
5      return left.getTitle() < right.getTitle();
6  }
```

問 3.5 **解図 3.1**, **解図 3.2**, **解図 3.3** は，3 章で紹介した整列アルゴリズムの実行時間の比較結果である．それぞれ，昇順に整列済み，降順に整列済み，ランダムな並びの数列を，昇順に整列するのに要した時間を示している．横軸は数列の長さを表す対数軸であり，縦軸は実行

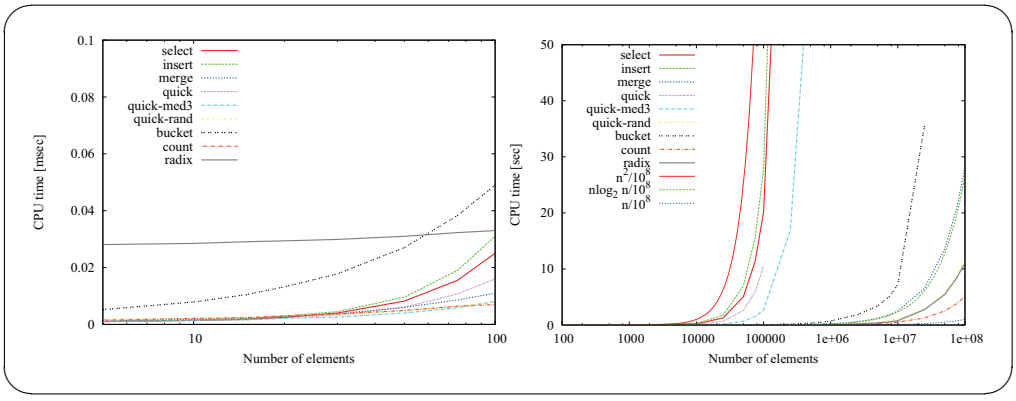
時間を表している．ランダムな数列は 0 から 1000 万までの数から構成されており，昇順に整列済の数列は $1, 2, 3, \cdots$ のような形式をしており，降順はその逆である．数列の長さが短いと整列アルゴリズムの実行時間はごく僅かとなり計測が難しい．そこで複数回実行して，その平均時間を求めている． また参考として n^2, $n \log_2 n$, n の曲線も載せてある（それぞれ $1/10^8$ を乗じてある）．実験環境は g++ 4.8.5, CentOS 7.3, Intel Core i5-6600 (3.30GHz), 16GB RAM である．

- 選択ソート（図中の select）と挿入ソート（insert）は，数列の長さが 30 程度までであれば，マージソートやクイックソートなどの高速なアルゴリズムと遜色ないが，それ以降は n^2 に比例した時間がかかるようになることが分かる．ただし挿入ソートは昇順に整列済みの数列に対して最も高速であり，その実行時間は n に比例している．
- 図中の quick は 3 章で示したクイックソートの実装である．これをもとにピボットの選択方法を工夫したものが quick-med3 と quick-rand であり，それぞれ，数列中の先頭，中央，末尾の 3 要素の中央値をピボットとして選択する手法，ランダムにピボットを選択する手法である．その実装例は問 **3.6** の解答例を参照してほしい．

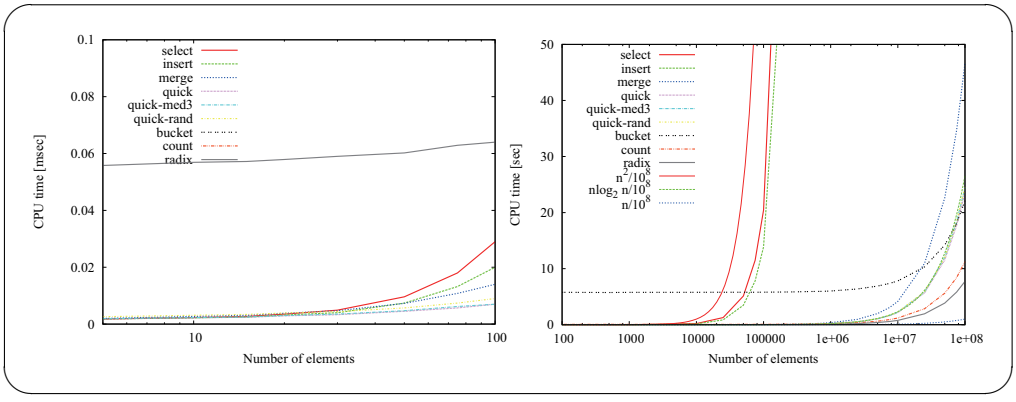
  quick は先頭要素をピボットとして選択するため，昇順および降順に整列済みのデータでは，常に最小値，最大値をとることになる．これはクイックソートにとって最悪のケースであり，このため選択・挿入ソートと同様の性能となる．また再帰呼び出しが多数繰り返されるため，長さが 10 万以降の数列ではメモリ不足となり整列できなかった．quick-med3 は，昇順データにおいて上記の欠点が解消されているが，降順データでは選択・挿入ソートと同様の性能である．quick-med3 では，その実装上の理由から，数列の先頭，中央，末尾の 3 要素のうち 2 つが，数列中の最大要素と，その 1 つ手前の要素になることがしばしばある．すなわちピボットがほぼ最大の要素となるため，分割があまり進まず性能が低下している．quick-rand は，いずれの数列に対しても高速であり，その実行時間はほぼ $n \log_2 n$ に比例している．
- マージソート (merge) は，クイックソートよりも遅いものの，quick, quick-med3 のような苦手な数列がないことが分かる．その実行時間はほぼ $n \log_2 n$ に比例している．
- バケットソート（bucket）では，要素の種類ごとにバケット（キュー）を用意する必要がある．この実験では，STL のクラステンプレート `queue` を利用している．バケットソートは， 計算量上はクイックやマージソートよりも良いアルゴリズムではあるが，`queue` オブジェクトの構築や破棄に時間がかかるため，この実験ではクイックソートよりも遅い結果となっている．また昇順・降順の数列では，すべての要素が異なるため，数列の長さに等しい数のバケットが必要となる．数列の長さが 1000 万を超えると，メモリ不足となり整列できなかった．ランダムな並びでは，数列の長さにかかわらず，常に 1000 万個のキューを用意する必要があるため，それに 6 秒程度の時間を要している．
- 計数ソート（count）と基数ソート（radix）は，いずれも高速な結果を示している．基数ソートでは 4096 を基数として用いた．これは 0 〜 1000 万の数値であれば，桁ごとの整列を 2 回実行すれば済むためである（$4096^2 \approx 1677$ 万）．ランダムな並びのデータにおいては基数ソートが最も高速であった．その実行時間は n に比例しているとまではいえないものの，増加傾向はクイックソートに比べて緩やかである．た



解図 3.1　昇順数列に対する整列アルゴリズムの比較



解図 3.2　降順数列に対する整列アルゴリズムの比較



解図 3.3　ランダムな数列に対する整列アルゴリズムの比較

だし計数および基数ソートでは，要素の出現頻度を数え上げるために，計数ソートでは大きさ 1000 万の配列を，基数ソートでは大きさ 4096 の配列を確保し，それを走査している．このため数列の長さが短い場合でも，計数ソートは 0.6 秒程度（解図 3.3 の左側のグラフでは範囲外となり描画されていない），基数ソートは 0.5 ミリ秒程度の時間を要している．

問 3.6 数列中の先頭，中央，末尾の 3 要素の中央値をピボットとして選択する手法（問 **3.5** の解答例で示した quick-med3）と，ランダムにピボットを選択する手法（quick-rand）の 2 つを紹介する．それぞれ，もとのクイックソートの実装（リスト **3.6**）との違いは数列を分割する関数 `partition` のみである．

まず quick-med3 の分割関数 `partition_med3` を以下に示す．2–10 行目までが追加した部分であり，それ以外に変更はない．この実装例では，先頭，中央，末尾の 3 要素の中央値を先頭要素と入れ替え，それをピボットとしている．

```
1  int partition_med3(vector<int>& a, int begin, int end) {
2    int x = a[begin];                  // 先頭要素
3    int y = a[(begin + end) / 2];      // 中央要素
4    int z = a[end];                    // 末尾要素
5    // もし 3 点の中央値が真ん中の要素であれば先頭と入れ替え
6    if ((x <= y && y <= z) || (z <= y && y <= x))
7      swap(a[begin], a[(begin + end) / 2]);
8    // もし 3 点の中央値が末尾要素であれば先頭と入れ替え
9    else if ((x <= z && z <= y) || (y <= z && z <= x))
10     swap(a[begin], a[end]);
11
12   int pivot_idx = begin;             // ピボットの添字（先頭要素）
13   int pivot     = a[pivot_idx];      // ピボット
14
15     ⋮          // 以下，partition と同様
16 }
```

次に quick-rand の分割関数 `partition_rand` を以下に示す．3,4 行目が追加した部分であり，それ以外に変更はない．この実装例では，分割範囲からランダムに選んだ要素を先頭要素と入れ替え，それをピボットとしている．

```
1  int partition_rand(vector<int>& a, int begin, int end) {
2    // ランダムに選んだ要素と先頭を入れ替え
3    int r = begin + rand() % (end - begin + 1);
4    swap(a[begin], a[r]);
5
6    int pivot_idx = begin;             // ピボットの添字（先頭要素）
7    int pivot     = a[pivot_idx];      // ピボット
8
9      ⋮          // 以下，partition と同様
10 }
```

問 3.7 マージソートやクイックソートでは，与えられた数列を分割しながら整列するが，分割後の数列の長さが短くなった場合に挿入ソートに切り替える実装例を示そう．そのために，まず指定された範囲を挿入ソートにより整列する関数 `void insertion_sort(vector<int>& a, int begin, int end)` を以下に示す（リスト **3.4** では配列全体を整列対象としていたが，整列範囲を引数で指定できるようにしたもの）．この関数は，配列 `a` の添字 `begin` から `end` までを挿入ソートにより整列する．

```
1  void insertion_sort(vector<int>& a, int begin, int end) {
2    int last_idx = end;
3    for (int i = begin + 1; i <= last_idx; i++) {
4      int v = a[i];
5      int j = i;
6      for (; j >= begin + 1 && a[j-1] > v; j--)
7        a[j] = a[j-1];
8      a[j] = v;
9    }
10 }
```

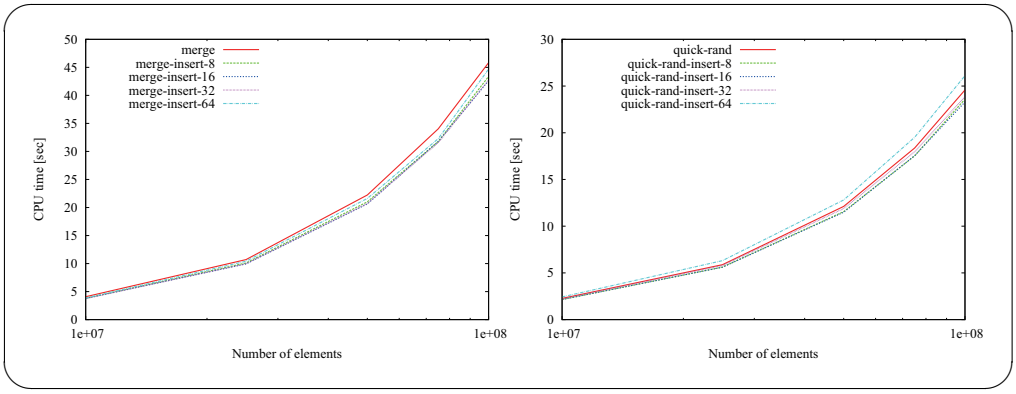
数列の長さが短くなった場合に挿入ソートに切り替えるマージソートの実装例を示そう．もとのマージソートの実装（リスト **3.5**）との違いは，4 引数の `merge_sort` 関数のみであり，その定義を以下に示す．4–7 行目が新たに追加した部分である．もし整列対象の数列の長さ `end - begin` が，閾値 `threashold` 以下になった場合，挿入ソートによりそれを整列している．

```
1  void merge_sort(vector<int>& a, vector<int>& tmp, int begin, int end) {
2    if (begin >= end) return;          // 整列すべき要素数が 1 なら整列の必要なし
3    // 範囲の大きさが指定値以下ならば，挿入ソートに切り替える
4    if (end - begin <= threashold) {
5      insertion_sort(a, begin, end);
6      return;
7    }
8    int mid = (begin + end) / 2;       // 中央位置を求める
9    merge_sort(a, tmp, begin, mid);    // 左部分列に対して再帰的に merge_sort を適用
10   merge_sort(a, tmp, mid+1, end);    // 右部分列に対して再帰的に merge_sort を適用
11   merge(a, tmp, begin, mid, end);    // 整列済みの 2 つの部分列を併合
12 }
```

次に，クイックソートにおける実装例を示そう．もとのクイックソートの実装（リスト **3.6**）との違いは，3 引数の `quick_sort` 関数のみであり，その定義を以下に示す．4–7 行目が新たに追加した部分であり，マージソートの場合と同様に，整列対象の範囲が閾値以下になった場合に挿入ソートに切り替えている．

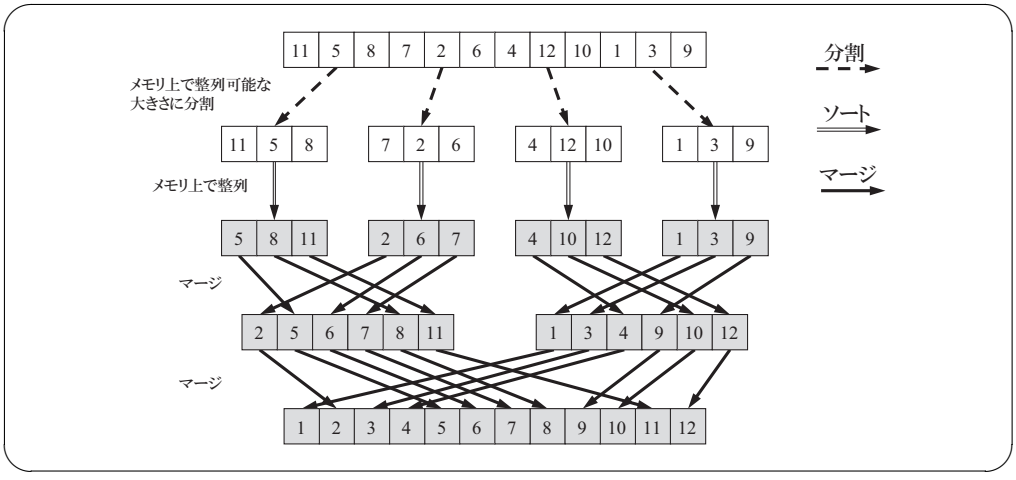
```
1  void quick_sort(vector<int>& a, int begin, int end) {
2    if (begin >= end) return;
3    // 範囲の大きさが指定値以下ならば，挿入ソートに切り替える
4    if (end - begin <= threashold) {
5      insertion_sort(a, begin, end);
6      return;
7    }
8    int pivot_idx = partition_rand(a, begin, end);
9    // 左部分列に対して再帰的に quick_sort を適用
10   quick_sort_rand_insert_aux(a, begin, pivot_idx - 1);
11   // 右部分列に対して再帰的に quick_sort を適用
12   quick_sort_rand_insert_aux(a, pivot_idx + 1, end);
13 }
```

解図 3.4 は，ランダムな並びの数列をこれらのプログラムにより整列したときの実行時間の比較結果である．実験環境は問 **3.5** の解答例と同じである．図中の merge-insert-n は，数列の長さが n 以下になったときに挿入ソートに切り替えるマージソートの結果を表す．quick-rand-insert-n についても同様である．この実験結果から，マージソート，クイックソートともに，数列の長さが 16 以下になったときに挿入ソートに切り替えると，実行時間が若干改善されることが分かる．



解図 3.4　数列の長さに応じて挿入ソートに切り替えるマージソート（左）とクイックソート（右）の評価

問 3.8　メモリに読み込むことができない巨大なデータ系列をハードディスクなどの補助記憶装置を利用して整列する手法を示そう．整列のための基本方針はマージソートと同様である．ただしマージソートでは，与えられた系列を大きさが 1 になるまで再帰的に等分割するが，ここではメモリ上で整列可能な大きさに分割する．その概要を**解図 3.5** に示す．



解図 3.5　補助記憶装置を利用した整列アルゴリズムの概要

いま巨大なデータ系列 X が補助記憶装置上のファイルとして与えられているとする．まず X をメモリ上で整列可能な大きさに分割する．分割後の部分系列群を $X_1, X_2, \cdots, X_n$ としよう（図では長さ 3 の系列 4 本に分割している）．部分系列 X_i をメモリに読み込み，適当なソートアルゴリズムによりソートし，整列後の系列 X'_i をファイルに書き出す．この操作を各部分系列に対して適用する．実際には，X から一定個数の要素をメモリに読み込み，それらを整列後，別のファイルとして書き出す操作を X の要素がなくなるまで繰り返せばよい．

いま整列済みの部分系列 $X'_1, X'_2, \cdots, X'_n$ がそれぞれ異なるファイルに保存されているとしよう．以降はマージソートにおけるマージ操作と同様に，整列済みの 2 本の部分系列をマージする操作を再帰的に繰り返せばよい．ただしマージ後の系列はファイルに保存するものとする．例として整列済みの 2 本の系列 Y と Z をマージする場合を考えよう．Y と Z はそれぞれ異なるファイルに保存されている．マージにあたって Y と Z のすべての要素をメモリに読み込む必要はなく，Y と Z の先頭からそれぞれ 1 つずつ要素を取り出せば十分である．もし昇順に整列するならば，いずれか小さい方を別のファイルに書き出す．例えば Y の先頭要素が小さければそれを書き出し，2 番目の要素を Y から取り出す．これを Y と Z の要素がなくなるまで繰り返せばよい．

問 3.9 整列における配列上でのデータの移動コストを抑えるため，各要素へのポインタからなる配列を整列した後に，ポインタの並び順にデータを整列するプログラムの実装例を示そう．ポインタ配列の整列は，任意の型の配列を整列可能な汎用的な関数があると簡単に実現できる．ここでは問 **3.4** の解答例で示した任意の型の配列を特定のキーに基づきクイックソートで整列する関数テンプレート `quick_sort` を利用する例を示そう．

まず巨大なデータの例として，以下に示すクラス `BigData` を用いる．`BigData` は 1024 個の整数（大きさ 1023 の整数配列と大小比較用の整数 1 つ）から構成されるクラスである．`int` 型の大きさは環境によって異なるが，もし 4 バイトであると仮定すると，`BigData` オブジェクトの大きさは 4 キロバイトになる（4×1024 バイト）．

```
1  // 巨大なデータ
2  class BigData {
3  private:
4    int data[1023];            // 大きなサイズのデータにするため配列を持たせている
5    int num;                   // 整数
6  public:
7    BigData(int n) { num = n; }
8    int getNum() const { return num; }
9  };
10
11 // BigData オブジェクトの大小関係の判定関数
12 bool bigDataLessThan(const BigData& left, const BigData& right) {
13   return left.getNum() < right.getNum();     // オブジェクトが保持する整数により判定
14 }
15
16 // BigData オブジェクトの大小関係の判定関数（ポインタ版）
17 bool bigDataPointerLessThan(const BigData* left, const BigData* right) {
18   return left->getNum() < right->getNum();  // オブジェクトが保持する整数により判定
19 }
```

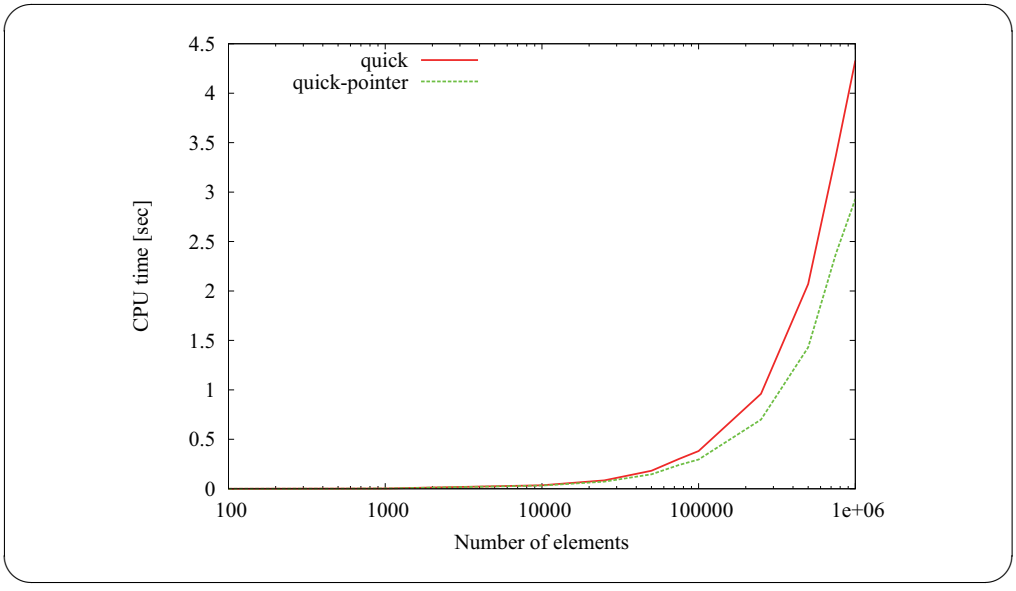
関数 `bigDataLessThan`（12 行目）は，2 つの `BigData` オブジェクトを受け取り，その大小関係を判定する．ここでは，それぞれの `BigData` オブジェクトが保持している整数 `num` の大小関係にもとづき判定している．関数 `bigDataPointerLessThan`（17 行目）は，そのポインタ版である．2 つの `BigData` オブジェクトへのポインタを受け取り，その大小関係を判定している．これら 2 つの関数は，`BigData` 型配列と `BigData` 型のポインタ配列における整列の速度差を比較するために利用する．

次に，オブジェクトの配列に対し，各要素へのポインタからなる配列を作成して，そのポインタ配列をクイックソートにより整列する関数 `quick_sort_by_pointer` の実装例を以下に示す．第 1 引数の `a` は整列対象のデータ配列（ポインタ配列ではない）であり，第 2 引数の `lessThan` には `bigDataPointerLessThan` のようなポインタを通してオブジェク

トの大小関係を判定する関数を指定する．まず整列対象の配列 a の内容を配列 b に移し（4 行目），次に b 上の各データへのポインタからなる配列 p を作成している（6–8 行目）．その後，問 **3.4** の解答例で示した関数 quick_sort を呼び出して，ポインタ配列 p を整列している（10 行目）．この整列では b 上のデータはまったく移動しておらず，各データを指すポインタのみが，そのポインタが指すデータの昇順に p 上で並んでいる．最後に，p 上の順序に従って，b 上の各データを a に書き戻している（12–14 行目）．

```
1  template <class T, class Comparator>
2  void quick_sort_by_pointer(vector<T>& a, Comparator lessThan) {
3    // 配列 a の内容を配列 b に移す
4    vector<T> b(a);
5    // 配列 b 上のデータのポインタ配列を作成
6    vector<T*> p;
7    for (int i=0; i < b.size(); i++)
8      p.push_back(&b[i]);
9    // ポインタ配列をソート
10   quick_sort(p, lessThan);
11   // 結果を配列 a に書き戻す
12   a.clear();
13   for (int i=0; i < p.size(); i++)
14     a.push_back(*p[i]);
15 }
```

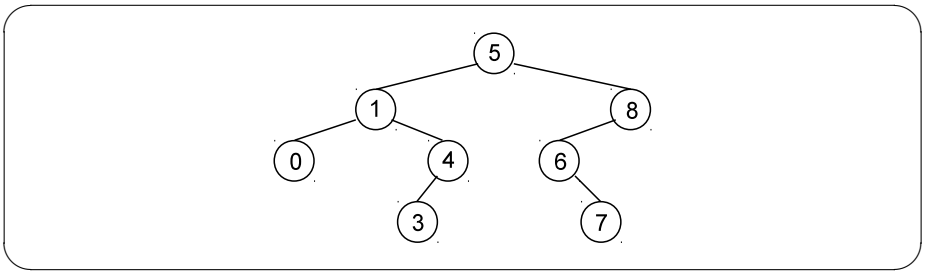
BigData 型のデータ配列 a に対し，上で示したように，ポインタを経由して整列する場合は quick_sort_by_pointer(a, bigDataPointerLessThan) のように指定する．逆にこれまでとおり a 上のデータを移動させながら整列する場合は quick_sort(a, bigDataLessThan) のようにすればよい．**解図 3.6** は，ランダムな並びの数列の整列にかかる時間の評価結果である．実験環境は問 **3.5** の解答例と同じである．図中の quick-pointer が，ポインタを経由して整列する手法の実行時間を示しており，データを何度も移動させる quick よりも高速であることが分かる．



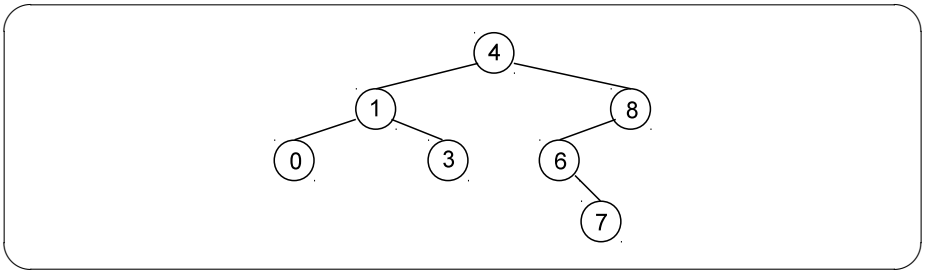
解図 3.6　ポインタ配列におけるクイックソートの性能評価

（4　章）

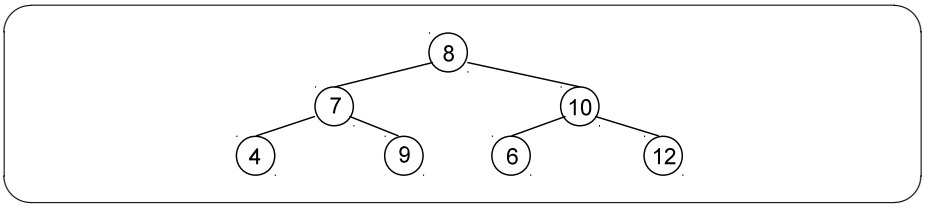
問 4.1



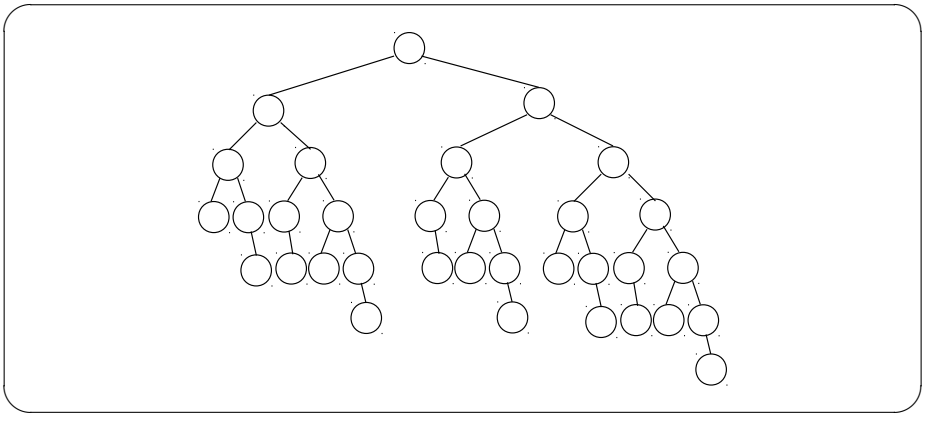
問 4.2　本書で紹介した削除方法には 2 通りの可能性があるが，以下はその一方の結果である．



問 4.3　以下の例では，左の子は常に親より小さく，右の子は常に親より大きいが，9 と 6 の配置は二分探索木における配置として正しくない．



問 4.4　一例は以下のとおり．

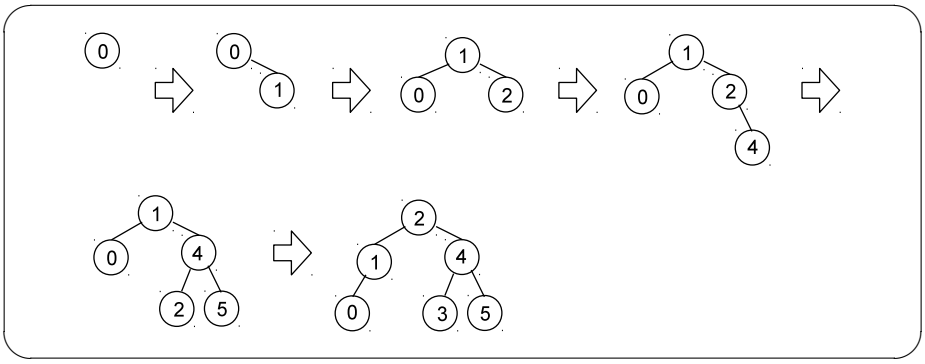


問 4.5　AVL 木の高さとそれに含まれる頂点数の最小値は以下の表のとおりとなる．

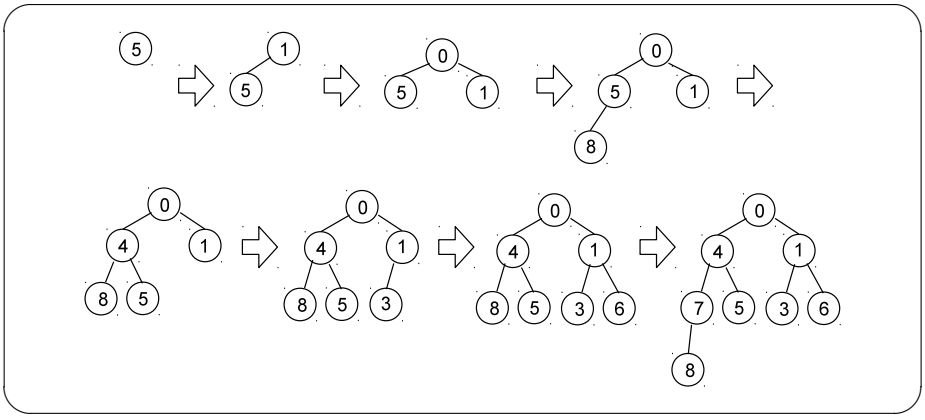
AVL 木の高さ	0	1	2	3	4	5	6
頂点数の最小値	1	2	4	7	12	20	30

頂点数が 30 の AVL 木の高さは 6 になることは出来ず，高さの上限は 5 である．

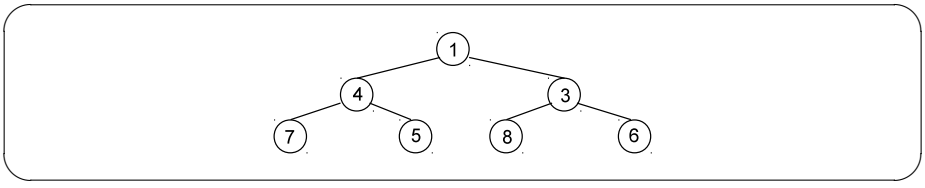
問 4.6



問 4.7



問 4.8



問 4.9 例として以下のようなテストプログラムによって誤りを確認できる．

```
1  int main( )
2  {
3    PriorityQueue pq(5);
4
5    pq.Enqueue(0);
6    pq.Enqueue(2);
7    pq.Enqueue(1);
8    pq.DequeueMin();
9    cout << pq.Min() << endl; // 正しくは1が出力されるべき.
10 }
```

問 4.10 17 行目を先に実行することより，dが更新され (小さくなり)，18,19 行目の実行を省略できる可能性が高くなる．順序を逆にすると実行速度が遅くなると予想される．

（5　章）

問 5.1　例として以下のようなテストプログラムによって誤りを確認できる．

```
class OpenAddrHash {
...
   // 例として key は整数，table_size による剰余をハッシュ値とする．
   int hash( int key ) const
         { return key % table_size; };

   // 簡単のため，線形走査を用いる．
   int alt_addr( int h, int collision ) const
         { return ( h + 1 ) % table_size; };
...
};

int main( )
{
  OpenAddrHash hash_table(10);

  hash_table.Insert(  1, 値 );
  hash_table.Insert( 11, 値 );
  hash_table.Erase( 1 );
  cout << hash_table.Find( 11, 値を受け取る変数 ) << endl;;
  // 正しくは true が返されるべき．
}
```

問 5.2　それぞれのハッシュ関数について，取りうる値の個数，値の出現確率，衝突確率の低い順は以下のとおりとなる．

ハッシュ関数	取りうる値の数	ハッシュ値の出現確率	衝突確率の低い順
$100 * x$	10	一様に 1/10	4
$x + 10 * y$	100	一様に 1/100	1
$(x + 10 * y)\%50$	50	一様に 1/50	2
$(x + y)\%1000$	19	1/100 〜 1/10	3
0	1	1	6
$x\%5$	5	一様に 1/5	5

問 5.3　hash_Eでは各文字コードの積を取り，2^{15} で割った余りをハッシュ値としているため，各文字コードに含まれる 2 のべき乗の因数はそのままハッシュ値の因数となる．十分大きな整数の素因数に含まれる 2 の個数の期待値は 1 であり，アルファベットに対応する ASCII コードについても，その素因数に含まれる 2 の個数の平均値はおよそ 0.9 である．そのため，n 文字のアルファベットからなるキーのハッシュ値は 2^n の倍数に限定されてしまう確率が高く，このようなハッシュ関数はハッシュ表の効率を悪くする．

（6　章）

問 6.1　省略.

問 6.2　以下では，木 T の頂点数を $|T|$ で表し，高さを $h(T)$ と表す．初めに帰納の基底を証明する．高さ $h(T) = 0$ の木 T は，Alg. 6.3 の INITIALIZE 関数で作成され，明らかに $|T| = 1 = 2^0 \geqq 2^{h(T)}$ となるので，基底が証明された．次に帰納ステップを証明する．まず帰納の仮定，即ち，UNION 関数の 2 つの引数 v_i と v_j が表す木 T_i と T_j に関して，$2^{h(T_i)} \leqq |T_i|$ と $2^{h(T_j)} \leqq |T_j|$ が成り立つと仮定する．UNION 関数は T_i と T_j を連結してより大きな木 $\widetilde{T}$ を生成するが，その計算は T_i と T_j の高さに応じて 3 つに場合分けされる．その場合分けに従って帰納ステップの証明を行う．

(1)　$h(T_i) > h(T_j)$ の場合，UNION 関数は T_j を T_i の根頂点の直下につなげて $\widetilde{T}$ を作り，高さは $h(\widetilde{T}) = h(T_i)$ となる．このとき以下が成り立つ．

$$\begin{aligned} |\widetilde{T}| &= |T_i| + |T_j| \\ &\geqq 2^{h(T_i)} + 2^{h(T_j)} \quad \text{(帰納の仮定より)} \\ &> 2^{h(T_i)} = 2^{h(\widetilde{T})} \end{aligned}$$

(2)　$h(T_i) = h(T_j)$ の場合，UNION 関数は T_j を T_i の根頂点の直下につなげて，高さ $h(\widetilde{T}) = h(T_i) + 1$ の木 $\widetilde{T}$ を生成する．このとき以下が成り立つ．

$$\begin{aligned} |\widetilde{T}| &= |T_i| + |T_j| \\ &\geqq 2^{h(T_i)} + 2^{h(T_j)} \quad \text{(帰納の仮定より)} \\ &= 2^{h(T_i)} + 2^{h(T_i)} = 2^{h(T_i)+1} = 2^{h(\widetilde{T})} \end{aligned}$$

(3)　$h(T_i) < h(T_j)$ の場合，UNION 関数は T_i を T_j の根頂点の直下につなげて，高さ $h(\widetilde{T}) = h(T_j)$ の $\widetilde{T}$ を生成する．この場合も (1) と同様に $|\widetilde{T}| \geqq 2^{h(\widetilde{T})}$ が成り立つ．

以上で帰納ステップの証明が終了した．

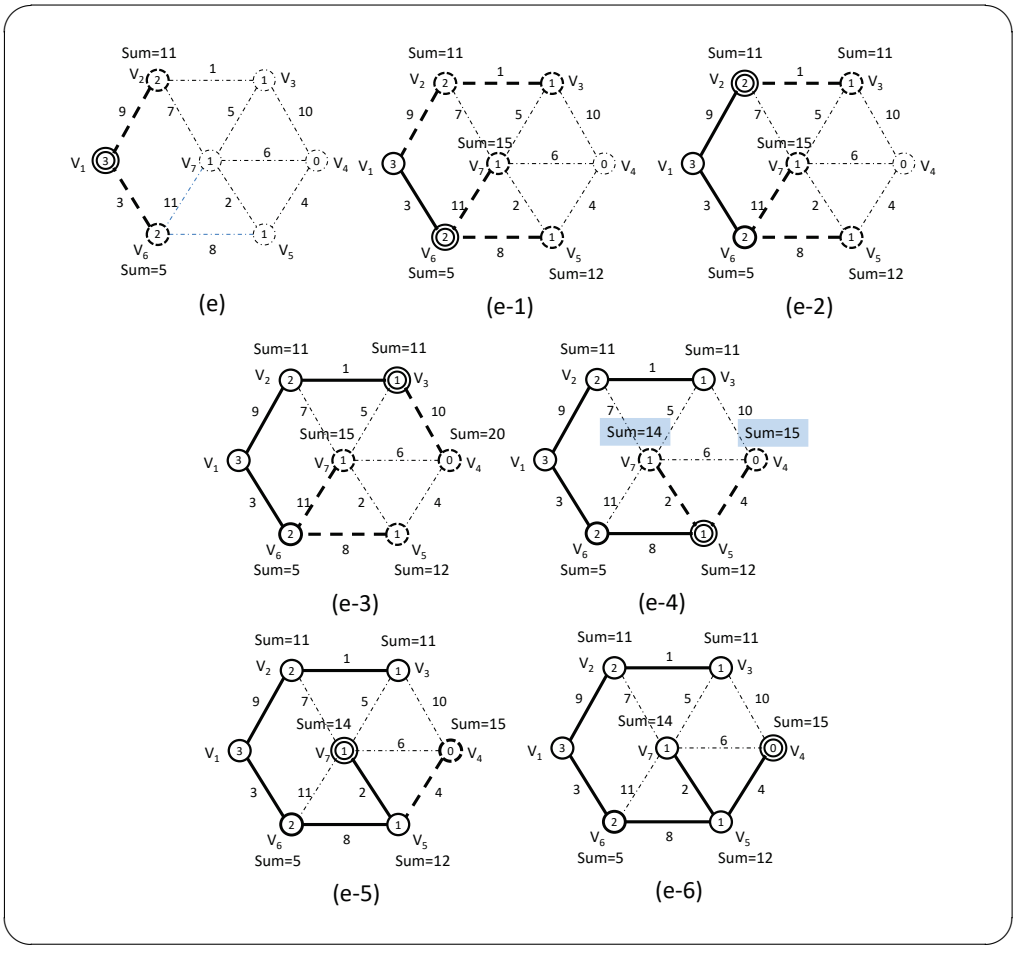
問 6.3　経路の圧縮を行う FIND 手続きの改良版コードの例を以下に示す．

```
function FindFlatting(v_i: 頂点)
    スタック SFP の初期化 ;                    ▷ SFP には圧縮する経路上の頂点を記録する
    tmp ← v_i
    while Sets[tmp] ≠ tmp do
        Push(SFP, tmp) ;                       ▷ 経路上の頂点 tmp を SFP に記録する
        tmp ← Sets[tmp]                        ▷ 親頂点への移動
    end while                                  ▷ tmp が根頂点となったときにループを抜ける
    while IsNotEmpty(SFP) do                   ▷ SFP が空でなければ以下を繰り返す
        fn ← Pop(SFP) ;                        ▷ 経路上の頂点を SFP から取り出す
        Sets[fn] ← tmp                         ▷ 頂点の親を根頂点 tmp にして経路を圧縮する
    end while
    return tmp                                 ▷ 根頂点 tmp を集合名として出力する
end function
```

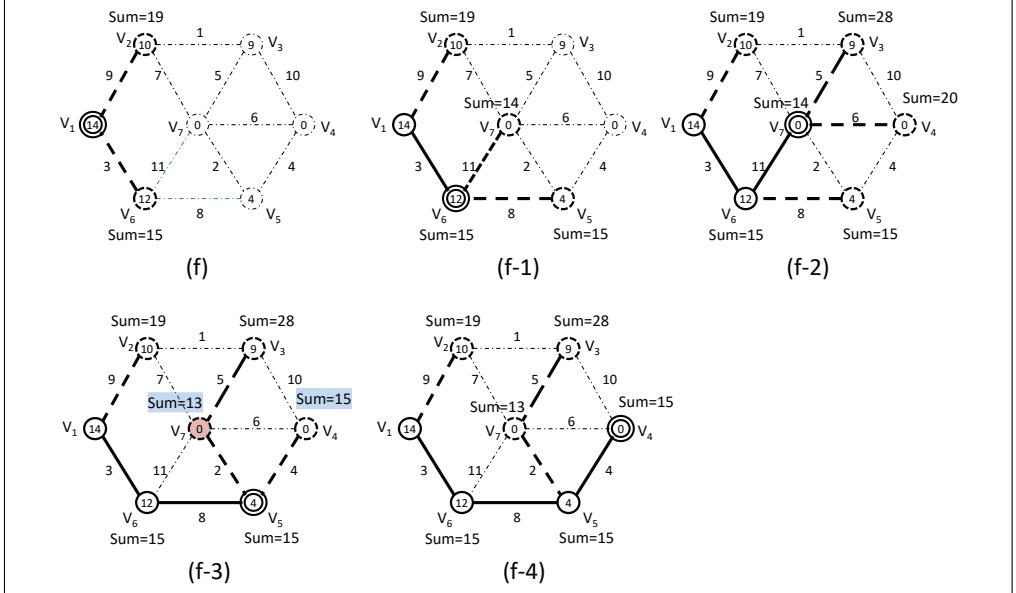
問 6.4　省略.

問 6.5　頂点数 N のグラフに対してベルマン・フォード法を適用したとき，終了時点においては，始点から各頂点への経路で $N-1$ 個以下の辺からなるもの全ての長さを調べ，その中で最短のものを各頂点に対して記録している．負閉路を構成する辺の数は $N-1$ 個以下であり，もし負閉路が存在すれば，その負閉路上の頂点への経路長はいくらでも小さくできるので，経路長の計算（配列 $Dist$ の更新計算）を続ければ，経路長をより小さくできる負閉路上の頂点が必ず 1 つ以上存在する（下記の問 6.8 の解説における図を参照のこと）．もし存在しなければ，負閉路が構成できない．よってベルマン・フォード法の最後の距離条件のチェックで負閉路が検出できる．

問 6.6　図 6.5 (e) に A* アルゴリズムを適用した場合，その探索過程は**解図 6.1** のようになる．頂点の探索順序は図 6.11 と全く同じになることに注意する．一方で，図 6.5 (f) のヒューリスティック関数を用いた場合の探索は**解図 6.2** のようになる．頂点 V_7 が，ステップ (f-2) で一度 close に入り，次のステップ (f-2) で再度 open に変化する点に注意してほしい．以上と本文の図 6.11 を比較すれば，ヒューリスティック関数の近似精度がよいほど，探索が効率的になることが確認できる．



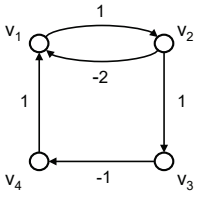
解図 6.1　ヒューリスティック関数として図 6.5 (e) を用いた A* 探索



解図 6.2　ヒューリスティック関数として図 6.5 (f) を用いた A* 探索

問 6.7　省略.

問 6.8　ベルマン・フォード法が最短の単純経路の検出に失敗するグラフの例を**解図 6.3** の左に示す．右側には Alg. 6.5 で計算した各頂点への経路長（配列 *Dist* の中身）の変化を示している．破線の下は，もう 1 回だけ余計に外側の **for** ループを回したときの値である．これは，**for** ループを抜けた後の負閉路の存在判定計算における不等式の右辺の値として利用される．この負グラフに対して，ベルマン・フォード法は負閉路の存在を報告し，最短の（単純）経路の計算に失敗する．もし頂点 v_2 から v_1 への辺の重みが -1 以上であれば，負閉路は存在しないので，ベルマンフォード法は最短の単純経路を求めることができる．



Dist	v_1	v_2	v_3	v_4
i = 0	0	-∞	-∞	-∞
i = 1	0	1	-∞	-∞
i = 2	-1	1	2	-∞
i = 3	-1	0	2	1
繰返し計算終了				
	-2	0	1	1

解図 6.3　ベルマン・フォード法が最短単純経路の検出に失敗する例

（7　章）

問 7.1　pos はパターンの先頭文字と照合されるテキスト上のインデックスである．$m - n < pos$ であるとき，pos 以降のテキスト上の文字数は n 未満となる．ところがパターン長は n なので照合に失敗する．よって，pos のとりうる値は 0 から $m - n$ までで十分である．

問 7.2　力まかせ法において時間計算量が最良となる一例は，任意の pos において照合に失敗する場合である．例えば，パターンを “zzz” とし，テキストを “aaaaaa” とすると，$0 \leqq pos \leqq 3$ を満たす各 pos において 1 度だけ照合を行う．よって，この例における照合回数は $m - n$ となる．逆に最悪の時間計算量となる一例は，任意の pos において照合に成功する場合である．例えば，パターンを “zzz” とし，テキストを “zzzzzz” とすると，$0 \leqq pos \leqq 3$ を満たす各 pos においてパターン長だけ照合を行う．よって，この例における照合回数は $n \times (m - n)$ となる．

問 7.3　教科書 P.143 で説明したとおり，パターンに出現する文字の移動量は，その文字が後ろからみて最初に出現する位置を調べれば良い．各 r，o，m，e，y，b の移動量はそれぞれ 1，2，4，6，7，9 となる．ただしパターンの末尾文字 e の移動量が 0 でないことに注意する．

問 7.4　力まかせ法と BM 法の実装例 (answer7-4.cpp) を下記に示す．この例では，照合に成功した各注目点の前後 5 文字を出力する仕様としている．なお例として記載するプログラムはすべて MacOS Sierra LLVM version 9.0.0 (clan-900.0.39.2) の環境にて動作を確認している．

```
/*
 * 問 7.4 力まかせ法とボイヤー・ムーア法を実装したプログラム
 *  1. コンパイル: 以下のコマンドを実行
 *  $ g++ -Wall answer7-4.cpp -o answer7-4 -std=c++14
 *
 *  2. 実行: 以下のコマンドを実行
 *  $ ./answer7-4 PATTERN SAMPLE_FILE_PATH
 * ただし  PATTERN はパターン，  SAMPLE_FILE_PATH はテキストに相当する入力ファイルパス
 */

#include <iostream>
#include <string>
#include <fstream>

using namespace std;

// 関数プロトタイプ宣言
// 力まかせ法
int brute_force(string haystack, string needle);
// Boyer-Moore 法
int bm_search(string haystack, string needle);

// main 関数
int main(int argc, char* argv[]){
    if(argc != 3){
        cout << "Usage: ./bm PATTERN TEXT.txt" << endl;
        return -1;
    }
    FILE* fp;
    // 検索対象のファイルを開く
    if ((fp = fopen(argv[2], "r")) == NULL) {
        // ファイルがない場合はが返されるNULL
        printf("Can't open file %s.\n", argv[2]);
        return 1;
    }
```

```
36      // ファイルサイズを取得する
37      fseek(fp, 0, SEEK_END);
38      long size = ftell(fp);
39      fseek(fp, 0, SEEK_SET);
40      char data[size];     // ファイルデータを格納する配列
41      fread(data, sizeof(char), size, fp);     // ファイル読み込み
42      // string 型の変換
43      string needle = string(argv[1]); // パターン
44      string haystack = string(data); // ファイル名
45
46      int pos, i; //一致した文字列における先頭文字のインデックス
47
48      // pos = brute_force(haystack, needle); // 力まかせ法
49      pos = bm_search(haystack, needle); // Boyer-Moore 法
50
51      // 文字列が見つかった場合
52      if (pos >= 0) {
53          // 一致した文字列と前後 5 文字を表示する
54          cout << "pos = " <<  pos << endl;
55          for(i = pos - 5; i < pos + (int)needle.size() + 5; i++){
56              if(i >= 0 && i < (int)haystack.size()){
57                  cout <<  haystack.at(i);
58              }
59          }
60          cout << endl;
61      }
62      // 文字列が見つからなかった場合
63      else {
64          cout << "Not found." << endl;
65      }
66      return 0;
67  }
68  // 力まかせ法
69  int brute_force(string haystack, string needle){
70      for( int i = 0; i < (int)haystack.size() - (int)needle.size() + 1; i++ ){
71          // i : 現在注目しているインデックス
72          int j = 0;
73          while(j < (int)needle.size()){
74              // j : 現在照合している needle 上のインデックス
75              if( haystack.at(i+j) != needle.at(j)){
76                  // 一致しなかった
77                  break;
78              }
79              j++;
80          }
81          if( j == (int)needle.size()){
82              // 一致するインデックスがあった
83              return i;
84          }
85      }
86      return -1;
87  }
88  // Boyer-Moore 法
89  int bm_search(string haystack, string needle)
90  {
91      int i,j;
92      // 移動量のテーブルを作成する
93      int table[128]; // ASCII コードに対応する
94      // 移動量テーブルの初期化
95      for(i = 0; i < 128; i ++){
96          // デフォルトの移動量はパターン長
97          table[i] = needle.size();
98      }
99      // 移動量の計算
```

```
    for(i = 0; i < (int)needle.size(); i++){
        // i は後ろから数えたインデックス
        j = (int)needle.size() - i - 1;
        // j は前から数えたインデックス
        table[(int)needle.at(j)] = i;
    }
    // パターンの探索
    i = (int)needle.size() - 1; // 注目点の初期値
    while(i < (int)haystack.size()){
        // 注目点がパターンの末尾にくるまで探索する
        j = (int)needle.size() - 1;
        int ii = i; // 今みている haystack の注目点の初期値
        while (haystack.at(i) == needle.at(j))
        {
            // 一致する限り注目点をずらしていく
            if(j == 0){
                // パターンを発見
                return i;
            }
            i--;
            j--;
        }
        // 一致していないときは，注目点 i を移動する
        i = i + table[(int)haystack.at(i)];
        // もし更新後の i が ii 以下のときは ii + 1 を代入する
        if(i <= ii){
            i = ii + 1;
        }
    }
    return -1;
}
```

問 7.5 以下にトライ行列の実装例 (answer7-5.cpp) を示す．

```
/*
 * 問 7.5 トライ行列を実装したプログラム
 *  1. コンパイル: 以下のコマンドを実行
 *  $ g++ -Wall answer7-5.cpp -o answer7-5 -std=c++14
 *
 *  2. 実行: 以下のコマンドを実行
 *  $ ./answer7-5 SAMPLE_FILE_PATH
 * ただし  SAMPLE_FILE_PATH はテキストに相当する入力ファイルパス
 *
 *  3. 実行例：プロンプトから照合するパターンを入力する
 *  $ input word < PATTERN
 *  found
 *  $ input word < PATTERN
 *  not found
 * ただし  PATTERN はテキスト文字列とする
 */

#include <iostream>
#include <fstream>
#include <string>
#include <vector>

#define num 129
using namespace std;

// 頂点のクラス
class N{
    public:
        int i; // インデックス
        int* e; // 辺ラベルの配列
```

```
31 public:
32     N(int ind); // コンストラクタ
33     ~N(); // デストラクタ
34 };
35
36 N::N(int ind){
37     e = new int[num];
38     for(int i = 0; i < num; i++){
39         e[i] = -1; // 初期化
40     }
41     i = ind;
42 }
43 N::~N(){
44     delete[] e;
45 }
46
47 // トライ行列のクラス
48 class Trie{
49     public:
50         int n; // 頂点数
51         vector<N*> table; //頂点の配列
52     public:
53         Trie(); // コンストラクタ
54         ~Trie(); // デストラクタ
55         void add(string word); // 単語を追加
56         bool Find(string word); // 単語を検索する
57         void find(string word, int& k, int& j); // 単語検索（追加用）
58 };
59 // コンストラクタ
60 Trie::Trie(){
61     n = 0;
62     N* root = new N(n); // 根頂点
63     table.push_back(root);
64 }
65 // デストラクタ
66 Trie::~Trie(){
67     for(auto itr = table.begin(); itr != table.end(); itr++){
68         delete *itr;
69     }
70 }
71 // トライ行列から単語を検索
72 void Trie::find(string w, int& k, int& j){
73     k = 0; // 頂点番号
74     j = 0; // 照合成功回数
75     while(table[k]->e[(int)w[j]] != -1){
76         k = table[k]->e[(int)w[j]];
77         j++;
78         if(j == w.size()){
79             break;
80         }
81     }
82 }
83 // トライ行列に単語を追加
84 void Trie::add(string w){
85     int k, j;
86     find(w, k, j);
87     // 未登録の新頂点を追加する
88     while(j < w.size()){
89         n++; // 頂点数をインクリメント
90         N* newNode = new N(n); // 新頂点の作成
91         table.push_back(newNode); // 新頂点の追加
92         table[k]->e[(int)w[j]] = n; // 頂点 k の辺情報に n を登録
93         k = n; // 頂点 k を更新
94         j++; // w 中の文字を更新
```

```
    }
}
// 単語を検索する
bool Trie::Find(string w){
    int k, j;
    find(w, k, j);
    if(j == w.size()){
        return true;
    }
    else{
        return false;
    }
}

// main 関数
int main(int argc, char* argv[]){

    if(argc != 2){
        cout << "Usage: ./answer7-5 TEXT" << endl;
        return -1;
    }
    Trie* t = new Trie(); // トライ行列
    ifstream ifs(argv[1]);
    string str;
    if(ifs.fail()){
        cout << "fail to read the file" << endl;
        return -1;
    }
    // ファイル読み込み
    while(getline(ifs, str)){
        t->add(str);
        cout << str << endl;
    }
    // 標準入力
    string pattern;
    while(1){
        cout << "input word < ";
        cin >> pattern;
        if(t->Find(pattern)){
            cout << "found" << endl;
        }
        else{
            cout << "not found" << endl;
        }
    }
    return 0;
}
```

問 7.6 接尾辞は $m+1$ 個あり，それぞれがサフィックス木の葉に相当するので葉の個数は $m+1$ である．各内部頂点は少なくとも 2 つ以上の子頂点を持つ．よって，内部頂点の個数が葉の個数以上になることはない．

（8 章）

問 8.1 最近点対を求めるプログラム (answer8-1.cpp) を下記に示す．本章の解答例に記載されているプログラムはすべて 7 章と同様，MacOS Sierra LLVM version 9.0.0 (clan-900.0.39.2) の環境にて動作を確認している．

```
/*
 *  1. コンパイル: 以下のコマンドを実行
 *  $ g++ -Wall answer8-1.cpp -o answer8-1 -std=c++14
 *
 *  2. 実行: 以下のコマンドを実行
 *  $ ./answer8-1 < SAMPLE_FILE_PATH
 * ただし SAMPLE_FILE_PATH は入力ファイルパス
 *
 * 入力ファイルのフォーマット
 *  1 行目: 頂点数
 *  2 行目以降: 各頂点の X 座標と Y 座標の組ただし ( X ,Y 座標は整数)
 * 入力ファイルの例
 *  6
 *  1 1
 *  2 2
 *  4 5
 *  5 4
 *  8 7
 *  10 6
 */

#include<iostream>
#include<vector>
#include<algorithm>
#include<cmath>
#include<limits>

using namespace std;

#define INF numeric_limits<int>::max()  // 非負整数最大値

// 頂点クラス
class P {
    public:
        int x;  // X 座標
        int y; // Y 座標
        int ind; // 頂点インデックス
        int kind; // 頂点が属する小問題を指す値
    public:
        // コンストラクタ
        P(int x, int y): x(x), y(y), ind(-1), kind(-1) {}
};

// 最近点対クラス
class CloPair {
    public:
        int p; // 最近点の頂点インデックス
        int q; // 最近点の頂点インデックス
        double d;  // 最近点対間の距離
    public:
        // コンストラクタ
        CloPair(): p(-1), q(-1), d(-1) {}
};

// X 座標に基づく比較関数の定義
bool ascX(const P* p, const P* q){
    return (p->x < q->x);
}
```

```
// Y 座標に基づく比較関数の定義
bool ascY(const P* p, const P* q){
    return (p->y < q->y);
}

// 2 頂点間のユークリッド距離を求める関数
double dist(P* p, P* q){
    int x = p->x - q->x;
    int y = p->y - q->y;
    return sqrt(x*x + y*y);
}

// のりしろ領域 N において距離が sigma 未満となる最近点対を求める関数
CloPair* minPair(vector<P*> N, double sigma)
{
    CloPair* ans = new CloPair();
    ans->d = sigma; // 初期値
    for(int i = 0; i < N.size(); i++){
        for(int j = i + 1; j < N.size() && (N[j]->y - N[i]->y <= sigma); j++)
            {
            // N は Y 座標値に関して整列済み。N[i].y <= N[j].y であり、
            // N[i] からの距離が sigma 未満の頂点 N[j] を探しているので
            // 少なくとも N[j].y - N[i].y <= sigma を満たす頂点 N[j] に絞られる
            // もしこの条件を満たさなければ 2 つ目の for 文を直ちに抜けて良い
            // ( N が Y 座標値に関して整列済みであることが効いている)
            // またこの条件を満たす頂点数は教科書のとおり高々 6 個
            // 2 つ目の for 文はループ回数は定数なので O(n) 時間で実行される
            double d = dist(N[i], N[j]);
            if(ans->d > d){
                ans->d = d;
                ans->p = N[i]->ind;
                ans->q = N[j]->ind;
            }
        }
    }
    return ans;
}

// 分割統治法により最近点対を求める関数
CloPair* findCloPair(vector<P*> Sx, vector<P*> Sy, int n){
    CloPair* ans = new CloPair();
    if(n < 2){
        ans->d = INF; // 最大値にセット
    }
    else if(n == 2){
        // 頂点数がちょうど 2 つの場合
        ans->p = Sx[0]->ind;
        ans->q = Sx[1]->ind;
        ans->d = dist(Sx[0], Sx[1]);
    }
    else{
        int m = n / 2;
        // x = b が境界線
        double b = (double)(Sx[m-1]->x + Sx[m]->x)/2;
        // Sx を 2 分割する (O(n)時間)
        vector<P*> Sx_l;
        vector<P*> Sx_r;
        for(int i = 0; i < n; i++){
            if(i < m){
                // kind はこの頂点が属する小問題境界線より左側か右側かのどちらか () を表
                    す値
                Sx[i]->kind = 0;
                Sx_l.push_back(Sx[i]);
```

```
            }
            else{
                // kind はこの頂点が属する小問題を表す値
                Sx[i]->kind = 1;
                Sx_r.push_back(Sx[i]);
            }
        }
        // 頂点の kind 値をもとに Sy を 2 分割する ( O(n) 時間 )
        vector<P*> Sy_l;
        vector<P*> Sy_r;
        for(int i = 0; i < n; i++){
            if(Sy[i]->kind == 0){
                Sy_l.push_back(Sy[i]);
            }
            else{
                Sy_r.push_back(Sy[i]);
            }
        }
        // 小問題を解く
        CloPair* ans_l = findCloPair(Sx_l, Sy_l, Sx_l.size());
        CloPair* ans_r = findCloPair(Sx_r, Sy_r, Sx_r.size());
        // 小問題における最近点間距離の小さい方の値
        double sigma = min(ans_l->d, ans_r->d);
        // のりしろ領域 ( 境界線付近の領域 ) を作成する ( O(n) 時間)
        vector<P*> N;
        for(int i = 0; i < n; i++){
            // のりしろ領域に属する頂点
            if((b - sigma <= Sy[i]->x) && (Sy[i]->x <= b + sigma)){
                N.push_back(Sy[i]);
            }
        }
        // N 上の最近点対を求める ( O(n) 時間)
        CloPair* v = minPair(N, sigma);
        // v, ans_l, ans_v から ans を求める
        if(v->d < sigma){
            ans->p = v->p;
            ans->q = v->q;
            ans->d = v->d;
        }
        else if(ans_l->d == sigma){
            ans->p = ans_l->p;
            ans->q = ans_l->q;
            ans->d = ans_l->d;
        }
        else{
            ans->p = ans_r->p;
            ans->q = ans_r->q;
            ans->d = ans_r->d;
        }
        // オブジェクトの後始末
        delete(ans_l);
        delete(ans_r);
        delete(v);
    }
    // 出力
    return ans;
}

int main()
{
    int nodeNum; // 頂点数
    double x; // 頂点の X 座標
    double y; // 頂点の Y 座標
    vector<P*> Sx; // 頂点配列 ( X 座標値に基づく整列)
```

```
185     vector<P*> Sy; // 頂点配列 ( Y 座標値に基づく整列)
186
187     // ファイル入力
188     scanf("%¥dn", &nodeNum);
189     while(cin >> x >> y ){
190         P* pos = new P(x, y);
191         Sx.push_back(pos);
192         Sy.push_back(pos);
193     }
194
195     // 整列 (O( n log n) 時間)
196     sort(Sx.begin(), Sx.end(), ascX);
197     sort(Sy.begin(), Sy.end(), ascY);
198
199     // 頂点インデックスの設定
200     for(int i = 0; i < nodeNum; i++){
201         Sx[i]->ind = i;
202     }
203
204     // 最近点対を求める
205     CloPair* pair = findCloPair(Sx, Sy, nodeNum);
206
207     // 出力
208     if(pair->p > -1 && pair->q > -1){
209         cout << "The closest pair of points are as follows:" << endl;
210         cout << Sx[pair->p]->x << "," << Sx[pair->p]->y << endl;
211         cout << Sx[pair->q]->x << "," << Sx[pair->q]->y << endl;
212         cout << "Distance: " << pair->d << endl;
213     }
214     else{
215         cout << "Input a problem that contains at least two points" << endl;
216     }
217     return 0;
218 }
```

上記のプログラムでは，教科書 P.166 にて説明したとおり，境界値付近の最近点を $O(n)$ 時間で求めるよう改良している．改良のポイントは 78 行目のループ処理にある．一見 $O(n)$ 時間かかるように見えるが実際には N が Y 座標値に関して整列済みである点と教科書 P.165 で述べた観測事実から，定数時間でループ処理は完了する．

問 8.2 図 8.5 に示した行列 A と B の行列積 $C = A \times B$ を考える．談話室で述べたとおり，シュトラッセンのアルゴリズムは以下の 7 個の部分行列 $D_1, \ldots, D_7$ を用いて C を求める:

$$D_1 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$$
$$D_2 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$$
$$D_3 = (A_{11} - A_{21}) \times (B_{11} + B_{12})$$
$$D_4 = (A_{11} + A_{12}) \times B_{22}$$
$$D_5 = A_{11} \times (B_{12} - B_{22})$$
$$D_6 = A_{22} \times (B_{21} - B_{11})$$
$$D_7 = (A_{21} + A_{22}) \times B_{11}$$

各 D_i は $2^{n-1} \times 2^{n-1}$ の部分行列積を計算することで求められる．その後，C は各 D_i から次式をもとに構成できる．

$$C_{11} = D_1 + D_2 - D_4 + D_6$$
$$C_{12} = D_4 + D_5$$
$$C_{21} = D_6 + D_7$$
$$C_{22} = D_2 - D_3 + D_5 - D_7$$

次に 2^n 桁 (ただし基底を b とする) 同士の実数 a，b の乗算を行う分割統治アルゴリズム (Karatsuba 法と呼ばれる) を簡単に紹介する．a，b をそれぞれ下位部 a_0, b_0 と上位部 a_1，b_1 に分割する．a_0，a_1，b_0，b_1 はそれぞれ 2^{n-1} 桁で表現されるものとする．このとき，以下の 4 つの乗算結果 r_1, r_2, r_3, r_4 を求めると，$a \times b$ は，$a \times b = r_1 + (r_2 + r_3)B + r_4 B^2$ と書くことができる．

$$r_1 = a_0 \times b_0, r_2 = a_0 \times b_1, r_3 = a_1 \times b_0, r_4 = a_1 \times b_1.$$

ただし，xB は x を 2^{n-1} 桁だけシフトして得られる値とする．すなわち b を基底 (例えば 10 進数ならば $b = 10$ である) とすると，xB は x に $b^{2^{n-1}}$ を掛け合わせた値と一致する．

各 r_i は 2^{n-1} 桁同士の乗算結果であることから，筆算風の上記の計算は，元問題を 4 つの部分乗算問題に分割して解く分割統治型アプローチをとっていると考えることができる．このとき，教科書 P.168 で示した簡易版マスター定理における分割数 a は 4 である．また各部分乗算問題のサイズは元問題の半分である (すなわち $c = 2$) ことから，マスター定理より，時間計算量は $O(n^{\log_2 4}) = O(n^2)$ となる．

ただしこの時間計算量は，各桁同士を単純に掛け合わせる素朴な手法と変わりない．実は部分乗算の回数を 4 回から 3 回に減らすことが可能である．実際，次の r_3' を求めると $r_2 + r_3 = r_1 + r_3' + r_4$ が成り立つことがわかる:

$$r_3' = (a_1 - a_0) \times (b_0 - b_1).$$

$a \times b = r_1 + (r_1 + r_3' + r_4)B + r_4 B^2$ と書けるので，3 つの部分乗算結果 r_1, r_3', r_4 を用いて元問題の解を構成できることがわかる．この場合の時間計算量は，マスター定理における定数が $a = 3, c = 2$ となるので，$O(n^{\log_2 3}) = O(n^{1.585})$ と表すことができる．

問 8.3 プログラムの実装例 (answer8-3.cpp) は下記のとおりである．$n = 40$ 以降で実行時間に顕著な差が現れてくる．ただしプログラムは int 型変数を用いており，適切な出力を得るのは，int 型変数で表現可能な $n \leqq 46$ までであることに注意する ($n > 46$ の n を入力する場合，long int 型変数を用いる等の修正が必要である)．

```
/*  1. コンパイル: 以下のコマンドを実行
 *  $ g++ -Wall answer8-3.cpp -o answer8-3 -std=c++14
 *
 *  2. 実行: 以下のコマンドを実行
 *  $ ./answer8-3 #n
 * ただし  #n は非負整数
 */

#include<iostream>
using namespace std;

// 再帰計算を用いてフィボナッチ数列を求めるプログラム
int GetF(int n){
    if(n <= 1){
```

```
15          return n;
16      }
17      int fibo_1 = GetF(n-1);
18      int fibo_2 = GetF(n-2);
19      int out = fibo_1 + fibo_2;
20      return out;
21  }
22
23  // 動的計画法を用いてフィボナッチ数列を求めるプログラム
24  int GetF_DP(int n, int* knownF){
25      if(n <= 1){
26          return n;
27      }
28      else if(knownF[n] > 0){
29          // メモ化による重複計算を防ぐ効用
30          return knownF[n];
31      }
32      else{
33          knownF[n] = GetF_DP(n-1, knownF) + GetF_DP(n-2, knownF);
34          return knownF[n];
35      }
36  }
37
38  int main(int argc, char* argv[])
39  {
40      int* knownF;
41      if(argc != 2){
42          cout << "usage: ./8-3 NUM" << endl;
43          exit(0);
44      }
45      int num = atoi(argv[1]);
46      knownF = new int[num+1];
47      // knownF 配列の初期化
48      for(int i = 0; i <= num; i++){
49          if(i <= 1){
50              knownF[i] = i;
51          }
52          else{
53              knownF[i] = 0;
54          }
55      }
56      // 動的計画法に基づく解法
57      cout << "by DP comp.: " << GetF_DP(num, knownF) << endl;
58      // 再帰計算に基づく解法
59      cout << "by Recursive comp.:" << GetF(num) << endl;
60      return 0;
61  }
```

問 8.4 動的計画法によって 0-1 ナップサック問題を解くプログラム (answer8-4.cpp) の実装例を以下に示す.

```
1   /*
2    * 動的計画法により  0-1 ナップサック問題を解くプログラム
3    *
4    *  macOS Sierra LLVMversion 9.0.0 clan-900.0.39.2
5    * の環境にて動作確認
6    *
7    *  1. コンパイル: 以下のコマンドを実行
8    *  $ g++ -Wall answer8-4.cpp -o answer8-4 -std=c++14
9    *
10   *  2. 実行: 以下のコマンドを実行
11   *  $ ./answer8-4 < SAMPLE_FILE_PATH
12   * ただし  SAMPLE_FILE_PATH は入力ファイルへのパスに相当する
13   *
```

```
14  * 入力ファイルのフォーマット
15  *  1 行目: 荷物の個数
16  *  2 行目: ナップサック容積
17  *  3 行目以降: 各荷物の容積と価値の組
18  * 例
19  *  5
20  *  5
21  *  1 1
22  *  3 4
23  *  4 5
24  *  2 4
25  *  1 2
26  *
27  */
28
29 #include<cstdio>
30 #include<iostream>
31 #include<cstdlib>
32 #include<cmath>
33
34 using namespace std;
35
36 // 0-1 ナップザック問題のクラス
37 class Knapsack {
38     public:
39         int  n;     // 荷物の数
40         int  b;     // ナップサックの容積
41         int *a;     // 各荷物の容積
42         int *c;     // 各荷物の価値
43     public:
44         // コンストラクタ
45         Knapsack(): n(0), b(0), a(NULL), c(NULL) {}
46         // 0-1 ナップサック問題を読み込む
47         void loadKnapsack();
48         // 0-1 ナップサック問題を破棄する
49         void destroyKnapsack();
50         // 0-1 ナップサック問題を表示する
51         void printKnapsack();
52         // 動的計画法により 0-1 ナップサック問題を解く
53         void solve_DP();
54 };
55
56 // 0-1 ナップサック問題を標準入力から読み込む
57 void Knapsack::loadKnapsack(){
58     // 荷物の数の読み込み
59     scanf("%d", &n);
60     // ナップザックの容積の読み込み
61     scanf("%d", &b);
62
63     // 各荷物の容積と価値の配列を確保
64     a = new int[n];
65     c = new int[n];
66
67     if (a == NULL || c == NULL) {
68         cout << "memory overflow!" << endl;
69         exit(1);
70     }
71     // 各荷物の容積と価値の配列の読み込み
72     for(int i = 0; i < n; i++)
73         scanf("%d %d", &a[i], &c[i]);
74
75 }
76
77 // 0-1 ナップザック問題を表示する
```

```
78  void Knapsack::printKnapsack(){
79    cout << "-- Knapsack problem --" << endl;
80    cout << "Objective: maximize " << endl;
81    for(int i = 0; i < n; i++){
82        cout << c[i] << "x" << (i+1);
83        if (i < n-1){
84            cout << " + ";
85        }
86    }
87    cout << endl;
88    cout << "Constraint: satisfy" << endl;
89    for(int i = 0; i < n; i++){
90        cout << a[i] << "x" << (i+1);
91        if (i < n-1){
92            cout << " + ";
93        }
94    }
95    cout << " <= " << b << endl;
96  }
97
98  // 動的計画法により 0-1 ナップサック問題を解く
99  void Knapsack::solve_DP(){
100     // 2 次元表を作る
101     int y[n+1][b+1];
102     int type[n+1][b+1];
103
104     // 最適解の組み合わせを表す配列
105     int answer[n];
106
107     // 表を完成させる
108     for(int i = 0; i < n; i++){
109         // 表の i 列 j 行目を完成させる
110         for(int j = 0; j < b+1; j++){
111             // 1 列目
112             if(i == 0){
113                 if(a[i] <= j ) {
114                     // 荷物 i を入れることができる
115                     y[i][j] = c[i];
116                     type[i][j] = 1;
117                 }
118                 else{
119                     // 荷物 i を入れることができない
120                     y[i][j] = 0;
121                     type[i][j] = 0;
122                 }
123             }
124             // 2 列目以降
125             else{
126                 // 荷物 i を含めないときの最適値 A
127                 int maxWithoutI = y[i-1][j];
128                 // 荷物 i を含めるときの最適値 B
129                 int maxWithI = 0;
130                 if(j - a[i] >= 0){
131                     // 荷物を含めることができるとき
132                     maxWithI = y[i-1][j - a[i]] + c[i];
133                 }
134                 //最適値 A と B で大きい方を y[i][j] とする
135                 if(maxWithoutI < maxWithI){
136                     y[i][j] = maxWithI;
137                     type[i][j] = 1;
138                 }
139                 else if(maxWithoutI > maxWithI){
140                     y[i][j] = maxWithoutI;
141                     type[i][j] = 0;
```

```
                }
                else {
                    y[i][j] = maxWithoutI;
                    type[i][j] = 2;
                }
            }
        }
    }

    // 出力
    int curVolume = b;

    cout << "Answer" << endl;
    cout << "x = (";

    for(int i = n-1; i >= 0; i--){
        if(type[i][curVolume] == 1){
            answer[i] = 1;
            curVolume -= a[i];
        }
        else {
            answer[i] = 0;
        }
    }
    for(int i = 0; i < n-1; i++){
        cout << answer[i] << ", ";
    }
    if(answer[n-1] == 1){
        cout << "1)" << endl;;
    }
    else{
        cout << "0)" << endl;;
    }
    cout << "Max-value = " << y[n-1][b] << endl;
}

// 0-1 ナップザック問題を破棄する
void Knapsack::destroyKnapsack()
{
  delete(a);
  delete(c);
}

/*
 * main 関数
 *
 */
int main()
{
    Knapsack* problem = new Knapsack();

    // 0-1 ナップサック問題を読み込む
    problem->loadKnapsack();

    // 0-1 ナップサック問題を表示する
    problem->printKnapsack();

    // 動的計画法により 0-1 ナップサック問題を解く
    problem->solve_DP();

    // 0-1 ナップサック問題を破棄する
    problem->destroyKnapsack();

    return 0;
```

```
}
```

問 8.5 各コインの金額を a_i $(1 \leqq i \leqq k)$ と表す．ただし $a_1 < \cdots < a_k$ とし，必ず $a_1 = 1$ である単位コインをもつものとする．いま $a_1, \ldots, a_k$ の k 個のコインを用いて金額 b を両替する問題を考える．この問題の解である最小コイン枚数を $y_k(b)$ と表すと，次の漸化式が成り立つ．

$$y_1(b) = \begin{cases} +\infty & \text{if } b < 0 \\ b & \text{if } 0 \leqq b \end{cases}$$

$$y_k(b) = min(y_{k-1}(b - a_k) + 1,\ y_{k-1}(b)) \text{ if } k > 1$$

教科書 P. 174 で示した 0-1 ナップサック問題と同様，この漸化式をもとに動的計画法により両替問題を解くことができる．プログラム (answer8-5.cpp) の実装例を以下に示す．

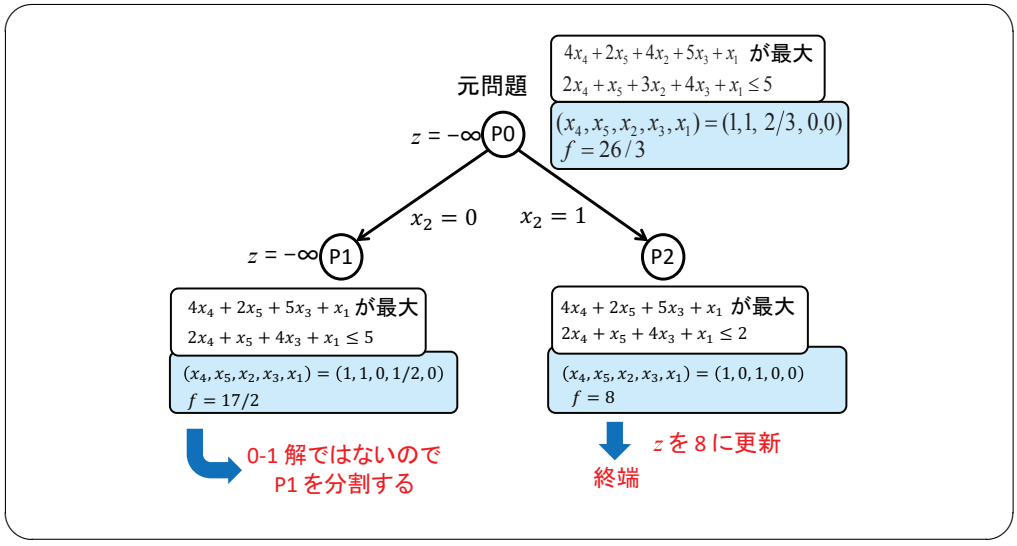
```
/*
 *  1. コンパイル: 以下のコマンドを実行
 *  $ g++ -Wall answer8-5.cpp -o answer8-5 -std=c++14
 *
 *  2. 実行: 以下のコマンドを実行
 *  $ ./answer8-5 < SAMPLE_FILE_PATH
 * ただし  SAMPLE_FILE_PATH は入力ファイルへのパスに相当する
 *
 * 入力ファイルのフォーマット
 *  1 行目: 両替金額
 *  2 行目以降: 各コインの金額
 * 例
 *  coin:63
 *  1
 *  5
 *  10
 *  25
 *  21
 *
 */

#include <iostream>
#include <vector>
#include <algorithm>
#include <cstdio>

using namespace std;

// 動的計画法により両替問題を解く関数
void makeChange(vector<int> dcoins, int value){
    int differentCoins = dcoins.size(); // コイン枚数
    // 0 から両替金額までの各金額において，その金額を作る最小コイン枚数が格納される配列
    vector<int> coinUsed;
    coinUsed.resize( value + 1 );

    // 最小の枚数でその金額を作るコインの組において，その組に最後に追加されるコインが格納される配列
    // 例 x = lastCoin[i] は，金額 i を作るコインの組において，x が最後に追加されていることを意味する
    vector<int> lastCoin;
    lastCoin.resize( value + 1 );

    coinUsed[0] = 0; // 金額 0 を作るのにコインは必要ない
    lastCoin[0] = 0; // 金額 0 を作るのにコインは必要ない
    for(int i = 1; i <= value; i++){
        int minCoins = i; // 初期値はすべて単位コインで作成した場合の値
```

```
        int newCoin = 1;  // 初期値はすべて単位コインで作成した場合の値
        for( int j = 0; j < differentCoins; j++){
            if( dcoins[j] > i ){ // j 番目のコインは利用できない
                continue;
            }
            if(coinUsed[i-dcoins[j]]+1 < minCoins){
                // j 番目のコインを使うことで最小コイン数を減らすことができる
                minCoins = coinUsed[ i - dcoins[ j ] ] + 1;
                // 新しく追加されるコインを登録しておく
                newCoin = dcoins[j];
            }
        }
        coinUsed[i] = minCoins;
        lastCoin[i] = newCoin;
    }
    int k = value;
    cout << "Combination: " << lastCoin[k];
    k = k - lastCoin[k];

    while(k != 0){
        cout << "," << lastCoin[k] ;
        k = k - lastCoin[k];
    }
    cout << endl;
    cout << "Min. num. of coins:" << coinUsed[value] << endl;
}

int main(){
    // ファイル読み込み
    int coin; // 両替する金額
    int denomi;  // コインの金額
    vector<int> dcoins; // コインの金額を格納している配列

    scanf("coin:%d \n", &coin);
    while(cin >> denomi){
        dcoins.push_back(denomi);
    }
    // 両替問題を解く
    makeChange(dcoins, coin);

    return 0;
}
```

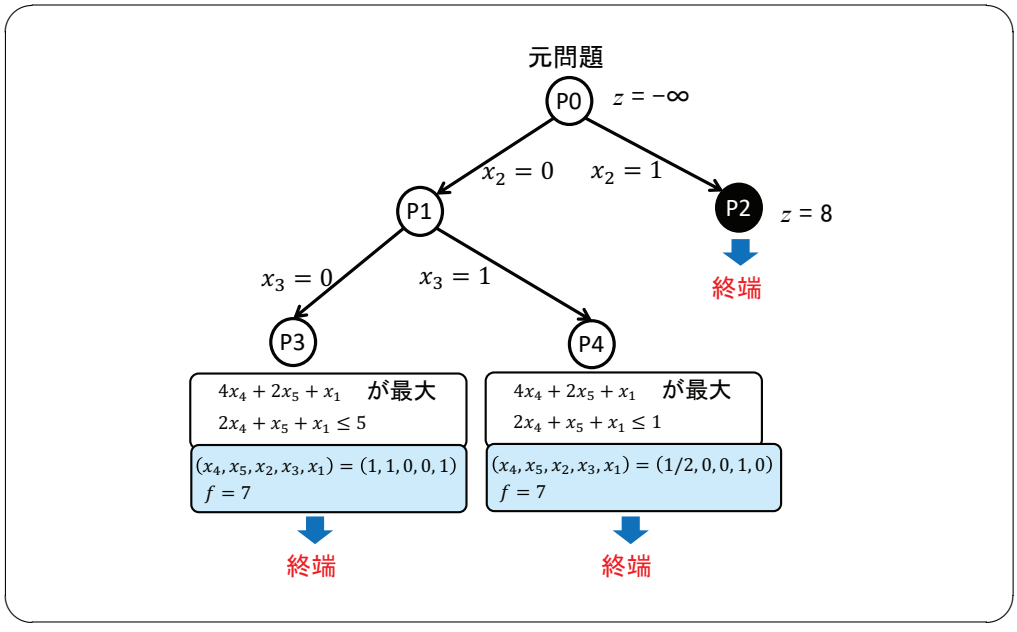
問 8.6 教科書 P.180 で述べたとおり，分枝操作を適用する変数の選択方法には検討の余地がある．本文では，計数比率の高い変数を選択する戦略を示した．ここでは緩和問題の解における変数 x_q を選択する場合を考える．x_q を選択することは，元問題の緩和問題の解において唯一整数でなかった変数 (すなわち x_q) を固定することを意味する．この選択の結果として，部分問題の解が暫定解となりやすくなる場合がある．

解図 8.1 は，問題 P_0 の x_q に相当する変数 x_2 を選択して得られる部分問題 P_1，P_2 とその緩和問題 $\overline{P_1}$，$\overline{P_2}$ の解を示している．実際，$\overline{P_2}$ の解は暫定解となっており，この時点で最適解 $(x_4, x_5, x_2, x_3, x_1) = (1, 0, 1, 0, 0)$ を得ている．この暫定解を用いることで，部分問題 P_3 と P_4 を限定操作により終端させることができる．結果として，x_q を選択した場合の探索木は**解図 8.2** のとおりとなる．計数比較に基づく変数選択の場合，10 個の部分問題を扱う必要があり (図 8.16 参照)，これと比べて格段に探索木が縮退される．

計数比率に基づく変数選択の場合，各変数の選択順位は一意に固定される．他方，x_q を選択する場合，現在解いている緩和問題の解に応じて変数の選択順位が動的に変わる．緩和問題の求解操作により得られる情報を活用している点から，合理的な選択戦略といえるかも



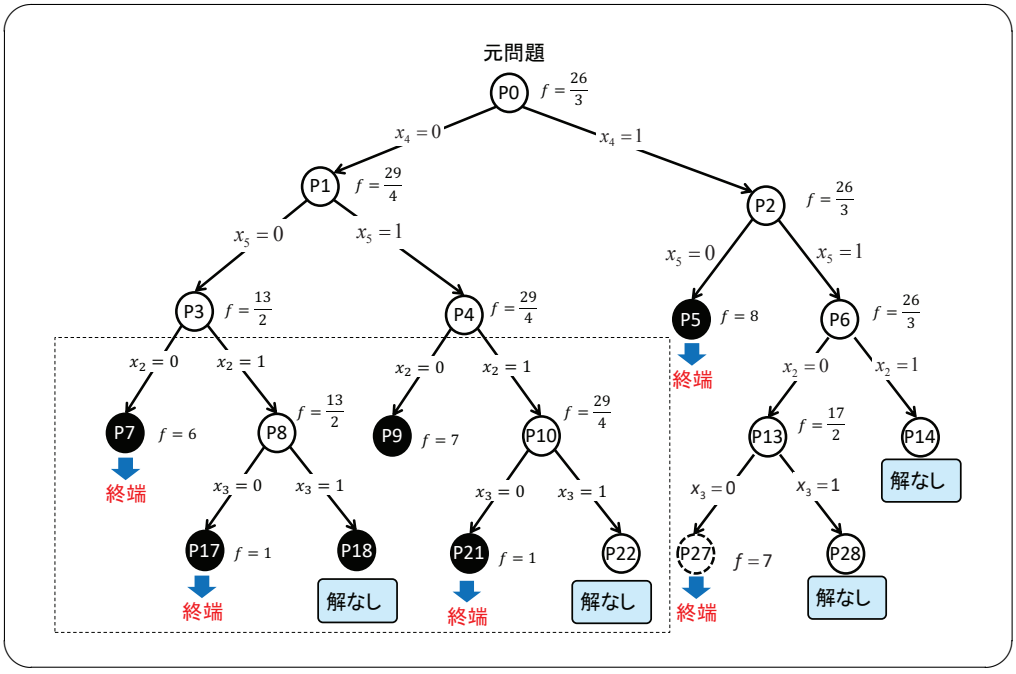
解図 8.1　変数 x_q を選択した場合の探索木（その 1）



解図 8.2　変数 x_q を選択した場合の探索木（その 2）

しれない．

問 8.7　教科書では幅優先探索に基づき探索木を構成したが，深さ優先探索を用いることも可能である．**解図 8.3** は，計数比率に基づく変数選択において左優先深さ優先探索を用いて構成した探索木を示している．図 8.16 と比較すると，点線部で囲まれた 8 個の部分問題が新たに含まれることに気づく．左優先のため，選択される各変数 x に対し，$x=0$ と固定して



解図 8.3　左優先深さ優先探索で構成される探索木

部分問題を構成する．他方，x は計数比率の高い (重要度の高い) 変数であり，最適解では $x=1$ となると考える方が妥当である．この点から探索空間において最適解から離れた部分問題から探索していると見ることができる．このように，変数の選択戦略だけでなく，選択変数の割り当てや幅優先・深さ優先等の探索方法に応じて，構成される探索木は大きく変わる．実問題においては，問題独自の特性をよく観察しながら，より効率的な探索法を設計することが必要であろう．

問 8.8　分枝限定法を用いて 0-1 ナップサック問題を解くプログラム (answer8-8.cpp) の実装例を以下に示す．なお，この例では本文で説明した幅優先探索を採用している．問 8-4 における実装例とは異なるデータ構造を用いている点に注意してほしい．今回の例では，各部分問題に相当するクラスを定義し，終端されていない部分問題のオブジェクトをキューにより管理している．これにより幅優先探索を実現している．問 8.7 の深さ優先探索の実装も行い，ぜひ両者を比較してみてほしい．

```
/*
 * 問 8.8 分枝限定法により 0-1 ナップサック問題を解くプログラム
 *
 *
 *  1. コンパイル: 以下のコマンドを実行
 *  $ g++ -Wall answer8-8.cpp -o answer8-8 -std=c++14
 *
 *  2. 実行: 以下のコマンドを実行
 *  $ ./answer8-8 < SAMPLE_FILE_PATH
 * ただし  SAMPLE_FILE_PATH は入力ファイルへのパスに相当する
 *
 * 入力ファイルのフォーマット
 *  1 行目: 荷物の個数
```

```
14  *   2 行目: ナップサック容積
15  *   3 行目以降: 各荷物の容積と価値の組
16  * 例
17  *   5
18  *   5
19  *   1 1
20  *   3 4
21  *   4 5
22  *   2 4
23  *   1 2
24  *
25  */
26
27  #include<iostream>
28  #include<cmath>
29  #include<queue>
30  #include<algorithm>
31
32  using namespace std;
33
34  #define INF numeric_limits<double>::min()  // 最小値
35
36  // 荷物のクラス
37  class I {
38      public:
39          int a; // 荷物の容積
40          int c; // 荷物の価値
41          int ind;  // 荷物のインデックス
42      public:
43          // コンストラクタ
44          I(int a, int c, int ind): a(a), c(c), ind(ind) {}
45  };
46
47  // 部分問題のクラス
48  class Task {
49  public:
50      int d; // この部分問題の深さ
51      int vol; // この部分問題の容積
52      int val; // この部分問題の価値
53      int* comb; // この部分問題の解
54  public:
55      // コンストラクタ
56      Task(int d, int vol, int val, int* c): d(d), vol(vol), val(val), comb(c)
            {}
57      ~Task(); // デストラクタ
58  };
59
60  // デストラクタ
61  Task::~Task(){
62      delete[] comb;
63  }
64
65  // 比較関数の定義 ( 比率係数に基づく整列用 )
66  bool asc(const I* p, const I* q){
67      double r_p = (double)((double)p->c/(double)p->a);
68      double r_q = (double)((double)q->c/(double)q->a);
69      return (r_p > r_q);
70  }
71
72  // 0-1 ナップザック問題のクラス
73  class Knapsack {
74      public:
75          int  n;     // 荷物の数
76          int  b;     // ナップサックの容積
```

```
77          vector<I*> v;     // 荷物オブジェクトの配列
78          int z; // 暫定解の値
79          int* zComb; // 暫定解
80      public:
81          // コンストラクタ
82          Knapsack();
83          // デストラクタ
84          ~Knapsack();
85          // 0-1 ナップサック問題を読み込む
86          void loadKnapsack();
87          // 0-1 ナップサック問題を表示する
88          void printKnapsack();
89          // 分枝限定法により 0-1 ナップサック問題を解く
90          void solve_BB();
91          // 緩和問題を解く
92          void solve(Task* t, double& value, bool& flag, int& q);
93  };
94
95  // コンストラクタ
96  Knapsack::Knapsack(){
97      n = 0;
98      b = 0;
99      z = INF;
100 }
101
102 // デストラクタ
103 Knapsack::~Knapsack(){
104     for(int i = 0; i < n; i++){
105         delete v[i];
106     }
107     delete[] zComb;
108 }
109
110 // 0-1 ナップサック問題を標準入力から読み込む
111 void Knapsack::loadKnapsack(){
112     // 荷物の数の読み込み
113     scanf("%d", &n);
114     // ナップザックの容積の読み込み
115     scanf("%d", &b);
116
117     int a; // 荷物の容積
118     int c; // 荷物の価値
119     // 各荷物の容積と価値の配列の読み込み
120     for(int i = 0; i < n; i++){
121         scanf("%d %d", &a, &c);
122         I* p = new I(a, c, (i+1));
123         v.push_back(p);
124     }
125     // 暫定解を保持する配列
126     zComb = new int[n];
127 }
128
129 // 0-1 ナップザック問題を表示する
130 void Knapsack::printKnapsack(){
131   cout << "-- Knapsack problem --" << endl;
132   cout << "Objective: maximize " << endl;
133   for(int i = 0; i < n; i++){
134       cout << v[i]->c << "x" << v[i]->ind;
135       if (i < n-1){
136           cout << " + ";
137       }
138   }
139   cout << endl;
140   cout << "Constraint: satisfy" << endl;
```

```
141     for(int i = 0; i < n; i++){
142         cout << v[i]->a << "x" << v[i]->ind;
143         if (i < n-1){
144             cout << " + ";
145         }
146     }
147     cout << " <= " << b << endl;
148 }
149
150 // 分枝限定法により 0-1 ナップサック問題を解く
151 // 幅優先探索 + 係数比率に基づく変数選択
152 void Knapsack::solve_BB(){
153
154     // 現問題の解
155     int* comb = new int[n];
156
157     // 係数比率に基づく整列 ( 降順 )
158     sort(v.begin(), v.end(), asc);
159
160     // 0-1 ナップサック問題を表示する
161     printKnapsack();
162
163     // タスクキュー
164     queue<Task*> queue;
165
166     // 元問題のタスクをキューに作成
167     queue.push(new Task(0, b, 0, comb));
168
169     while(!queue.empty()){
170         Task* t = queue.front(); // 先頭のタスクを取得
171         int depth = t->d; // 部分問題 t の深さ
172
173         if(t->vol >= 0){
174             // 部分問題 t の緩和問題を解く
175             double value = t->val; // 緩和問題の最適解の値
176             bool flag = true; // 緩和問題の解が 0-1 解かどうかを示すフラグ
177             int q = depth;
178
179             solve(t, value, flag, q);
180
181             if(flag && z < value){
182                 // 暫定解の更新を行う
183                 z = value;
184                 for(int i = 0; i < n; i++){
185                     if(i < depth){
186                         zComb[i] = t->comb[i];
187                     }
188                     else if(i <= q){
189                         zComb[i] = 1;
190                     }
191                     else{
192                         zComb[i] = 0;
193                     }
194                 }
195             }
196
197             if(!flag && value > z){
198                 // 0-1 解ではないが緩和問題の解が暫定解を上回る
199                 // 分枝操作を適用して 2 つの部分問題を作成する
200                 int* comb_0 = new int[n];
201                 int* comb_1 = new int[n];
202                 for(int i = 0; i < depth; i++){
203                     comb_0[i] = t->comb[i];
204                     comb_1[i] = t->comb[i];
```

```
                }
                comb_0[depth] = 0;
                comb_1[depth] = 1;

                Task* t_0 = new Task(depth+1, t->vol, t->val, comb_0);
                // depth 番目の荷物を使う場合
                int t_1_vol = t->vol - v[depth]->a;
                int t_1_val = t->val + v[depth]->c;
                Task* t_1 = new Task(depth+1, t_1_vol, t_1_val, comb_1);

                // queue に部分問題を追加する
                queue.push(t_0);
                queue.push(t_1);
            }
        }
        queue.pop(); // 先頭のタスクを削除
    }

    // 出力
    for(int i = 0; i < n; i++){
        cout << "x" << v[i]->ind << "=" << zComb[i] << " ";
    }
    cout << endl;
    cout << "Max-value: " << z << endl;

    delete[] comb;
}

// 緩和問題を解く
void Knapsack::solve(Task* t, double& value, bool& flag, int& q){

    // 緩和問題（連続ナップサック問題）の解を求める
    int tmp_volume = 0; // 緩和問題を解くための作業用変数その 1
    int volume = t->vol; // 緩和問題を解くための作業用変数その 2
    double x_q;

    while(q < n){
        tmp_volume += v[q]->a;
        if(tmp_volume > t->vol){
            flag = false; // 整数解ではない
            break;
        }
        else{
            value += v[q]->c;
            volume -= v[q]->a;
            if(tmp_volume == t->vol){
                break;
            }
            q++;
        }
    }
    // x_q と緩和問題の最適解の値を求める
    if(!flag){
        // 0-1 解でない
        x_q = (double)((double)volume/(double)v[q]->a);
        value += (double)(x_q*v[q]->c);
    }
}

/*
 * main 関数
 */
int main()
{
```

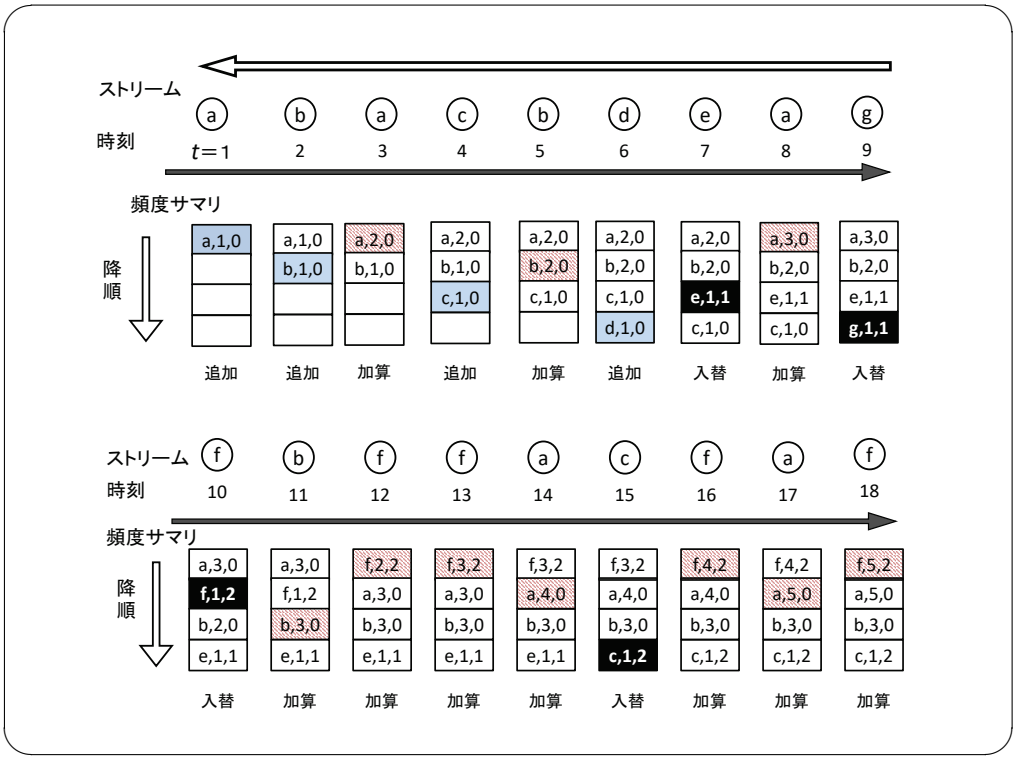
```
    Knapsack* problem = new Knapsack();

    // 0-1 ナップサック問題を読み込む
    problem->loadKnapsack();

    // 分枝限定法により 0-1 ナップサック問題を解く
    problem->solve_BB();

    // 0-1 ナップサック問題を破棄する
    delete problem;

    return 0;
}
```

この問題では，ナップサック容積 b と荷物の数 n に応じてプログラムの実行時間がどのように推移するかも問われている．実装例を参考にすれば分枝操作のみ適用するプログラムを直ちに作成することができる．限定操作を適用しない場合，解候補となる変数の任意の組を列挙して解を求めることとなる．よって n の増加に対して指数爆発的に実行時間がかかる．以下にランダムに 0-1 ナップサック問題を作成するプログラム (rand.cpp) を載せる．このプログラムを用いて，実際の実行時間を確かめると良い．

```
/*
 * ランダムに  0-1 ナップサック問題を作成するプログラム
 *  1. コンパイル: 以下のコマンドを実行
 *  $ g++ -Wall rand.cpp -o rand -std=c++14
 *
 *  2. 実行: 以下のコマンドを実行
 *  $ ./rand NUM VOLUME MAX_VALUE
 * ただし  NUM は荷物の個数 , VOLUME はナップサック容積, MAX_VALUE
 * は各荷物の最大価値とする
 */

#include<ctime>
#include<cstdio>
#include<cstdlib>
#include<iostream>

using namespace std;

/*
 * main 関数
 * 引数  : argc - コマンドライン引数の数
 *       : argv - コマンドライン引数配列
 * 戻値  : 成功した場合 0 を，失敗した場合 0 以外を返す
 */
int main(int argc, char *argv[])
{
    int i;
    int num;          // 荷物の数
    int max_value = 0;  // 荷物の価値の最大値
    int volume = 0; // ナップサック容量

    if (argc < 3) {
        cout << "Usage: " << argv[0] << " [NUM] [VOLUME] [MAX_VALUE]" << endl
            ;
        return 1;
    }
    // 2 番目の引数は荷物の個数
    num = atoi(argv[1]);
    // 3 番目の引数はナップサック容量
    volume = atoi(argv[2]);

```

```
41      if (argc == 4){
42          // 4 番目の引数は価値と容量の最大値
43          max_value = atoi(argv[3]) + 1;
44      }
45      // 乱数の種の設定
46      srand(time(NULL));
47
48      cout << num << endl;
49      cout << volume << endl;
50
51      // 乱数の出力
52      for (i=0; i < num; i++)
53          if ((volume == 0) & (max_value == 0))
54              cout << rand() << " " << rand() << endl;
55          else
56              cout << (rand()%volume)+1 << " " << (rand()%max_value)+1 << endl;
57      return 0;
58  }
```

問 8.9　実行過程を**解図 8.4** に示した．相対最大誤差 $\epsilon = 0.26$ であるので，頻度サマリ中のカウンタ（3 項組）の数は $\lceil 1/\epsilon \rceil = 4$ となっている．時刻 $t = 18$ の時点で許容される頻度誤差の最大値は $18 \cdot \epsilon \approx 4.68$ であるが，カウンタ内の誤差（第 3 項）は全てその範囲内に収まっている．また出現頻度が 4.68 以上のアイテム a と f も頻度サマリ中に保持されていること（ϵ-完全性）が確認できる．



解図 8.4　Space Saving の実行過程

問 8.10 まず，疑似コード設計の前提となるデータ構造について考えてみる．**解図 8.5** に，解図 8.4 の実行過程の $t = 5$ から $t = 10$ の間における頻度サマリ周辺の変化の様子を示した．なおここでは，解図 8.4 との対応をとりやすくために，頻度サマリ内の 3 項組の配置を，本文の図 8.21 とは上下を逆にして書いてある．

このデータ構造はポインタ等を利用すればそのまま実現できるが，処理が若干複雑になる．そこで少し工夫して配列を用いて実現し，処理手順を簡潔にして高速化を試みる．配列を用いた実装方法もいろいろと考えられるが，一つの例を**解図 8.6** と**解図 8.7** に示す．使用した配列は頻度サマリ FS とハッシュ表 HS，およびバケット配列 BS の 3 つである．

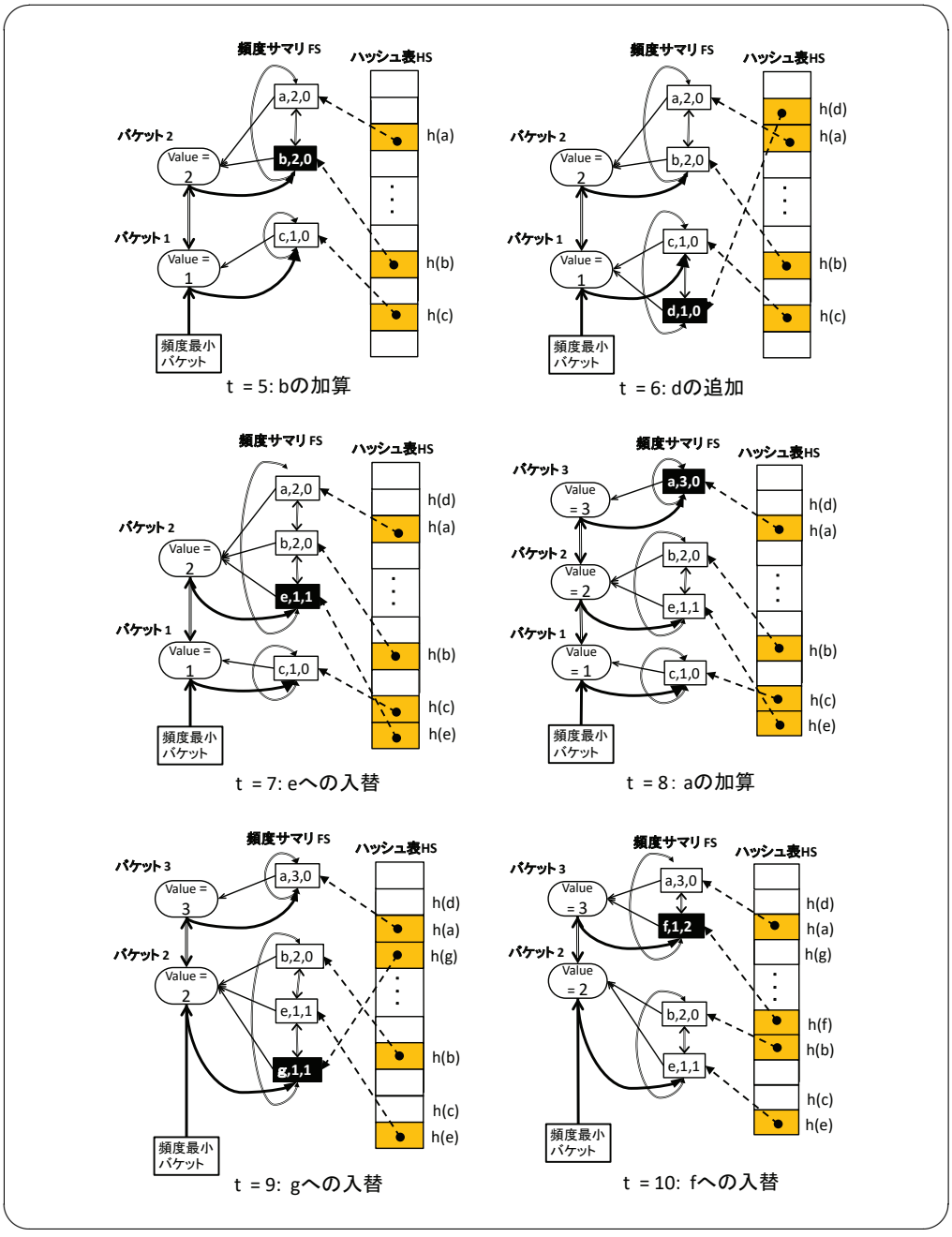
バケット配列 BS はバケットを単位要素とする配列で，個々のバケットは" 頻度" と "前"，"後" および "始点" の 4 つのメンバーからなる構造体である．"前" は，頻度が小さい直前のバケットのインデックス（配列要素の番号）を格納し，"後" は頻度が大きい直後のバケットのインデックスを格納するメンバーである．"始点" には，バケットが表す頻度を持つ FS 中の最初の 3 項組の位置（FS 配列内のインデックス）を格納する．配列 BS, FS, HS 内の背景色がついた正整数は全て頻度サマリ FS の要素のインデックスである．背景色がついていない正整数は BS 配列の要素のインデックスである（但し，BS の" 頻度" を除く．" 頻度" 内の数値は出現頻度値である）．また空白の場所は，適当な初期値（例えば -1）が格納されていることを表している．なお解図 8.6 と解図 8.7 では，BS のインデックスは解図 8.5 との対応の理解を容易にするために，下から上へ番号を振ってある．また BS 配列の大きさ（配列要素の最大数）は $\lceil 1/\epsilon \rceil$ とすれば十分である．

頻度サマリ FS は 3 項組を集積した配列であり，配列の基本要素は "3 項組"，"上"，"下"，"親" の 4 つのメンバーからなる構造体である．"上" と "下" は，それぞれ解図 8.5 で上と下に存在する 3 項組の場所を示すインデックスを格納する．"親" は同じ頻度をまとめているバケットの BS 配列内のインデックスを格納している．ハッシュ表である HS 配列は，各アイテムの 3 項組が格納されている FS 配列内の位置（インデックス）を格納している．

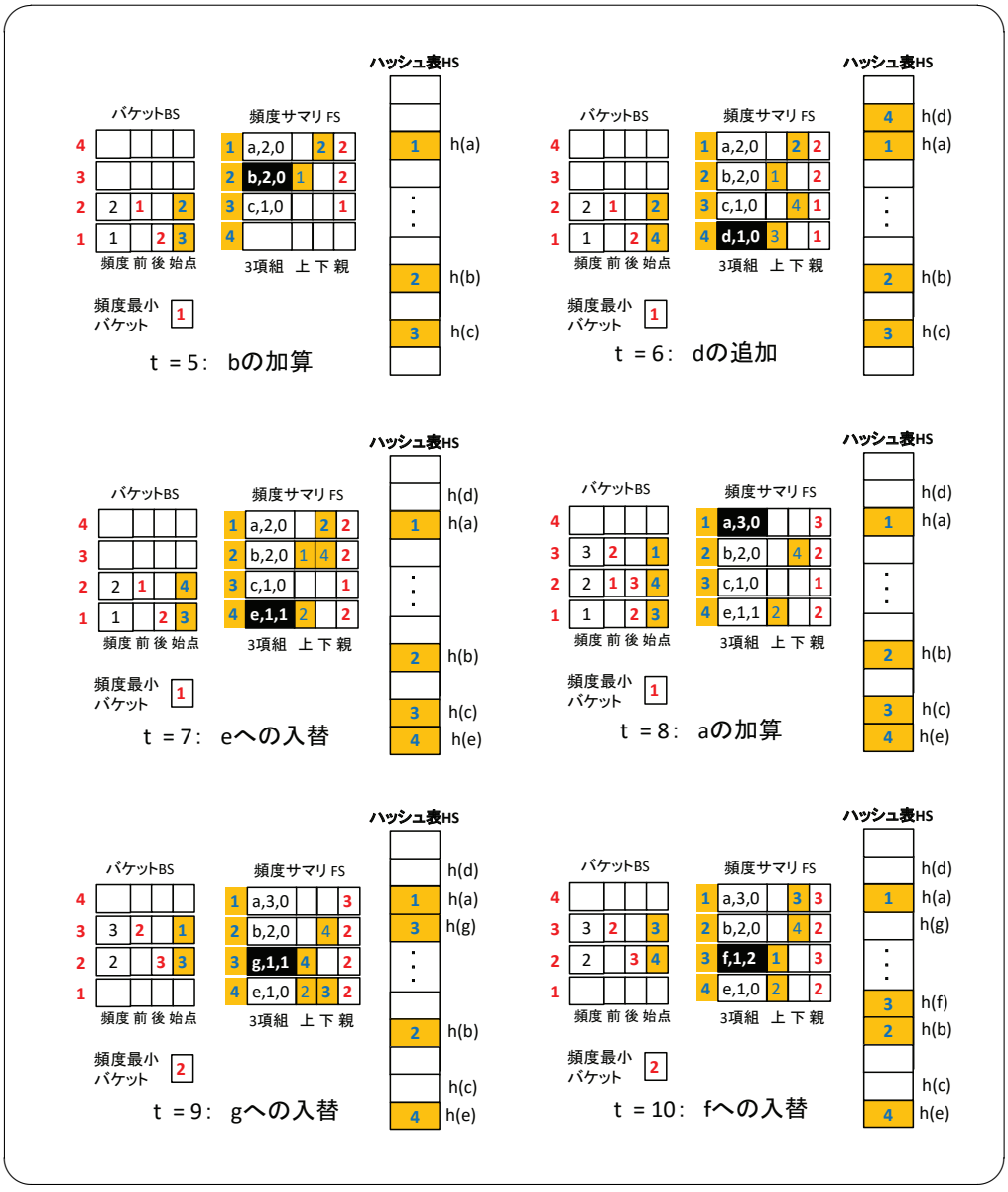
以上を前提とした疑似コードを設計すればよい．その詳細は省略するが，設計に際しての注意点を幾つか述べる．まず (1) の「ストリームから読み込んだアイテム x が頻度サマリ FS に登録済みか否かの判定」は，ハッシュ表 HS の検索そのものである．5 章で学んだ概念と技術を用いて設計して欲しい．

(2) の「アイテム x が FS に登録済みの場合のカウンタ増分操作」では，頻度を増分させたアイテムの 3 項組は所属するバケットを移動させる必要がある．このとき次の 2 点に注意する必要がある．第 1 の注意点 は，頻度が 1 だけ大きいバケットがその時点で存在しない場合がある点である．この場合は新しくバケットを作る必要がある．解図 8.6 と解図 8.7 では，$t = 8$，$t = 12$，$t = 13$，$t = 14$，$t = 16$ の 加算処理がそれに相当する（$t = 5$ と $t = 11$ の加算処理では，既に存在するバケットを利用している）．第 2 の注意点は，頻度が増えた 3 項組を移動させると，元のバケットが空になる場合がある点である．この場合は空のバケットを消去する必要がある．解図 8.6 と解図 8.7 では，$t = 13$ と $t = 16$ の加算処理で，元のバケットの消去が発生している．$t = 13$ と $t = 16$ の加算処理ではバケットの新規作成と消去の両方の作業を行っていることに注意してほしい．

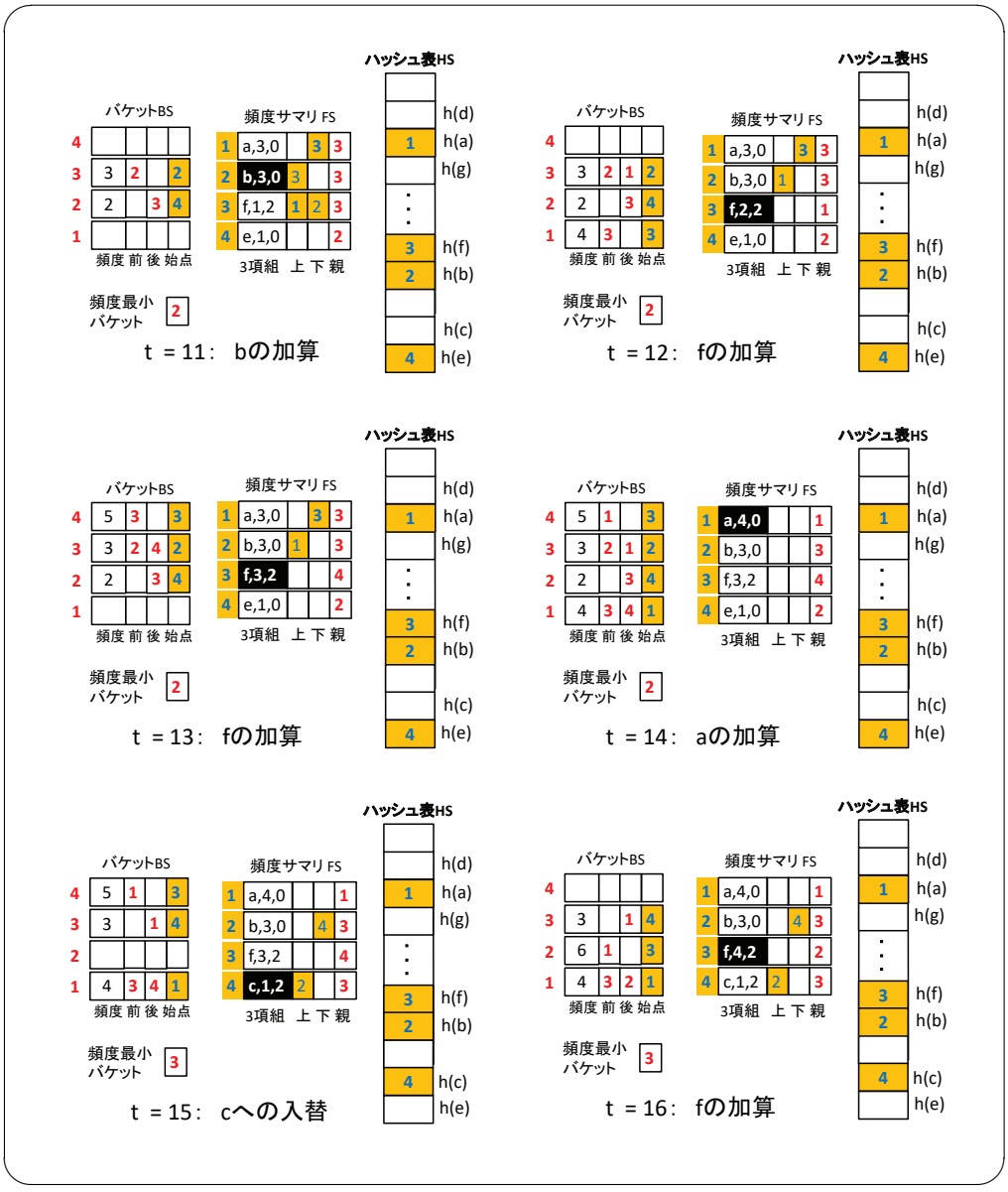
バケット配列 BS の大きさは $\lceil 1/\epsilon \rceil$ に固定しているので，新規バケットを自由に生成するためには，バケットの消去によって空いた配列要素を再利用する必要がある．解図 8.6 と



解図 8.5　ポインタを用いた頻度サマリ



解図 8.6　配列を用いた頻度サマリその１



解図 8.7　配列を用いた頻度サマリその２

解図 8.7 では，インデックスが 1 と 2 の配列要素が再利用されている．再利用を実現するためには，バケットを削除して空になった配列要素のインデックスをキューなどに格納しておけばよい．新たにバケットが必要になったときにキューから取り出して再利用を行う．

(3) の「アイテム x が FS に未登録で，FS に空きが有るときの新規登録操作」は比較的単純な操作である．登録する 3 項組は常に $\langle x, 1, 0\rangle$ であるので，出現頻度 1 に対応するバケットが既にあれば，そこに新規登録する．もし無ければ，新らしく頻度 1 のバケットを作成し，そこに 3 項組を登録を行えばよいだけである．

(4) の「アイテム x が FS に未登録で，FS に空きが無いときの登録操作」は，FS に登録された 3 項組との交換が必要である．まず FS 中の頻度最小のアイテム y を求める．これは「頻度最小バケットの位置を示す変数」からたどることで定数時間で求められる．次に，求めた y の 3 項組 $\langle y, c(y), \Delta(y)\rangle$ を $\langle x, 1, c(y) + \Delta(y)\rangle$ へ置き換える．この際，頻度 $c(y)+\Delta(y)$ のバケットが空になれば，それを消去する．解図 8.6 と解図 8.7 の例では，$t = 9$ と $t = 15$ の入替操作でバケットの消去が起こっている．この場合には，頻度最小のバケットの位置を示す変数の値も変更する必要がある．また解図 8.6 と解図 8.7 では起きていないが，新しく登録する x の頻度 $1 + c(y) + \Delta(y)$ に対応するバケットが BS 配列内に無い場合も想定しておく必要がある．もし対応するバケットが無ければ，新規に作成すればよい．