これまでの復習 (後期中間試験に向けて)

山本昌志*

2005 年 11 月 28 日

これまでの，学習をまとめる。後期中間試験には，このプリントに書いてある内容を理解して臨むこと。授業内容の分かっていない者は，少なくとも 10 時間は集中して勉強すること．どうしても理解できない場合は，成績の良い者に聞くなり，オフィスアワーを利用して，私に質問すること．

1　アルゴリズムとデータ構造

1.1　アルゴリズム

アルゴリズムは，問題を解くための手順を定めたものである．情報科学の分野では，コンピューターを使って問題を解く場合の手順のことをいう．プログラムは，その手順を表現したものである．そのため，アルゴリズムでは曖昧なことは許されず，厳格に手順を決めなくてはならない．そうしないと，プログラムは書けない．

問題を解く場合，いろいろなアルゴリズムが考えられるが，以下のようなものが良いアルゴリズムである．

- 計算の早い方が良いアルゴリズムである．
- 使用するメモリーがすくない方が良いアルゴリズムである．

1.2　データ構造

データ構造 (data structure) とは，データのメモリー上の表現方法のことを言う．諸君は，これまでにいろいろなデータ構造を学習した．まずは，C 言語で用意されている次のようなものである．

- 単純型変数
- 配列
- 構造体

後期の授業では構造体を使ったリストと言われるデータ構造を学習した．これは重要である．

*国立秋田工業高等専門学校　電気情報工学科

2　ソートのアルゴリズム

ソートとは，データをある規則にしたがい順番に並び替えることである．ここでは，ランダムに並んだ整数 (数列) を昇順に並び替えることを学習した．

2.1　バブルソート

2.1.1　原理

数列の隣どうしの要素の大小を比較してそれらを交換しながらソートする方法である．交換が 1 回も生じなかったら，ソートが完了である．これは，小さい値のデータが泡 (バブル；bubble) のように浮かんで行くように見える[1]ことからこの名前がつけられた．

リスト 1 が配列 `sort` に格納された整数を昇順に並び替えるプログラムである．このプログラムの動作を考えるために，データ数を `N` として，`sort[0]` を左端，`sort[N]` を右端とする．すると，以下のことが分かる．

- もっとも値の大きいデータは，外側の 1 回のループ (5 行目の `do`) で右端に移動する．
- 左に移動すべき小さいデータは，このループで一つずつしか移動できない．この一つずつしか移動できないので，整列に多くの計算回数が必要となる．

このプログラムでは，`do` ループ中の 11 行目の

```
if(sort[i]>sort[i+1])
```

がもっとも多く実行される文である．この文の最小実行回数は，最初から整列できている場合である．このときの，実行回数は，$N-1$ 回であるのは明らかである．最悪の場合は，最小の値が右端にある場合である．この場合の実行回数は，$(N-1)^2$ となる．なぜならば，

- この最小の値は，1 回外側のループ (`do` 文のところ) を実行する毎に，一つ左に移動する．
- 所定の位置に移動するために，$N-1$ 回，外側のループを実行する必要がある．
- 外側のループを 1 回実行する毎に，内側のループは $N-1$ 回，実行される．

となるからである．

最初のデータの並び方により，$N-1$ 回から $(N-1)^2$ まで開きがある．初期位置と整列終了位置との差の最大の期待値を計算することになるが，それは $(N-2)/2$ 程度であることは分かるだろう．最小値の初期位置と整列終了位置との差の期待値である．この程度とすると，先の比較の実行回数は，

$$\frac{(N-1)(N-2)}{2} = \frac{N^2}{2} - \frac{3}{2}N + 1 \tag{1}$$

となる．データ数 N が大きい場合，N^2 に比例して計算量が増加する．

この N^2 に比例して計算量が必要な場合，$O(N^2)$ と書く．この表記法を O **記法**(O-notation) といい，計算オーダーを示す．数学で使うランダウの記号に似ている．バブルソートの計算のオーダーは，$O(N^2)$ である．

[1] 昇順にソートする場合．

2.1.2 プログラム

教科書の List 1-1(p.3-4) のバブルソートの関数は，リスト 1 の通りである．これを main 関数から呼び出すことにより，配列 sort に格納された整数が昇順に並び替えられる．

- 配列 sort はグローバル変数[2]なので，関数の引数として渡していない．
- 数列の個数 N は，#define で与えられる．
- 入れ替えが生じると，flag の値が 1 になる．入れ替えが起きないときは，flag の値は 0 である．これはフラグ (flag:旗) と呼ばれるもので，計算の状態を表す．

リスト 1: バブルソートの関数

```
1  void BubbleSort(void)
2  {
3      int      i,j,flag;
4
5      do
6      {
7          flag=0;
8          for(i=0;i<N-1;i++)
9          /* 先頭から順に見ていって */
10         {
11             if(sort[i]>sort[i+1])
12             /* 左右の並びがおかしければ */
13             {
14                 /* 入れ替える */
15                 flag=1;
16                 j=sort[i];
17                 sort[i]=sort[i+1];
18                 sort[i+1]=j;
19             }
20         }
21         /* 1度も並べ替えをせずに見終わったら終了 */
22     }   while(flag==1);
23 }
```

2.2 クイックソート

2.2.1 原理

ある一つの要素を基準とし基準値より大きいグループと小さいグループに分ける．分けられたグループも同様の方法で分ける．全てのグループの要素が 1 つになるまでこの作業を繰りかえし，このときソートが完了する．いろいろなソートの中で，平均的には最速であるから，この名前がついている．

2.2.2 プログラム

教科書の List 1-3(p.8-9) のクイックソートの関数は，リスト 2 の通りである．これを main 関数から呼び出すことにより，配列 data に格納された整数が昇順に並び替えられる．

[2]関数の外で宣言されたので，どの関数からでもアクセスできる．

- 関数の引数は，ソートする配列の両端 (bottom, top) と配列を表すポインター (*data) である．
- 配列のポインター (*data) は，配列名として使える．

リスト 2: クイックソートの関数

```
1  void QuickSort(int bottom,int top,int *data)
2  {
3      int lower,upper,div,temp;
4      if(bottom>=top)
5          return;
6      /* 先頭の値を「適当な値」とする */
7      div=data[bottom];
8      for(lower=bottom,upper=top;lower<upper;)
9      {
10         while(lower<=upper&&data[lower]<=div)
11             lower++;
12         while(lower<=upper&&data[upper]>div)
13             upper--;
14         if(lower<upper)
15         {
16             temp=data[lower];
17             data[lower]=data[upper];
18             data[upper]=temp;
19         }
20     }
21     /* 最初に選択した値を中央に移動する */
22     temp=data[bottom];
23     data[bottom]=data[upper];
24     data[upper]=temp;
25
26     QuickSort(bottom,upper-1,data);
27     QuickSort(upper+1,top,data);
28 }
```

2.3 マージソート

2.3.1 原理

最初に少ない個数の数列を並べ替えたグループを作る．次に，2 つのグループを併合 (マージ) して，より大きな並び替えられたグループを作る．これを繰り返すことにより，大きなグループができ，最終的には一つのグループになり並び替えが終了する．

2.3.2 プログラム

リスト 3: マージソートの関数

```
1  void MergeSort(int n,int x[])
2  {
3      int      i,j,k,m;
4      if(n<=1)
5          return;
6      m=n/2;
```

```
7
8       /* ブロックを前半と後半に分ける */
9       MergeSort(m,x);
10      MergeSort(n-m,x+m);
11
12      /* 併合（マージ）操作 */
13      for(i=0;i<m;++i)
14          buffer[i]=x[i];
15      j=m;
16      i=k=0;
17      while(i<m&&j<n)
18      {
19          if(buffer[i]<=x[j])
20              x[k++]=buffer[i++];
21          else
22              x[k++]=x[j++];
23      }
24      while(i<m)
25          x[k++]=buffer[i++];
26  }
```

2.4 コームソート

2.4.1 原理

バブルソートは隣との比較のため，小さい値は一つずつしか移動できない (昇順)．比較する間隔を広くして，一気に移動させる方法がコームソートである．このソートでは，適当な間隔でバブルソートを行い徐々にその間隔を狭める方法がとられる．実験から，間隔は 1/1.3 ずつ小さくしていくのが良いと言われている．最終的には隣との比較 (間隔が 1) を行い，交換が起こらなければソートの完了である．

2.4.2 プログラム

リスト 4: マージソートの関数

```
1   void CombSort(void)
2   {
3       int i,temp,flag,gap;
4       gap=N;
5
6       do
7       {
8           gap=(gap*10)/13;
9           /* 収縮率は1.3（gapが毎回1/1.3になる） */
10          if (gap==0)
11              gap=1;
12
13          flag=1;
14          /* 先頭から順に見ていって */
15          for (i=0; i<N-gap; ++i)
16          {
17              /* 距離がgapだけ離れた要素を比較し，
18                  並びがおかしければ */
19              if (sort[i]>sort[i+gap])
20              {
```

```
21                      /* 入れ替える */
22                      flag=0;
23                      temp=sort[i];
24                      sort[i]=sort[i+gap];
25                      sort[i+gap]=temp;
26                  }
27              }
28
29      /* 1度も並べ替えをせずに，gap=1で見終わったら終了 */
30      }   while((gap>1) || flag!=1);
31  }
```

2.5 単純挿入ソート

2.5.1 原理

まず，最初の数列の 2 つを比較して，小さいほうを左に，大きいほうを右にする．次に数列の 3 番目を，その左にある 2 つの数列と小さいほうから順に比較し，収まる位置を探索して，挿入する．これを，4 番目,5 番目，··· と順次，数列の最後まで繰り返す．最後の数列が挿入されたらソートが完了である．

2.5.2 プログラム

リスト 5: 単純挿入ソートの関数

```
1   void MergeSort(int n,int x[])
2   {
3       int     i,j,k,m;
4       if(n<=1)
5           return;
6       m=n/2;
7
8       /* ブロックを前半と後半に分ける */
9       MergeSort(m,x);
10      MergeSort(n-m,x+m);
11
12      /* 併合（マージ）操作 */
13      for(i=0;i<m;++i)
14          buffer[i]=x[i];
15      j=m;
16      i=k=0;
17      while(i<m&&j<n)
18      {
19          if(buffer[i]<=x[j])
20              x[k++]=buffer[i++];
21          else
22              x[k++]=x[j++];
23      }
24      while(i<m)
25          x[k++]=buffer[i++];
26  }
```

2.6 2分挿入ソート

2.6.1 原理

単純挿入ソートでは，数列のある値を挿入する適当な位置は端から順に探している (リニアーサーチ)，位置を探す数列は並べ替えられているので，真中の値から比較するバイナリーサーチが可能で，それを適用して速度の向上を図った方法が，2 分挿入ソートである．

2.6.2 プログラム

リスト 6: 2 分挿入ソートの関数

```
1  void BinaryInsertSort(void)
2  {
3      int i,sorted,temp,insert;
4      int left,mid,right; /* バイナリサーチ用の追加変数 */
5
6      /* 最初から最後まですべてソート済みになるまで繰り返す */
7      for(sorted=1; sorted<N; ++sorted)
8      {
9          /* ソート済み領域の直後の値を取り出す */
10         insert=sort[sorted];
11
12         /* 挿入する場所を見つける（バイナリサーチ） */
13         left=0;
14         right=sorted;
15         while(left<right)
16         {
17             mid=(left+right)/2;
18
19             if(sort[mid]<insert)
20                 left=mid+1;
21             else
22                 right=mid;
23         }
24         i=left;
25
26         /* ソート済み領域直後の値を挿入する
27              （単純挿入ソートと同じ） */
28         while(i<=sorted)
29         {
30             temp=sort[i];
31             sort[i]=insert;
32             insert=temp;
33             ++i;
34         }
35     }
36 }
```

2.7 ソートのアルゴリズムの比較

ソートのアルゴリズムの評価に計算量の他に，安定性というものが使われる．

- 安定なソートとは，同じ要素があったとき，それらを入れ替えない．

- 不安定なソートとは，同じ要素があったとき．それらを入れ替える可能性がある．

これら学習したソートの計算量と安定性については，表 1(教科書の Table 1-2) のとおりである．

表 1: ソートのアルゴリズムの特徴

アルゴリズム	計算量オーダー	安定性
バブルソート	$O(N^2)$	安定
クイックソート	$O(N \log N)$ ～$O(N^2)$	不安定
マージソート	$O(N \log N)$	安定
コームソート	$O(N \log N)$	不安定
単純挿入ソート	$O(N^2)$	安定
2 分挿入ソート	$O(N^2)$	安定

3 サーチのアルゴリズム

サーチとは，

> 文字どおりたくさんのデータの中から目的のデータ (キー) がどこにあるか (もしくは，あるかないか) を調べる作業

である．整数のデータが配列に格納されている場合について，目的のデータを探す方法を学習した．

3.1 リニアサーチ

3.1.1 原理

目的のデータを端から順に探索する方法である．データがランダムに並んでいる場合，この方法しかない．

このアルゴリズムの計算のオーダーは，$O(N)$ である．なぜならば，端から探索していくため，データが一致するためには平均 $N/2$ 回の比較が必要であるからである[3]．

3.1.2 プログラム

教科書の List 2-1(p.37-38) のリニアサーチの関数は，リスト 7 の通りである．これを `main` 関数から呼び出すことにより，整数 `x` と一致する配列 `a` の添え字の番号が求められる．

- `x` が探索するデータである．
- 配列はポインターとして渡されている．`main` 関数の実引数の配列名は，ポインターである．この関数では，`main` 関数の呼び出し側の配列は，`a[]` という配列になる．

[3]目的のデータが探索すべき列の中にある場合

- num がデータ数を表す.
- 目的のデータと一致するものが無かった場合，NOT_DOUND を返す．この値は，#define で置換される．

リスト 7: リニアサーチの関数

```
1  int linear_search(int x,int *a,int num)
2  {
3      int n=0;
4
5      /* 配列の範囲内で目的の値を探す */
6      while(n<num&&a[n]!=x)
7          n++;
8
9      if(n<num)
10         return n;
11
12     return NOT_FOUND;
13 }
```

3.2 番兵をつかったリニアサーチ

3.2.1 原理

リスト 7 の計算速度を上げるために，通常のリニアサーチでは番兵をつかう．リスト 7 では，1 回のループで，

```
n<num&&a[n]!=x
```

のように 2 回の比較 (n<num と a[n]!=x) を行っている．目的のデータ (キー) を配列の最後に入れる (番兵) ことにより，n<num の比較の計算を行わないで済むようにできる．こうすると配列の最後では，a[n]!=x] が成立しなくなるので，ループから抜けることになる．ループから抜けた後，キーと元々あった配列の最後の値と比較すればよいのである．

このようにすると，比較の数が半分になる．しかし，計算量のオーダーは $O(N)$ と変わらないことに注意しよう．

3.2.2 プログラム

教科書の List 2-2(p.38-39) の番兵を用いたリニアサーチの関数は，リスト 8 の通りである．引数は，先ほどの番兵を用いないリニアサーチと同じ．

リスト 8: 番兵をつかったリニアサーチの関数

```
1  int search(int x,int *a,int num)
2  {
3      int      n=0,t;
4
5      /* 最後の値を x に入れ替える（番兵） */
6      t=a[num-1];
```

```
 7       a[num-1]=x;
 8
 9       /*目的の値を探す*/
10       while(a[n]!=x)
11           n++;
12
13       a[num-1]=t;        /* 配列最後の値を元に戻す */
14       if(n<num-1)
15           return n;      /* いちばん最後以外で一致 */
16       if(x==t)
17           return n;      /* いちばん最後が一致 */
18
19       return NOT_FOUND;
20   }
```

3.3 バイナリーサーチ

3.3.1 原理

データの並びに規則性が無い場合，リニアサーチのように端から順に捜すしかない．しかし，データの並びに規則性がある場合，その性質を利用できる．

たとえば，データが昇順に並んでいた場合，端から比較するのではなく，最初に真ん中に位置するデータと比較する．すると，サーチすべきデータの個数が 1 回の比較でに半分になる．同じことを繰り返すと，比較すべき対象が 1/2 ずつ減少する．データの個数が多い場合，この方法は劇的にサーチが早くなる．この方法をバイナリーサーチと言う．

リニアサーチだと，1 回の比較で 1 個しかデータの候補が減らないのに，バイナリーサーチだと半分に減少させることができるのである．ただし，バイナリーサーチを使うためには，データを予めソートしておく必要がある．サーチの回数が 1 回であれば，ソートの手間を考えるとリニアサーチの方が良い．サーチの回数が増加すると，バイナリーサーチの方が有利になる．

1 回の計算，リスト 9 の 5〜15 行のループで，検索する範囲は 1/2 になる．したがって，α 回のループでその範囲が 1 になれば計算が終了する．データの個数を N として，式で表すと，

$$N\left(\frac{1}{2}\right)^{\alpha} = 1 \tag{2}$$

となる．これから，計算回数 α は，

$$\alpha = \log_2 N \tag{3}$$

と導き出せる[4]．バイナリーサーチの場合の計算量のオーダーは，$O(\log_2 N)$ である．

3.3.2 プログラム

教科書の List 2-3(p.41-42) のバイナリーサーチの関数は，リスト 9 の通りである．これを `main` 関数から呼び出すことにより，整数 `x` と一致する配列 `a` の添え字の番号が求められる．

[4] ここの計算は，$\frac{1}{2}^{\alpha} = \frac{1}{N} \quad \Rightarrow \quad 2^{\alpha} = N \quad \Rightarrow \quad \log_2 2^{\alpha} = \log_2 N \quad \Rightarrow \quad \alpha \log_2 2 = \log_2 N \quad \Rightarrow \quad \alpha = \log_2 N$

- 配列はポインターとして渡されている．main 関数の実引数の配列名は，ポインターである．
- left がデータが存在する左端で，right が右端を表す．
- 目的のデータと一致するものが無かった場合，NOT_DOUND を返す．この値は，#define で置換される．

リスト 9: バイナリーサーチの関数

```
1  int binary_search(int x,int *a,int left,int right)
2  {
3      int      mid;
4
5      while(left<=right)
6      {
7          mid=(left+right)/2;
8          if(a[mid]==x)
9              return mid;
10
11         if(a[mid]<x)
12             left=mid+1;      /* midより左側にxは存在しない */
13         else
14             right=mid-1;     /* midより右側にxは存在しない */
15     }
16
17     /* サーチ範囲がなくなっても一致するものはなかった */
18     return NOT_FOUND;
19 }
```

4 リスト

4.1 原理

リストは前後のデータのある位置をメモリーに入れることにより，データの列 (ここでは数列) を構成する．最初のデータの位置を元に，それから順にたどりすべてのデータをつなぎデータ列 (数列) を構成する．この様子を図 1 に示す．

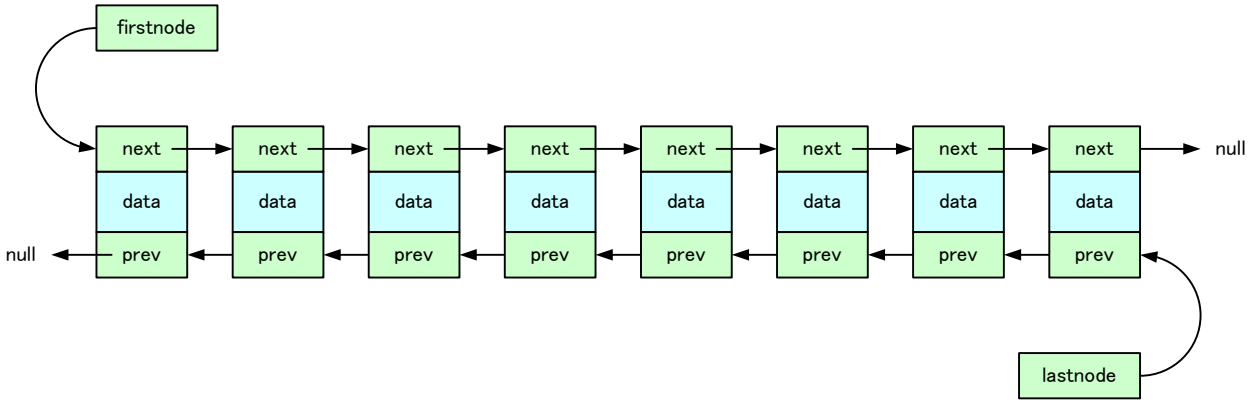

図 1: リスト

このようにすると，ここのデータは順にアクセス (シーケンシャルアクセス) することになり，その速度は一般に遅い．一方，データの削除と挿入は，データの位置を記憶するポインターを変更するだけなので，高速になる．

4.2 プログラム

教科書の List 3-3(p.75-77) では，リスト 10 に示す構造体を用いてリストを作っている．

- `*prev` が前のノードを示すポインターである．
- `*next` が次のノードを示すポインターである．
- `*data` がこのノードのデータである．

リスト 10: リストを表す構造体

```
1 typedef struct tagListNode
2 {
3     struct tagListNode *prev;     /* 前の要素へのポインタ */
4     struct tagListNode *next;     /* 次の要素へのポインタ */
5     int data;    /* この要素がもっているデータ */
6 } ListNode;
```

4.3 リストと配列の違い

配列もリストも，複数のデータの値と順序を記憶するデータ構造である．しかし，その使い方はかなり異なり，その性質をよく理解しなくてはならない．

配列は目的のデータにランダムアクセスが可能で，目的のデータの値を得たり，データの値の変更が高速にできる．添え字を指定するだけで，それらが可能である．しかし，データの削除と挿入には計算回数が多

くなる．たとえば，10 番目のデータを削除するとなると，11 番目を 10 番目に移動，12 番目を 11 番目に移動，13 番目を 12 番目に移動 · · · とデータを順次移動させる必要がある．

一方，リストは目的のデータにアクセスするためには，シーケンシャルアクセス[5]を行う．そのため，データへのアクセス回数が多くなり配列より低速になる．しかし，データの削除と挿入は簡単で，その前後へのノードのポインターの値を変更するだけである．したがって，挿入と削除は高速になる．

表 2: 配列とリストとの違い

	配列	リスト
データへのアクセス	添え字によるランダムアクセス可能	リストを順にたどる
アクセスのための計算量	$O(1)$	$O(N)$
データの挿入/削除	計算コスト大 $O(N)$	計算コスト小 $O(1)$
メモリーのコスト	小	配列より大

5　C 言語プログラムテクニック

教科書の List 3-2(p.72) では，C 言語のプログラムでよく使われるメモリーの確保と配列に関するテクニックが書かれている．また，List 3-4(p.75-77) では型定義という方法が使われている．これらは重要なので，よく理解すること．

- `array=(int *)malloc(sizeof(int)*count);`

  少々複雑に見えるが，処理される順に見ていけば簡単である．

  1. 最初に `sizeof(int)` が評価される．関数 `sizeof()` は，型のバイト数を返す．ここでは，整数型 `int` のバイト数である 4 が返される．
  2. 次に，この 4 とデータ数である `count` の乗算が行われる．この結果は，データを格納するために必要なバイト数の計算になっている．
  3. `malloc()` 関数は，*Memory ALLOCate*(記憶割付) の略で，引数で渡されたバイト数のメモリーを確保して，その先頭のポインター (アドレス) を返す．
  4. `(int *)` はキャストと呼ばれる強制型変換である．`malloc` で返されるポインターは強制的に整数型としている．従って，ポインターを+1 加算すると，アドレスは 4 つ進むことになる．
  5. 整数型のポインター (アドレス) を整数型のポインター `array` に代入している．

- `free(array);`

  関数 `free()` は，そのプログラムで使用しているメモリー領域を開放するためにある．ここでは，`malloc` で確保された領域を開放している．

- `int *array` と `array[n]`

[5]データを先頭から順番に読み込み、あるいは書き込みを行なう方法。

ポインターで宣言しているものを配列として使っている．これは，配列がどのように処理されているか，考えればよく分かる．配列 array[n] は，*(array+n) と評価される．すなわち，配列 array[n] は，ポインター array に n 加算したポインター (アドレス) が示す値と言うことである．このようなことから，配列で宣言していなくても，配列のように使うことができるのである．

- typedef struct tagListNode{
  ```
  struct tagListNode *prev;
  struct tagListNode *next;
  int data;
  ```
  } ListNode;

  これは，*TYPE DEFine*(型定義) と呼ばれるもので，型に別名をつける役割がある．ここでは，struct tagListNode と言う型を，ListNode という別名で使える．このことにより，宣言が簡単になる．

- lastnode=NULL

  これは，ヌルポインター，あるいは空ポインターと呼ばれるものである．これは，lastnode が何も指し示していないことを保証するために，使っている．

- newnode->data=buf;

  newnode はポインターである．ポインターが指し示す構造体のメンバーにアクセスする場合，アロー演算子 (->) を使う．ここでは，ポインター newnode が示す構造体のメンバー data に buf の値を代入している．ポインターではなく，普通の変数となっている構造体の場合，そのメンバーにアクセスするにはドット演算子 (.) を使う．

6 合格点を取るためには

- ここで学習したアルゴリズムを用いて，10 個程度の整数がソートできること．
- 少なくても，バブルソートとクイックソートのプログラムを理解すること．
- 計算量を示す O **記表**を理解すること．バブルソートとバイナリーサーチの計算量のオーダーについて，説明できること．
- リニアサーチとバイナリーサーチのプログラムが理解できること．
- リニアサーチにおける番兵の役割とそれを実現するプログラムを理解すること．
- リストを作成するための構造体がか書けること．
- リストと配列の違いが理解できること．
- ここで学習した C 言語のプログラムテクニックが理解できること．