第8回　検索とその効率

～効率の良いプログラムとは？～

学習目標

- 効率が考慮されたプログラムが書ける
  - 効率の悪いプログラムを使うとどうなるか説明できる
  - 効率を指標（BigO 記法）を使って説明できる
  - バイナリサーチを実装できる
- 簡単なクラス設計について議論できる

8.1.　プログラムと効率

8.1.1.　プログラムの効率が悪いと、どうなる？

一般的に品質の高いソフトウエアの条件には、以下の要素があると言われています。

- 正しさ
- 使いやすさ
- 再利用性
- 可読性（読みやすさ）
- 効率（速さ）

では、プログラムの効率が悪かったらどうなるか、考えてみましょう。

- １万人の社員がいる会社があります。
- ログインするためには、社員番号でパスワードを検索します。
- １人の検索に０．１秒かかります。

では、全員がログインし終わるまでに何分かかりますか？

答え：

今回は、以下のようなたくさんの商品種類を扱ったらどうなるか、そうした問題を解決するにはどうしたら良いか、を考えていきたいと思います。

１．商品種類を１万個、追加します
２．商品種類を１万回、検索します

１．商品種類を１０万個、追加します
２．商品種類を１０万回、検索します

8.1.2.　性能を測るプログラム

①.StopWatch クラスを作ろう

性能比較は、第 7 回においても行いましたが、今回は、プログラムの意味を明確にして、読みやすいプログラムを目指して時間を計るためのクラスを作ります。StopWatch クラスを以下に示します。

例題 8-1：ストップウォッチクラスを使う(StopWatch.java)

```
1:      /**
2:      * オブジェクト指向哲学　入門編
3:      * 例題 8-1：ストップウォッチクラスを使う
4:      * 追加と検索の性能を比較するプログラム
5:      *
6:      * ストップウォッチクラス
7:      * ある処理の開始から終了までの時間をはかるためのクラス
8:      */
9:      public class StopWatch {
10:
11:       long startTime;//開始した時間を保存する変数
12:       long stopTime;//停止した時間を保存する変数
13:
14:       /**
15:       * ストップウオッチをスタートさせる
16:       */
17:       public void start(){
18:         startTime = System.currentTimeMillis();//開始した時間を保存しておく
19:       }
20:
21:       /**
22:       * ストップウオッチを停止させる
23:       */
24:       public void stop(){
25:         stopTime = System.currentTimeMillis();//停止した時間を保存しておく
26:       }
27:
28:       /**
29:       * 開始から終了までにかかった時間を取得する
30:       */
31:       public long getTime(){
32:         long time = stopTime - startTime;//開始時間-停止時間をかかった時間とする
33:         return time;
34:       }
35:     }
```

(1)start( )メソッドと stop( )メソッド

```
public void start(){
  //開始時間を保存しておく
  startTime = System.currentTimeMillis();
}
```

System.currentTimeMillis()メソッドは、現在の時刻を long型で返すためのメソッドです。具体的には、「１９７０年１月１日午前０時０分０秒」を０として、現在時刻との差をミリ秒単位で表したものが返ってきます。

例えばパソコンの内部時計が「１９７０年１月１日午前０時１分０秒」を指している瞬間にこのメソッドを呼び出すと「６００００」が返ってきます。これは６０秒をミリ秒単位で表現するので、６０×１０００＝６００００となるからです。

start( )メソッドと stop( )メソッドは、メソッドが呼び出された時点での現在時刻をそれぞれ startTime、stopTime に保存します。

(2)getTime( )メソッド

```
public long getTime(){
  //開始時間-停止時間をかかった時間とする
  long time = stopTime - startTime;
  return time;
}
```

stopTime と startTime の差を取ってそれを返します。その結果、start()メソッドが呼び出された時の時刻から stop()メソッドが呼び出された時の時刻までに、どれくらいの時間が経過したのかを知ることができます。

②.StopWatch クラスを使って処理時間を計測するプログラム

例題 8-1：ストップウォッチクラスを使う(Example8_1.java)

```
1:      /**
2:      * オブジェクト指向哲学　入門編
3:      * 例題 8-1：ストップウォッチクラスを使う
4:      * 追加と検索の性能を比較するプログラム
5:      *
6:      * メインクラス
7:      */
8:      public class Example8_1 {
9:
10:       /**
11:       * メイン
12:       * 1 万件の種類をリストに追加して、その時間を計る
13:       */
14:       public static void main(String[] args) {
15:
16:         int addItemTypeNum = 10000;//商品種類を追加/検索する回数
17:         long startTime;//開始時間を保存する
18:         long endTime;  //終了時間を保存する
19:         long time;//計測した時間
20:
21:         //商品種類リストを生成する
22:         ItemTypeList itemTypeList = new ItemTypeList();
23:
24:         //ストップウォッチを生成する
25:         StopWatch stopWatch = new StopWatch();
26:
27:         //-------追加の性能を測定する-------
28:
29:         stopWatch.start();//ストップウォッチをスタートさせる
30:
31:         //商品種類を 1 万件追加する
32:         for(int i=0; i<addItemTypeNum; i++){
33:           itemTypeList.add(new ItemType(i,"コーラ",120));//商品番号 i の商品を追加する
34:         }
35:
36:         stopWatch.stop();//ストップウォッチを停止させる
37:
38:         time = stopWatch.getTime();//計測した時間を取得する
39:         System.out.println("追加にかかった時間は"+ time + "ミリ秒です");//計測した時間
を表示する
40:
41:         //-------検索の性能を測定する-------
42:         boolean result;   //検索結果
43:         stopWatch.start();//ストップウォッチをスタートさせる
44:
```

```
45:         //商品種類を指定された件数検索する
46:         for(int i=0; i<addItemTypeNum; i++){
47:           result=itemTypeList.search(i);//商品番号 i の商品を検索する
48:           if(result == false){
49:             System.out.println("見つかりませんでした");
50:           }
51:         }
52:
53:         stopWatch.stop();//ストップウォッチを停止させる
54:
55:         time = stopWatch.getTime();//計測した時間を取得する
56:         System.out.println("検索にかかった時間は"+ time + "ミリ秒です");//計測した時間
を表示する
57:
58:       }
59:     }
```

今回のプログラムは、効率に注目するために、第 7 回での継承のソースとは異なり、ItemTypeList クラスとして、配列リストが利用されています。第 7 回のプログラムとは異なるので注意してください。ItemTypeList クラスを示します。

例題 8-1：ストップウォッチクラスを使う(ItemTypeList.java)

```
1:      /**
2:      * オブジェクト指向哲学　入門編
3:      * 例題 8-1：ストップウォッチクラスを使う
4:      * 追加と検索の性能を比較するプログラム
5:      *
6:      * 商品種類リストクラス
7:      * リニアサーチによって商品種類を検索することができる
8:      */
9:      public class ItemTypeList {
10:
11:       private int ARRAY_SIZE = 10000;                                    //用意する配列の大
きさ
12:       private ItemType[] itemTypeArray = new ItemType[ARRAY_SIZE];  //商品種類を保存す
るための配列
13:
14:       /**
15:       * 商品種類を追加する
16:       */
17:       public void add(ItemType addItemType){
18:         //商品種類が入っていない箱を探す
19:         for(int i=0; i<ARRAY_SIZE; i++){
20:           if(itemTypeArray[i] == null){//入っていない
21:             itemTypeArray[i] = addItemType;//書き込む
22:             break;
23:           }
```

```
24:         }
25:       }
26:
27:       /**
28:       * 商品種類を検索する
29:       * (リニアサーチ)
30:       */
31:       public boolean search(int searchID){
32:         //配列に格納されている商品種類と検索対象のデータを一つ一つ照らし合わせて検索
する
33:
34:         for(int i=0; i<ARRAY_SIZE; i++){//配列の各要素に対して繰り返す
35:           if(itemTypeArray[i]!=null){
36:             //配列の要素が空ではなかったとき
37:             if(itemTypeArray[i].getId() == searchID){
38:               //見つかった
39:               return true;
40:             }
41:           }
42:         }
43:
44:         //見つからなかった
45:         return false;
46:       }
47:
48:     }
```

8.1.3.　効率の計測結果についての考察

①.計測結果

StopWatch クラスを使って計測した結果はどうだったでしょうか。

	処理に要した時間
1 万要素追加	
1 万要素検索	
2 万要素追加	
2 万要素検索	
5 万要素登録	
5 万要素検索	
10 万要素登録（行わないほうが賢明）	

②.考察

```
/**
* 商品種類を追加する
*/
public void add(ItemType addItemType){
  //商品種類が入っていない箱を探す
  for(int i=0; i<ARRAY_SIZE; i++){
    if(itemTypeArray[i] == null){//入っていない
      itemTypeArray[i] = addItemType;//書き込む
      break;
    }
  }
}
```

add メソッドを見てみましょう。このメソッドの中に if 文がありますが、この if 文は何回実行されているでしょうか。

	if 文の実行回数
1 要素追加	回
2 要素追加	＋　＝　回
3 要素追加	＋　＋　＝　回
100 要素追加	回
N 要素追加	回
1 万要素追加	回
10 万要素追加	回

8.1.4. 追加の効率を上げる

追加に時間がかかるのは、if 文を異常なほどたくさん実行する必要があったからです。そこで add メソッドを変更して、if 文を使わなくてもいいように変更してみましょう。

これまでは追加する際に、追加する場所をいちいち探していたために処理に時間がかかっていました。そこで今度は「追加するべき位置」を覚えるような仕様に変更しましょう。

例題 8-2：追加の効率を上げる(ItemTypeList.java)

```
1:      /**
2:      * オブジェクト指向哲学　入門編
3:      * 例題 8-2：追加の効率を上げる
4:      * 追加と検索の性能を比較するプログラム
5:      *
6:      * 商品種類リストクラス
7:      * リニアサーチによって商品種類を検索することができる
8:      * 現在の要素数を覚えておくことによって追加の効率化を図る
9:      */
10:     public class ItemTypeList {
11:
12:       private int ARRAY_SIZE = 10000;                                    //用意する配列の大
きさ
13:
14:       private int size=0;                                                //現在配列に保存さ
れている要素数
15:
16:       private ItemType[] itemTypeArray = new ItemType[ARRAY_SIZE];  //商品種類を保存す
るための配列
17:
18:
19:       /**
20:       * 商品種類を追加する
21:       */
22:       public void add(ItemType addItemType){
23:         itemTypeArray[size] = addItemType;//空の箱から順に書き込む
24:         size++;//要素数を 1 つ増やす
25:       }
26:
27:       /**
28:       * 商品種類を検索する
29:       * (リニアサーチ)
30:       */
31:       public boolean search(int searchID){
32:         //配列に格納されている商品種類と検索対象のデータを一つ一つ照らし合わせて検索
する
33:
34:         for(int i=0; i<size; i++){//配列の各要素に対して繰り返す
35:           if(itemTypeArray[i].getId() == searchID){
```

```
36:             //見つかった
37:             return true;
38:           }
39:         }
40:
41:         //見つからなかった
42:         return false;
43:       }
44:
45:     }
```

このような変更を行った上で、もう一度それぞれの処理速度を計測してみてください。

	処理に要した時間
1 万要素追加	
1 万要素検索	
2 万要素追加	
2 万要素検索	
5 万要素登録	
5 万要素検索	
10 万要素登録（行わないほうが賢明）	

8.2.　検索アルゴリズム

8.2.1.　検索の効率を考える

先ほどの変更により追加は早くなりましたが、検索はまだ遅いままです。そこで検索メソッドを見てみましょう。

```
public boolean search(int searchId){
  //一つ一つ商品種類を探す
  for(int i=0;i<size;i++){
    if(itemTypeArray[i].getNo() == searchId){//見つかった
      return true;
    }
  }
  //見つからなかった
  return false;
}
```

このメソッドの中で if 文は何回実行されているでしょうか？

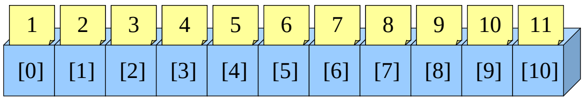
1 万要素から検索する場合を考えてみましょう。

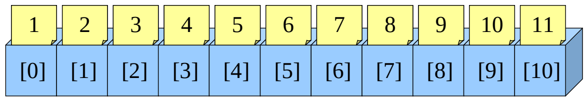
	if 文の実行回数
0 を検索する	回
1 を検索する	回
2 を検索する	回
9999 を検索する	回

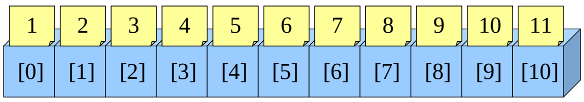
また、見つからない場合を考えるとどうなるでしょうか？

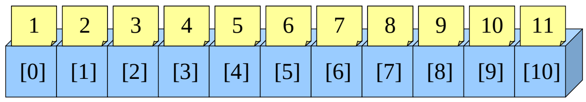
8.2.2.　　　リニアサーチ

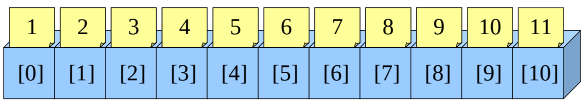
　これまで検索に使っていたアルゴリズムを「リニアサーチ」と言います。例えば９を検索する場合はどのような順番に配列をサーチするでしょうか。書き込んでみてください。

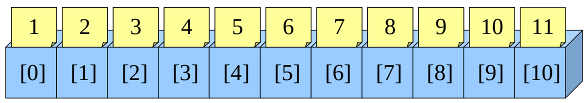


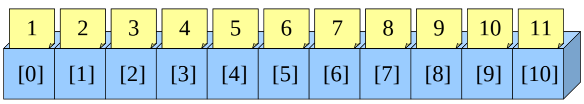


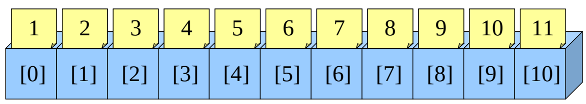


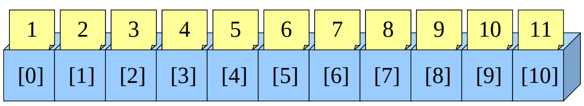


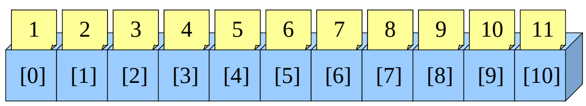








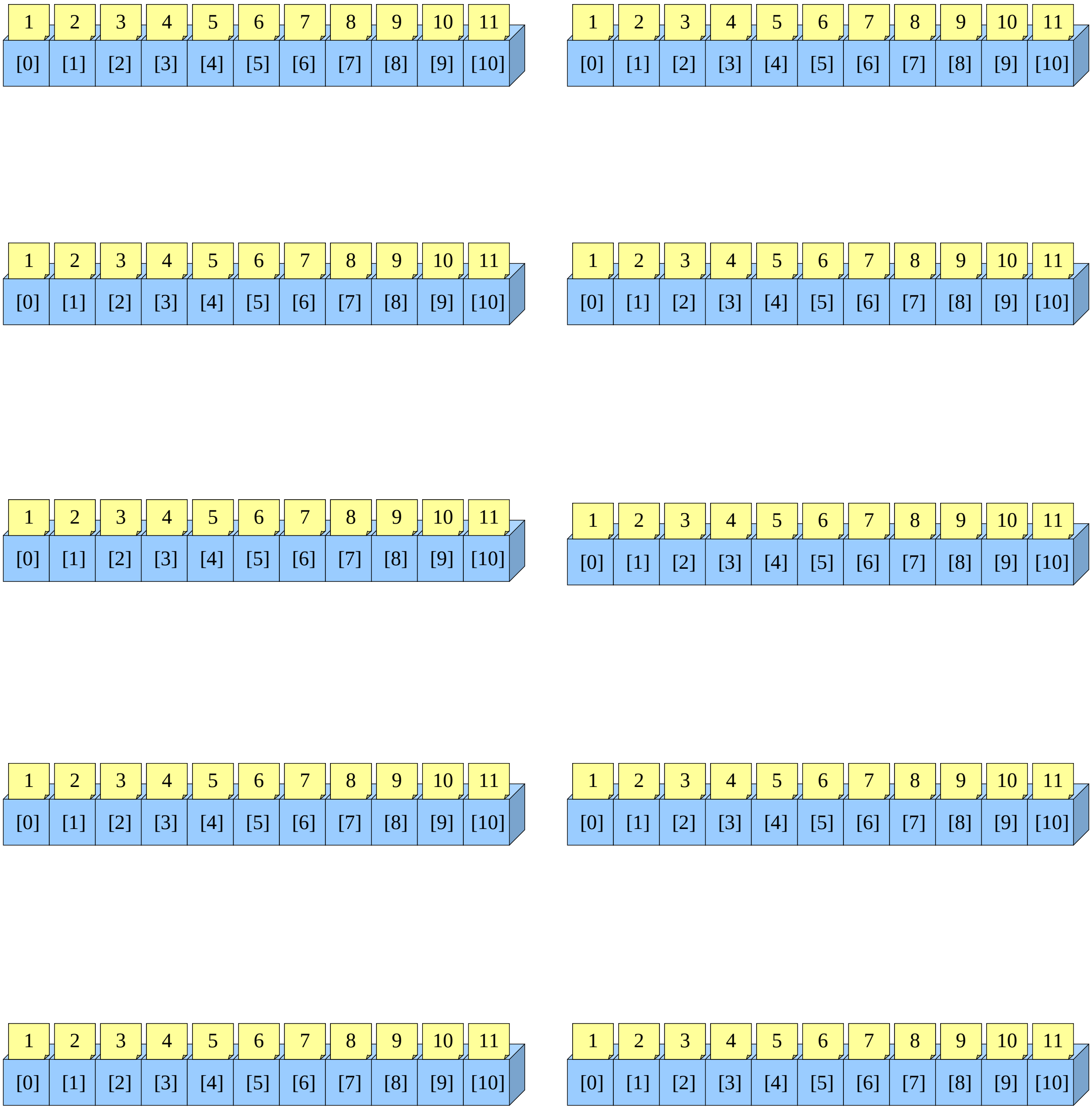




8.2.3.　バイナリサーチ

　ここでは新しく、「バイナリサーチ」という検索の手法を紹介します。このアルゴリズムは、配列の中身がソート済み（順番に並んでいる）時のみに有効です。今回扱う配列はソート済みになっているのでバイナリサーチを使うことができます。

　バイナリサーチの手順を書き込んで見ましょう。



①.バイナリサーチの実装

例題 8-3：バイナリサーチを使う(ItemTypeList.java)

```
1:      /**
2:      * オブジェクト指向哲学　入門編
3:      * 例題 8-3：バイナリサーチを使う
4:      * 追加と検索の性能を比較するプログラム
5:      *
6:      * 商品種類リストクラス
7:      * バイナリサーチによって商品種類を検索することができる
8:      * 現在の要素数を覚えておくことによって追加の効率化を図る
9:      */
10:     public class ItemTypeList {
11:
12:       private int ARRAY_SIZE = 10000;                              //用意する配列の大
きさ
13:
14:       private int size=0;                                          //現在配列に保存さ
れている要素数
15:
16:       private ItemType[] itemTypeArray = new ItemType[ARRAY_SIZE];  //商品種類を保存す
るための配列
17:
18:
19:       /**
20:       * 商品種類を追加する
21:       */
22:       public void add(ItemType addItemType){
23:         itemTypeArray[size] = addItemType;//空の箱から順に書き込む
24:         size++;//要素数を 1 つ増やす
25:       }
26:
27:       /**
28:       * 商品種類を検索する
29:       * (バイナリサーチ)
30:       */
31:       public boolean search(int searchID){
32:         //一度の検索で見つからなかったときに次に探す配列の範囲を限定して検索していく
33:         int lower = 0;//探す配列の範囲の最小値
34:         int upper = size-1;//探す配列の範囲の最大値
35:         int center;//次に調べるべき配列の番地
36:         while(true){
37:           center = (lower + upper)/2;//次に調べるのを範囲の真中に設定
38:           if(itemTypeArray[center].getId() == searchID){
39:             return true;//見つかった
40:           }else if(lower > upper){
41:             return false;//見つからず、かつ、もう調べる範囲がなくなった
42:           }else{
43:             //見つからないので、次に調べる範囲を狭める
```

```
44:             if(itemTypeArray[center].getId() < searchID){
45:               lower = center +1;//範囲を右半分にする
46:             }else{
47:               upper = center -1;//範囲を左半分にする
48:             }
49:           }
50:         }
51:       }
52:
53:     }
```

②.バイナリサーチの効率

バイナリサーチは、リニアサーチと比べてどれくらい効率が上がるのでしょうか。比較をしてみましょう。

要素数	比較最大回数	
	リニアサーチ	バイナリサーチ
10	10	2
100	100	7
1000	1000	10
10000	10000	14
100000	100000	17
1000000	1000000	20
10000000	10000000	24

このように扱う要素数が増えていくに従ってリニアサーチも比較回数が増加していきますが、バイナリサーチはそんなに増えません。なぜでしょうか。

バイナリサーチの比較回数	検索できる範囲
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

このように「２＾比較回数」、つまり２の巾乗がバイナリサーチの検索範囲になります。

では、以下の問題を考えてみましょう。

１００をバイナリサーチすると、最大何回の比較が必要でしょう？ 計算をして求めてください。

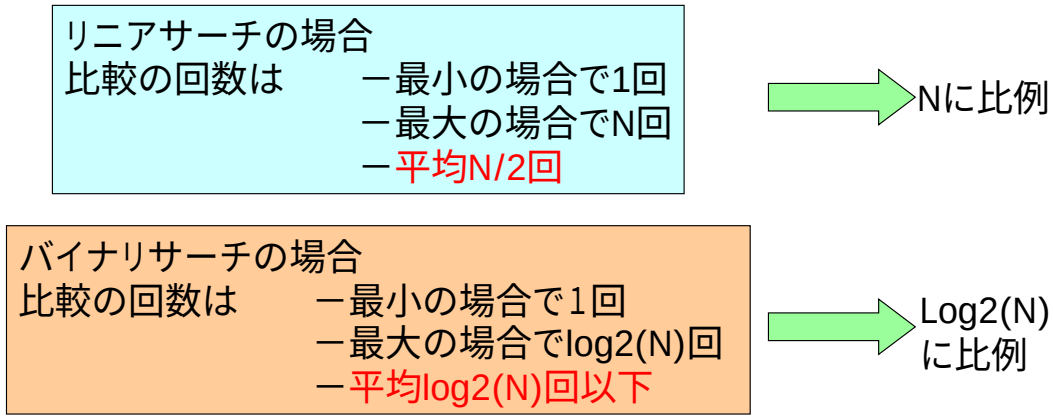
答え

計算式　　） 比較回数　）

8.3.　効率を比較する

8.3.1.　BigO 記法

仮にN個の要素の中から検索をする時、リニアサーチとバイナリサーチは平均で何回の検索で要素を見つけることができるでしょう。



平均検索回数は、リニアサーチは「Nに比例する」、バイナリサーチは「Log2(N)に比例する」ことになります。

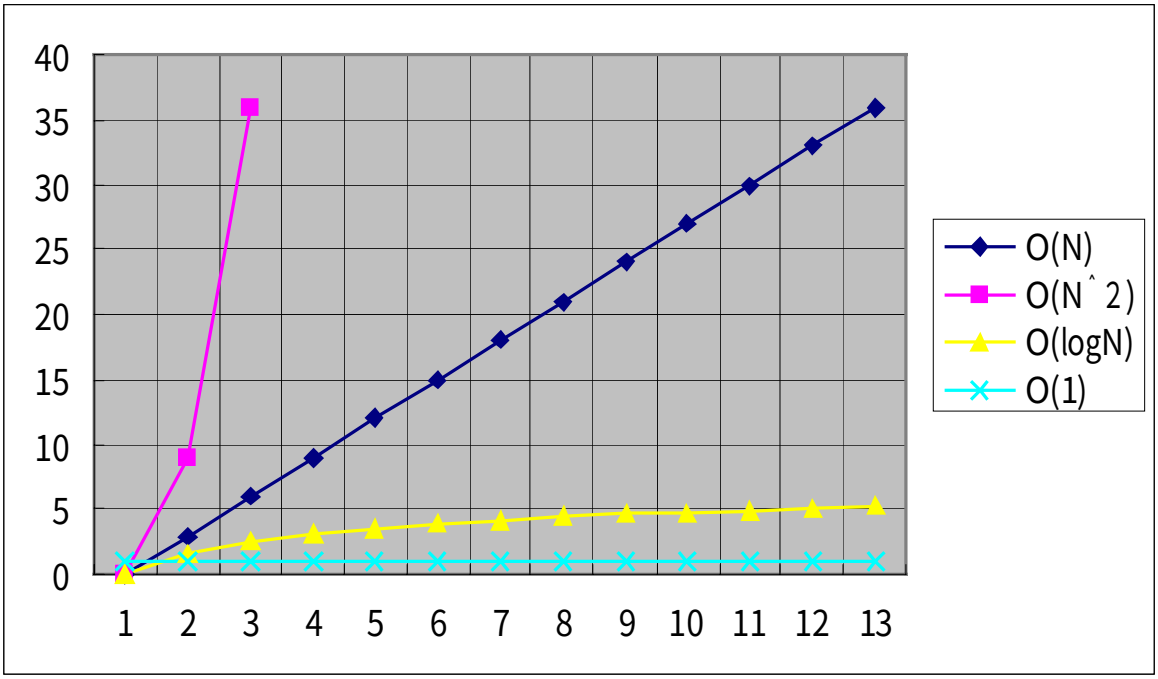
こうしたアルゴリズムの効率を表現するのに BigO 記法というものがあります。

	bigO記法
Nに比例	O(N)
Log2(N)に比例	O(LogN)
Nの２乗に比例	O(N^2)

リニアサーチとバイナリサーチの検索の効率を、BigO 記法で表現してみましょう。

リニアサーチ	
バイナリサーチ	

BigO 記法をグラフで表すと、以下のようになります。

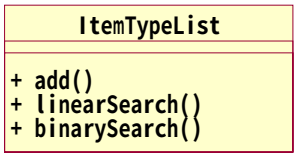


8.4.　アルゴリズムを比較できるプログラムの設計

リニアサーチとバイナリサーチの処理速度を比較できるようなプログラムを作りましょう。その為にはリニアサーチとバイナリサーチの使い分けが可能な設計にする必要があります。

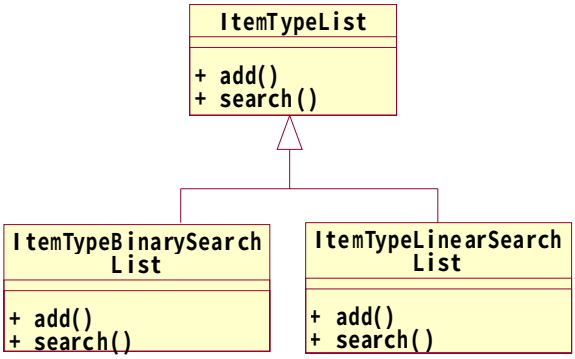
8.4.1.　案 I　メソッドを分ける

一つ目の案は、サーチするためのメソッドを二つ作ることです。以下のようにItemTypeList クラスにメソッドを追加します。



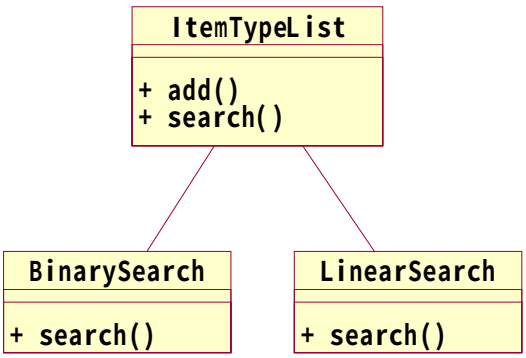
8.4.2.　案 II　抽象クラスを作る

オブジェクト指向では、ポリモーフィズムが使えます。以下のような設計ではいかがでしょうか？

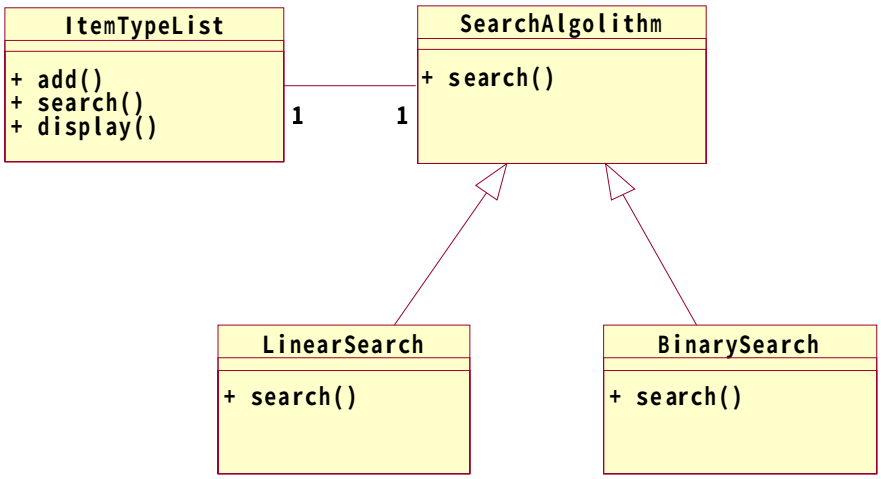


8.4.3.　案III　アルゴリズムだけをクラスにする

もう一つの案としては、アルゴリズムをクラスにしてしまうという方法があります。LinearSearch クラスと BinarySearch クラスがそれぞれの方法で実装された検索メソッドを持ち、ItemTypeList クラスが二つのクラスをもつ、という方法です。



さらに、バイナリサーチとリニアサーチは双方とも「検索する」という同じ意味を持つクラスです。これを「抽象的な意味」として、抽象クラスを作ってみるのも手です。



これら課題は if 文かポリモーフィズムかという前回の問題を考える上でも役立ちます。だけど難しいですね。分からなければ、やってみるのが一番良い方法です。やってみて、プログラムの意味が明らかになるものを選んでみましょう。

練習問題

＜記述問題＞

☆記述問題 8-1

プログラム問題 8-1 で作った性能比較プログラムの実行結果より、リニアサーチとバイナリサーチの効率について考察せよ。

☆記述問題 8-2

遅くてもリニアサーチを利用する場合を 3 つ以上考えよ。

＜プログラム問題＞

☆プログラム問題 8-1

リニアサーチとバイナリサーチの性能比較をするプログラムを書け。ただし、どのような実装方法でもかまわない。