問題 0(基本中の基本)
アルゴリズムとは何か。最も大事なことを言え

答え 0
「必ず止まる」
(ちなみに、止まらないプログラムを巧く利用するのがクラッカーの手段だったりする)

問題 1
線形探索とは何か

問題 2
二分探索とは何か

問題 3
二分探索を用いて、次の配列の中から 9 を探し出せ
1,2,3,4,5,6,7,8,9,10

問題 4-1
(教科書 52 ページを用いて、素数の JAVA プログラムを穴埋めして完成させる問題を作れ)
以下の JAVA プログラムは、1000 以下の素数を見つけるものである。穴埋めせよ。

```
(ア) Prime Number1
{
    (イ) main((ウ))
    {
        int counter = 0;//(エ)の回数
        for(int n=(オ); n<=1000;n++)
        {
            int i;
            for(i =(オ); i<=(カ); i++)
            {
                counter++;
                if((キ))//i で割れたのなら(ク)
                    (ケ);
            }
            if(i>(カ))//(コ)
                (サ);
        }
        (シ).println("(エ)の回数:" + counter);
    }
}
```

問題 4-2

(問題 4-1 カ)における工夫をさらに拡張しよう。(問題 4-1 カ)の工夫によって、i の範囲の大きさが 2 分の 1 になったといえる。では、3 分の 1 にするための方法を述べ、問題 4-1 のプログラムの書き換えよ。

問題 4-3

問題 4-2 のことをさらに拡張してみよう。

n を n/4 越 n/3 未満の値で割ったものは 3 越 4 未満となるから、割り切れることはない。ただ、4 で割れるのであれば当然 2 でも割れ、そのようなものは問題 4-2 の議論で直ちに非素数判定することになっているので、3 越 4 未満ではなく、3 越 5 未満を考えてもよさそうだ。

このことから、i の範囲から n/5 越 n/3 未満を除外しても問題なさそうである。

ただし、そのようなアルゴリズムでは、n を何かで割ったものが 3 になる可能性を否定していない。その為、n が「整数 n/3」で割れるのか、即ち、n/3 が整数であるかを別途調べる必要がある。さらに、n=3 の場合は、自分自身で割れる分には素数候補から除外できないの

だから、例外的に素数となることに注意する必要がある。
さて、このことは 5 越 7 未満、7 越 11 未満、...にも累積的に適応されるが、ここに挙げた 5、7、11 という数字(ここでは「区間素数」と呼ぶことにする)もやはり素数が根拠である。その為、このままでは「素数を効率的に求めるためだけに、区間素数という素数を求める必要がある」というパラドクスが生じてしまう。(素数を効率的に見つけるために区間素数を用いているが、ではその区間素数はどう求めたのか、ということである。)
このパラドクスを解決する方法を考え、プログラムを書き換えよ

問題 4-4
問題 4-3 のプログラムにおいても、結局は n/2 越の素数も用いて n が素数であるか否かを判定している。もっといえば n/3 を越える素数も、n/5 を越える素数も...不要であるといいたい。問題 4-3 までの理屈ではこの無駄を省けなかった理由と、どのように考えれば省くことが出来るのか述べよ。またプログラムを書き換えよ。

答え 4-4
問題 4-3 で n/2、n/3、...を越える値で割って確かめることをしないというのは、あくまで原則であり、その例外として、素数である場合はたとえ n/2、n/3、...n/m (m は自然数)を越えるものであっても、割って確かめていた。その理由は、原則の処理では m が自然数である可能性を取り除けなかったからである。
仕方ない無駄にも思えるが、少なくとも n/m<m の範囲では、n/(n/m)>n/m の確認と全く同じことを行っていて、即ち無駄であるから、素数で割って確かめる場合でも m は扱わなくてよい。ここで n/m<m を解くと、m>√n である。即ち、n の平方根を越えるすべての整数で割って確かめるのは、無駄であるということである。

プログラムは、問題 4-3 のもので、
i[m]!=0
を
i[m]!=0 && i[m]*i[m]<=n
に書き換え、さらに、
if(isPrime)の処理の一番最初に
for(; i[m]!=0; m++);
を追加する。(これをしないと素数リストが、途中で上書きされてしまう)

ちなみにこのアルゴリズムによる除算の回数は 1970 であり、問題 4-1 のアルゴリズムによる除算の回数は 39876 である。

問題 5
(アルゴリズムにおける)探索とは、(ア)データを探し出すことである。データを構成するどんな(イ)に着目するかによって、探し方はさまざまである。この注目する(イ)のことを(ウ)という。配列からの探索、線形リストからの探索、2 分探索木からの探索などが考えられる。線形リストとは、配列の最後に(エ)を与えて、配列を(オ)ようなものであると考えてよいだろう。(カ)ために使われる。2 分探索木を理解するためには、その前に 2 分木を理解する必要がある。2 分木とは(キ)データ構造((ク)ともいう)であり、2 分探索木は(ケ)2 分木である。

問題 6
ラピッドプロトタイピングとは何か。また他の開発手法を上げよ。

問題 7
ハッシュ値を用いた探索(ハッシュ法)とは何か

問題 8
2 分探索木の構造にすると、2 分探索が便利になる。その理由を説明せよ。

問題 9
番兵法とは何か

問題 10
参考: https://nobuo-create.net/string/
(78 ページを利用して、JAVA プログラムで線形探索するプログラムの問題を作れ。また、インスタンスとは何か説明し、配列がインスタンスであるか説明せよ。)
String はクラス型変数である。つまり String はクラスであり、
String s = ”str”;
は
String s = new String(”str”);

であり
String のフィールド(String クラス内の変数)には
private final char value[]
というものと
public char[] toCharArray()
があり、char 型の配列として文字が記録され、カプセル化されている。
今、
String univName = “TokyoDenkiUniversity”;
とした。k が 0 オリジンで最初に何番目に表れるのか求めるための strpos メソッドをかき、
それを線形探索で求めるプログラムを書け(java)。但し、String のフィールドでは、
private final int count;
で文字数を数えていて、
```
public int length() {
    return this.count;
}
```
となっている。

問題 11
番兵法を用いたとき、問題 10 の文字列はどうなるのか答えよ。

問題 12
Call By Value と Call By Reference の違いを「基本データ型」、「参照型」という言葉を用いて説明せよ。また、JAVA における例を示せ。

問題 13
C 言語における static と、Java における static をそれぞれ説明せよ。

答え 13
C 言語における static は、関数の中にある変数や関数を、同じファイルの中の別の関数内や、関数外でも使用できるようにし、同時に別のファイルからは使用できなくするためのものである。
Java における static は、インスタンス化できないクラスメソッドやクラス変数を作るものである。
例えば

```
class クラス
{
    static int static ﾒﾝﾊﾞ =0;
    int 非 static ﾒﾝﾊﾞ =0;
}
class テキトー
{
    クラス クラス型変数= new クラス();
}
```

としたとき、

クラス型変数.非 static ﾒﾝﾊﾞ

と

クラス.staticメンバ

が使用できる。

問題 14
2 分探索のプログラムを穴埋めせよ。

BinSearch.java

```
import java.util.Scanner;
class BinSearch
{
   static int binSearch(int[] a, int n, int key)
   //配列 a の先頭 n 個の要素から key と一致する要素を 2 分探索
   {
      int pl = (ア);//探索範囲先頭のインデックス
      int pr = (イ);//探索範囲末尾のインデックス
      do
      {
            (ウ)
      }
      while(エ)
      return -1;//探索失敗
}
public static void main(String[] args)
{
   (オ)
   int num=(カ);
   int[] x=new int[num];
   System.out.println(“昇順に入力してください。”);
   System.out.print(“x[0]:”);
```

```
    x[0]=(カ);
    for(int i=1; i<num; i++)
    {
        (キ)
    }
    System.out.print("探す値:");
    int ky=(カ);
    int idx=binSearch((ク));
    if (idx==-1)
        System.out.println("(ケ)");
    else
        System.out.println("(コ)");
}
```

問題×(問題として採用するかはわからない)
次の Java プログラムの「実行回数」と「時間計算量(オーダ)」を求めよ。

```
Static int segSearch(int[] a, int n, int key)
{
    int i=0;
    while(i < n)
    {
        if(a[i]==key)
            return i;
        i++;
    }
    return -1;
}
```

答え×

```
Static int segSearch(int[] a, int n, int key)
{
    int i=0;//実行回数 1、時間計算量 O(1)
    while(i < n)//実行回数 n/2、時間計算量 O(n)
    {
        if(a[i]==key)//実行回数 n/2、時間計算量 O(n)
/*
```

ここで、while の中を独立して考えて、if をの実行回数を 1 とすると、次の 2 点が問題になるからやめよ。
・return が説明つかなくなる
・※がいえなくなる
*/
                return i;//実行回数 1、時間計算量 O(1)
            i++;//実行回数 n/2、時間計算量 O(n)
        }
        return -1;//実行回数 1、時間計算量 O(1)
}

アルゴリズム全体のオーダは、出てきた中で、最大のオーダを示す(※)。従って、O(n)

問題×2(問題として採用するかはわからない)
時間計算量のオーダを、小さい順に並べよ。
1,n,2^n,n^2,log n,n log n

答え×2
1<log n<n<n log n<n^2<2^n

問題 15
参考: https://detail.chiebukuro.yahoo.co.jp/qa/question_detail/q13182372713
ハッシュ関数として、よく用いられるのが
「キーを素数で割った余剰」
が用いられる。
なぜ、素数なのか。なぜ合成数ではいけないのか説明せよ。

問題 16
例えば、
x[キーを 13 で割った余り]=キー
となるような配列を考えよう。このようなものをハッシュ表という。ハッシュ法では、キー毎に「データ」を与えることもある。
ハッシュ表に格納する利点は、「ずらす」操作が不要なので速いということである。
5 をハッシュ表に格納したいときは、x[5]に格納される。
その後、18 を格納したいときも、同じように x[5]に格納されることになり、すでに格納されている 5 が消されてしまう。このようなことを「衝突」(コリジョン)という。解決法を 2

通りいえ。

問題 17
チェイン法を説明せよ。

答え 17
同一のハッシュ値を持つデータを、「線形リスト」というリストにする。
線形リストとは、キーどうしを、ハッシュ表に対して垂直
に並べたものである。並べるために、今のキーと、次のキーへのアドレスを書いておく。また、ハッシュ表には、線形リストの先頭のアドレスを格納しておく。

問題 18
ジェネリックとは何か説明せよ。
例となるプログラムを書いて説明せよ。

答え 18
クラス内、またはメソッド内で、型にこだわらずにフィールド(変数)を利用したいときにつかわれる手法である。(但し、クラス型に限る)

```
/*
例えば、
xyz という変数を、「型にこだわらずに」利用するクラス「GenericClass」を作ろう。
*/
class GenericClass<T>
{
    private T xyz;
    GenericClass(T t)
    {
        this.xyz=t;
    }
    T getXyz()
    {
        return xyz;
    }
}
/*
```

これを、利用するクラスは、
*/
class GenericClassTester
{
	public static void main(String[] args)
	{
		GenericClass<String> str = new GenericClass<String>("str");
		GenericClass<Integer> i = new GenericClass<Integer>(1);
		System.out.println(str.getXyz());
		System.out.println(i.getXyz());
	}
}
/*
である。
*/

問題 19〜25
問題 19･･･(オ)まで
問題 20･･･(カ)
問題 21･･･(キ)、(ク)
問題 22･･･(ケ)
問題 23･･･(コ)
問題 24･･･(サ)
問題 25･･･(シ)
※問題 22〜25 の答えはとても長いが、実際に紙に書いて覚えよ。その際、コメントは理解出来次第省略せよ。すべてのコメントが省略できるようになるまで繰り返せ

次のプログラムは、チェイン法によるハッシュを実現するプログラム ChainHash.java である。穴埋めせよ。
public class ChainHash(ア)
{
	class Node<K,V>//ノード(線形リストの各要素)を定義
	{
		private K key;//キー値
		private V data;//データ
		private (イ) next;//後続ノードへの参照

```
    Node(K key, V data, Node<K,V> next)//(ウ)
    {
        this.key = key;
        this.data = data;
        this.next = next;
    }

    K getKey()//キーを返す
    {
        return key;
    }

    V getValue()//データを返す
    {
        return data;
    }
    public int hashCode()//キーを強制的に数値(以下「キー数値」)化したものを返す
    {
        (エ)
    }
}
private int size;//ハッシュ表の大きさ
(オ)//ハッシュ表

public ChainHash(int capacity)//(ウ)
{
    (カ 1)
    {
        table = new Node[capacity];//ハッシュ表の実体
        this.size = capacity;
    }
    (カ 2)
    {
        this.size = 0;
```

```
        }
    }

    public int hashValue((キ) key)//キーのハッシュ値を求める
    {
        (ク)
    }

    public V search(K key)//指定されたキーに対応するデータを探し出すメソッド
    {
        (ケ|データがあった場合はそのデータを、ない場合は null を返却せよ。
        ヒント:「a.hashCode()==b.hashCode()」は「a.equals(b)」で代用できる)
    }

    public int add(K key, V data)//ノードを追加するメソッド
    {
        (コ|登録済みのキー値であることが分かった場合は追加せず 1 を返却せよ。追加に
        成功した場合は 0 を返却せよ。(ケ)と同じヒントが有効)
    }

    public int remove(K key)//ノードを削除するメソッド
    {
        (サ|削除に成功したら 0 を、キーが存在しなかったため削除できなかった場合は 1
        を返却せよ。(ケ)と同じヒントが有効)
    }

    public int void dump()//ハッシュ表と線形リストをまとめて表示
    {
        (シ)
    }

}
```

答え 19〜25
(ア):<K,V>
解説:クラス内で使用されているジェネリックをすべて記述する。(但し、内部クラス内で完

結しているものは除く）
(イ):Node<K,V>
(ウ):コンストラクタ

(エ):return key.hashCode();
解説:こっちの hashCode()は、
java.lang.Object クラスであらかじめ定義されている hashCode メソッドである。すべてのクラスは、java.lang.Object の子孫クラスなので、
即ち、すべてのクラス型で(当然、K でも)利用できるメンバメソッドであるといえる。
(ちなみに、K=Integer のとき、hashCode()では、コンストラクタに代入した int 型の値がそのまま返ってくることが実験により確認されている。)

(オ):private Node<K,V>[] table;
解説:「ハッシュ表の実体」のところをみればわかるはず。

(カ 1):try//とりあえずやってみる
(カ 2):
catch(OutOfMemoryError e)//try した結果、OutOfMMemoryError が発生したときに実行する

(キ):Object
(K でもいいと思う（？）)

(ク):return key.hashCode() % size;
解説:キーからキー数値を求め、それをハッシュ表の大きさで割った余剰によりハッシュ値を求めている。また、ハッシュ表の大きさとしては、素数が好まれる→問題 15

(ケ):

```
Node<K,V> p= table[hashValue(key)];
/*
ハッシュ表を左右、線形リストを上下で表現すると、
与えられたキーに対応するデータが格納されている場合、左右方向は確実に
table[hashValue(key)]となる。
言い方を変えると、対応するデータがあるのなら、それは p つまり table[hashValue(key)]
の真下にあるデータなはずなのである。
*/
while(p != null)
/*p を一つずつ真下にしていく。これを null つまり一番下になるまで繰り返す*/
{
    if(p.getKey().equals(key))
    /*
    p.getKey.hashCode()==key.hashCode()
    と同じ。
    p のキー数値と与えられたキーのキー数値が一致するか調べている。
    ただし、(エ)があるので、
    p.equals(key)としてもよかったはずである。(？)
    */
    {
        return p.getValue();
        //探索成功。データを返す。
    }
    p = p.next;//真下を探す。
}
return null;//探索失敗
```

(コ):

```
Node<K,V> p= table[hashValue(key)];
while(p != null)
{
    if(p.getKey().equals(key))
    {
        return 1;
    }
    p = p.next;//真下を探す。
```

```
}
Node<K,V> temp = new Node<K,V>(key, data, table[hashValue(key)]);
/*
temp という、線形リストの要素(ノード)を作る。
key と data は、メソッドに与えられた通り。
next は、table[hashValue(key)]と書いてあるが、
実際には table[hashValue(key)]に格納された値、即ち「現段階での(temp を追加する前の)」
線形リストの先頭ノードへの参照が格納される。
*/
table[hashValue(key)]=temp;
/*temp は、線形リストの一番上(即ち先頭ノード)に追加する。*/
return 0;
```

(サ):

```
int hash = hashValue(key);//削除するデータのハッシュ値
Node<K,V> p = table[hash];//着目するノード
Node<K,V> pp=null;//前回の着目ノード
while(p != null)//一番下まで繰り返す
{
    if(p.getKey().equals(key))//見つけたら
    {
        if(pp==null)//一番上に見つかったのなら
        /*
        pp が null であるということは、
        『pp が一度も「前回の p」になったことがない』
        ということを必ず意味する。
        これは、pp に null を代入する処理が、一番最初にしか存在しないことからも確認
        できる。
        */
            table[hash]=p.next;//一番上を削除する
            /*
            目的:一番最初を削除する。
            手段:「2 番目に上」だったものを 1 番上に移動させる
            */
        else//一番最初以外で見つけたのなら
            pp.next=p.next;// 「前の次(＝今)のノード」=「今の次(＝次)のノード」
            /*
            目的:対象のキーを削除する
            手段: 「次のノード」だったものを「今のノード」に移動させる
            */
        return 0;//remove 完了
    }//まだ見つけていないのなら
    pp=p;//「前の着目ノード」を「着目ノード」にして
    p=p.next;//「着目ノード」を次のノードにする
}//一番下まで探したけど見つからなかったなら
return 1;//そのキー値は存在しない。
```

(シ):

```
for(int i=0; i<size; i++)//ハッシュ表の大きさだけ繰り返す
{
    Node<K,V>p==table[i];
    System.out.printf("%02d  ",i);//整形のため書式指定。printf
    while(p != null)
    {
        System.out.printf("→ %s (%s)",p.getKey(),p/getValue());
        p = p.next;
    }
    System.out.println();
}
```

問題 28
(ジェネリックを継承で代用させる問題)

問題 29
スタックの順番、キューの順番をいえ。また、入れることや出すことを何というかいえ。また、最初と最後の名称もいえ。

問題 30
デキューの際には、(ア)なければいけないのが問題となる。この手間を省くため、(イ)をリングバッファという。インデックス(配列の番目数)は front と rear という変数で表現される。論理的な先頭は、配列の(ウ)番目の要素、論理的な末尾は配列の(エ)番目の要素、としてあらわされる。このことより、次にエンキューされるデータが格納されるのは、配列の(オ)番目である。
エンキューされた際には、変数(カ)が(キ)クリメントされる。
デキューされた際には、変数(ク)が(ケ)クリメントされる。

問題 31
次のプログラムで、何が返されるのか、あるいはエラーなのか。Call by Reference を用いて説明せよ。

```
int a[]={1};
int b[]=a;
b[0]=2;
```

System.out.println(a[0]);

問題 32〜__
次のプログラムは、スタックのアルゴリズムを実現するものである。穴埋めせよ。

IntStack.java

```
public class IntStack
{
   private int max;//スタックの容量
   private int prt;
   private int[] stk;
   (カ)
   (サ)
   public IntStack(int capacity)
   {
      ptr=0;
      max=(ア);
      try
      (イ|スタックの実体を作れ。失敗した場合に強制終了せず、かつ不正参照を防げ)
   }
   public int push(int x) (オ)
   {
      if(ptr>=max)//(ウ)なら
         (エ); //例外 OverflowIntStackException を投げる
      return (キ|一行でかけ);//(ウ)でない場合(つまり通常通りの時)、やる。
      /*
      (ク|オの解説をかけ。どうしてその書き方ができるのか。)
      */
   public int pop() (ケ|push メソッドを参考にせよ)
   {
      (コ|push メソッドを参考にせよ)
   }
   public int peek() (ケ)/*最新のデータを覗き見する*/{(シ)}
   public int indexOf(int x)/*x を探してその場所を返す(なければ-1)*/{(ス)}
   public void clear()/*スタックを空にする*/{(セ)}
   public int capacity()/*スタックの容量*/{return max;}
   public int size()/*スタックにあるデータ数*/{(ソ)}
   public boolean isEmpty()/*スタックは空であるか*/{return ptr<=0;}
   public boolean isFull()/*スタックは満杯であるか*/{return (タ);}
   public void dump(){(チ)}
}
```

IntStackTester.java

```
import java.util.Scanner;
class IntStackTester
{
   public static void main(String[] args)
   {
      (ツ)
      (テ)
      while(true)
      {
         System.out.println("現在のデータ数:"+ s.size() + " /" s.capacity());
         System.out.print("(1)プッシュ (2)ポップ (3)ピーク (4)ダンプ (0)終了 : ");
         int menu=stdIn.nextInt();
         (ト)
      }
   }
}
```

(ア):capacity
解説:関数頭部に書かれている仮引数は何故書かれているのか、という疑問を持つべき。
(イ):

```
try//まず～～みる
{
   stk = new int[max];//～～=実際に作って
}
catch(OutOfMemoryError e)//それでダメだったら
{
   max=0;
   /*
   配列のサイズは 0 であると明示することで
   他のメソッドによる不正参照を防いでいる。
   */
}
```

(ウ):スタックが満杯
(エ):

throw new OverflowIntStackException()
(オ):
throws OverflowIntStackException
解説:(エ)を投げうるメソッドにはこれを書くこと。
英文的に解釈すると、method which throws ~~「〜〜を投げるメソッド」となり、その which が省略されている。
(カ):
public class OverflowIntStackException extends RuntimeException
{
　public OverflowIntStackException(){}
}
(キ):stk[ptr++]=x
(ク):
このように書くと、stk[ptr]に x を代入してから stk[ptr]が返却され、また x の代入後に ptr がインクリメントされ、整合性に問題が生じないからである。
まず、
a=1;
return a=2;
とすると、2 が返却されるまた、a は 2 となる。これは、a=2 を return しようとするためにまず a=2 を評価するので、(return の前に)代入が実行されるからである。
　また、stk[ptr]=x としてから ptr をしたいため、
stk[ptr++]=x
としている。この書き方であれば、ptr が評価されてからインクリメントされる。(☆)
反対に、
stk[++ptr]=x
と書いてしまうと、
ptr がインクリメントされてから評価されてしまうので、一つ後ろに代入されてしまう。
　以上のことより、
return stk[ptr++]=x;
は、stk[ptr]を評価してから ptr をインクリメントし、そこに x を代入してから stk[ptr]を返却している。
(ケ):throws EmptyIntStackException
(コ):
if(ptr<=0)//スタックが空なら
　throw new EmptyIntStackException();
return stk[--ptr];

解説:
(ク)の☆に示したことから分かろうが、ptr は最新のデータのある場所の一つ次を常にさしている(このことは(シ)の return からもわかる)。その為、先にデクリメントしてから返却しないと、pop された要素を返却することができない。
(サ):
```
public class EmptyIntStackException extends RuntimeException
{
   public EmptyIntStackException(){}
}
```
(シ):
```
if(ptr<=0)
   throw new EmptyIntStackException();
return stk[ptr-1];
```
(ス):
```
for(int i=ptr-1;i>=0;i--)
   if(stk[i]==x)
      return i;
return -1;
```
(セ):ptr=0;
解説:
頓智(とんち)にも見えるが、ptr は常に「最新のデータの場所の次」を返すので、これを 0 としてしまえば、データがないということになる。実際に stk[0]や stk[1]や...がどうであろうと、それらは ptr がゼロとなった瞬間にゴミとなる。

(ソ):return ptr;
解説:
ptr は常に「最新のデータの場所の次」を返すが、これは即ちデータの個数である。
例えば stk[0]、stk[1]、stk[2]、... 、stk[n]までデータが埋まっているとしよう。
最新のデータは stk[n]であり、その次は stk[n+1]である。よって ptr は n+1 だ。
一方データの個数も、確かに n+1 となっている。

(タ):ptr>=max
(チ):
```
if(ptr<=0)
   System.out.println(“スタックは空です。”);
else
```

```
{
   for(int i=0; i<ptr; i++)
      System.out.print(stk[i] + " ");
   System.out.println();
}
```

(ツ):Scanner stdIn = new Scanner(System.in);
(テ):IntStack s = new IntStack(64);

(ト):

```
if (menu==0) break;
//Switch 文内では break が意味違いになってしまうのでここにかく
int x;
Switch(menu)
{
   case 1://プッシュ
      System.out.print("データ:");
      x=stdIn.nextInt();
      try
      {
         s.push(x);
      }
      catch (IntStack.OverflowIntStackException e)
      {
         System.out.println("スタックが満杯です。");
      }
   break;
   case 2://ポップ
      try
      {
         x=s.pop();
         System.out.println("ポップしたデータは"+x+ "です。")
      }
      catch (IntStack.EmptyIntStackException e)
      {
         System.out.println("スタックが空です。");
```

```
      }
    break;
    case 3://ピーク
      try
      {
        x=s.peek();
        System.out.println("ピークしたデータは"+x+ "です。")
      }
      catch (IntStack.EmptyIntStackException e)
      {
        System.out.println("スタックが空です。");
      }
    break;
    case 4://ダンプ
      s.dump();
    break;
}
```

問題□1
リングバッファとは何か。

答え□1
末尾が先頭につながった配列

問題□2
リングバッファを用いてキューを作った。穴埋めせよ。
IntQueue.java

```
public class IntQueue
{
   private int max;//キューの容量
   private int front;//(ア)
   private int rear;//(イ)
   private int num;//現在のデータ数
   private int[] que;//キューの本体
   public class EmptyIntQueueException extends RuntimeException
   {
      public EmptyIntQueueException(){}
   }
   public class OverflowIntQueueException extends RuntimeException
   {
      public OverflowIntQueueException(){}
   }

   public IntQueue(int capacity)
   {
      num=front=rear=0;
      max=capacity;
      (ウ|キューの実体を作れ。失敗した場合に強制終了せず、かつ不正参照を防げ)
   }
   public int enque(int x)(エ)/*エンキュー*/{(オ)}
   public int deque()(カ)/*デキュー*/{(キ)}
   public int peek()(カ)/*ピーク*/{(ク)}
   public int indexOf(int x)/*x の場所(なければ-1)を返す*/{(ケ)}
   public void clear()/*キューを空にする*/{num=front=rear=0;}
   public int capacity()/*キューの容量*/{return max;}
   public int size()/*キューにあるデータ数*/{return num;}
   public boolean isEmpty()/*キューは空であるか*/{return num<=0;}
   public boolean isFull()/*キューは満杯であるか*/{return num>=max;}
   public void dump(){(コ)}
}
```

IntQueueTester.java

(IntStackTester.java を簡単に書き加えるだけなので省略する。)
答え□2

(ア):先頭要素のカーソル
(イ):末尾要素のカーソル
(ウ):

```
try
{
   que = new int[max];
}
catch(OutOfMemoryError e)
{
   max=0;
}
```

(エ):throws OverflowIntQueueException
(オ):

```
if(num>=max)//データ数が容量以上なら
   throw new OverflowIntQueueException();//例外を投げる
que[rear++]=x;//末尾に x を追加してから末尾のポインタを一つ後ろへ。
num++;//当然、データ数も増える
if(rear==max)//末尾のポインタが「最後の次」に来たら
/*最後のポインタは max-1 であり、max ではないことに注意。これは前者が 0 オリジンであり、数える数(＝自然数)が 1 オリジンであることに起因する。このことにより、max は「最後の次」である。*/
   rear=0;//「最初」と解釈しなおす。(これがリングバッファである。)
return x;
```

(カ):throws EmptyIntQueueException
(キ):

```
if(num<=0)
   throw new EmptyIntQueueException();
int x=que[front++];//先頭の要素を x に格納してから先頭のポインタを一つ後ろへ。
num--;//当然、データ数も減る
if(front==max)//末尾のポインタが「最後の次」に来たら
   rear=0;//「最初」と解釈しなおす。
return x;
```

(ク):

```
if(num<=0)
   throw new EmptyIntQueueException();
return que[front];
(ケ):
for(int i=0;i<num;i++)
{
   int idx=(i+front)%max;
   /*このようにすると、front のポインタがどこであろうと、
   リングバッファ全体をくまなく探せる*/
   if(que[idx]==x)//あったら
      return idx;//それを返すし
}
return -1;//なければ-1 を返す。
(コ):
if(num<=0)
   System.out.println("キューは空です。");
else
{
   for(int i=0;i<num;i++)
      System.out.print(que[(i+front)%max]+"");
      //こうすると、front からリングバッファを一周できる
   System.out.println();
}
```

問題□3

バブルソートを行うプログラムを完成させよ。

```
import java.util.Scanner;
class BubbleSort
{
   static void swap(int[] a, int idx1, int idx2){(ア)}
   static void bubbleSort(int[] a, int n){(イ|もっとも重要)}//第二引数は配列の要素数
   public static void main(String[] args)
   {
      Scanner stdIn=new Scanner(System.in);
      System.out.print("要素数:");
      int nx=stdIn.nextInt();//nx は要素数。
```

```
    (ウ)//ソートするための配列をここで作る。(各値はユーザに聞く)
    bubbleSort(x,nx);
    System.out.println("昇順にソートしました。");
    for(int i=0; i<nx; i++)
      System.out.println((エ));
}
```

答え□3

(ア):

```
int t=a[idx1];
a[idx1]=a[idx2];
a[idx2]=t;
```

(イ):

```
for(int i=0; i<n-1; i++)//配列 a[0~n-1]内任意の要素を a[i]とする。
  for(int j=n-1;j>i;j--)//i 超 n-1 以下の範囲で
    if(a[j-1]>a[j])//自分と一つ前の要素を比較して、昇順になっていないものがあったら
      swap(a,j-1,j);//入れ替える
```

補足:(イ)で繰り返している操作のように、アルゴリズムの主役となる繰り返し操作のことを「パス」という。

(ウ):

```
int[] x=new int[nx];
for(int i=0; i<nx; i++)
{
  System.out.print("x[" + i + "]:");
  x[i]=stdIn.nextInt();
}
```

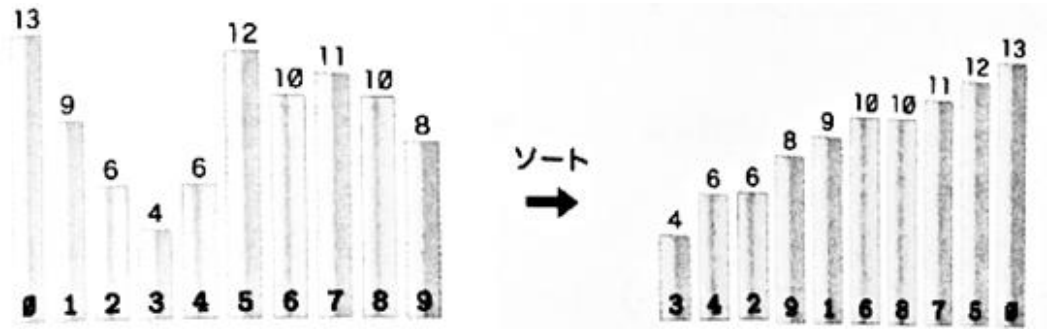
解説:

すぐ下にある「bubbleSort(x,nx)」より、配列名は x でなければいけないということがわかる。

(エ):"x[" + i + "]=" + x[i]

問題□4

次の図では、昇順にソートされている。しかし、「安定にソートされていない」という。どういうことか説明せよ。また、安定にソートした結果を示せ。



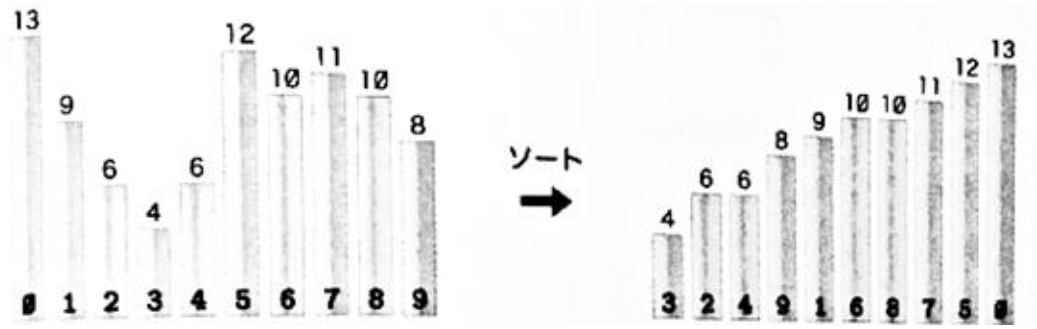
答え□4

要素番号 4 と 2 に注目せよ。

不必要に順番が入れ替わってしまっている。

このようなことが起こりうるのが不安定。起こり得ないのが安定。

したがって、安定であれば次のようになる。



問題□5

問題□3 をみよ。単純交換ソート(＝バブルソート)では、あるパスにおける要素の交換回数が 0 であったとすると、それは「全ての要素がソート済みであった」ということを意味する。このことを利用して、問題□3 のプログラムを高速化せよ。また、この高速化の手法のことを何というか。

答え□5

bubbleSort メソッドの中身に、次の太字部分を加える。

```
for(int i=0; i<n-1; i++)//配列 a[0~n-1]内任意の要素を a[i]とする。
  int exchg=0;
  for(int j=n-1;j>i;j--)//i 超 n-1 以下の範囲で
    if(a[j-1]>a[j])//自分と一つ前の要素を比較して、昇順になっていないものがあったら
    {
      swap(a,j-1,j);//入れ替える
      exchg++;
```

```
    }
  if(exchg==0) break;
```
この手法を「打ち切り」という。

問題□6

問題□3 をみよ。次のプログラムは単純交換ソート(＝バブルソート)を問題□5 とは別の手法で高速化したものである。アイデアを説明せよ。

問題□3 のプログラムにおいて bubbleSort メソッドを次のように書き換えることで高速化した。

```
int k=0;//a[k]より前はソート済み
while(k<n-1)
{
  int last=n-1;
  for(int j=n-1; j>k; j--)
    if(a[j-1]>a[j])
    {
      swap(a, j-1, j);
      last=j;
    }
  k=last;
}
```

答え□6

まず、与えられたプログラムにコメントを書いて、動作を理解しよう。

```
int k=0;//a[k]より前はソート済み
while(k<n-1)
/*
k>n-1 はあり得ない。
k==n-1 のとき、a[n-1]より前がソート済み、つまり全体でソートが終わっている。
k<n-1 のとき、全体ではソートが終わっていない･･･①
*/
{
  int last=n-1;//⑧変数「last」を利用可能にする。
  for(int j=n-1; j>k; j--)//②ソート済みでない要素を後ろから見ていく
    if(a[j-1]>a[j])//③並び替えが必要なところを見つけたら
    {
```

```
            swap(a, j-1, j);//④並び変えて、
            last=j;//⑤そこを「last」としてメモしておく。
          }
          //⑥最終的に、「last」は「ソート済みの部分の一番後ろ」を意味するようになる。
        k=last;//⑦a[k]より前はソート済み
      }
```

よって、これは「ソート済みのところをみつけてそこの確認を省く」という手法である。

問題□7

ヒープとは、「親の値が必ず子の値以上になる 2 分木構造」のことをいう。(「以下」でもよい)

1,7,14,12,0,8,4 をヒープにした図をかけ

答え□7

まず、適当な順番で 2 分木構造にする

1
┣7
┃ ┣12
┃ ┗0
┗14
  ┣8
  ┗4

まず、一番若い世代の親子 2 世代をヒープ化する(以下、ヒープ化中のものは太字、ヒープ化済みのものは赤字で示す)。2 世代間のヒープ化は、その中の最大値を親にして、それ以外を子にするだけでよい。

1
┣7
┃ ┣12
┃ ┗0
┗**14**
  ┣**8**
  ┗**4**

1
┣7
┃ ┣12
┃ ┗0
┗14
  ┣8
  ┗4
1
┣12
┃ ┣7
┃ ┗0
┗14
  ┣8
  ┗4

次に、3 世代をヒープ化する。(ヒープ化済みは青で示す)

1
┣12
┃ ┣7
┃ ┗0
┗14
  ┣8
  ┗4

ヒープ化け中のものの中の最大値(ここでは 14)を一番上の親にする。

14
┣12
┃ ┣7
┃ ┗0
┗1
  ┣8
  ┗4

14
┣12
┃ ┣7
┃ ┗0
┗8
  ┣1
  ┗4

14
┣12
┃ ┣7
┃ ┗0
┗8
  ┣1
  ┗4

ヒープ
完全２分木
親の値は子の値以上
根は最大値
兄弟の大小関係は任意
半順序木←意味は？
・図 6-34 確認せよ。
任意の要素 a[i]に対して、
親は a[(i-1)/2]余剰は切り捨て
左の子は a[i*2+1]
右の子は a[i*2+2]

ボドムアップ的積み上げによる(最初の)ヒープ化(図 6-38)

問題□□1
再帰とは。メリットも含めて説明せよ。

答え□□1
ある事象が自分自身を含んでいたり、自分自身を用いて定義されていたりするもの。

効果的にアルゴリズムを記述可。

問題□□2
階乗 n!の定義を(再帰的に)いえ

答え□□2
1. 0!=1
2. n>0 ならば
　n!=n×(n-1)!
注意:ナンバリング(1.および 2.を「・」で示したら不可なので気を付けよ)

問題□□3
階乗を再帰的に求める java プログラムを穴埋めせよ。

```
import java.util.Scanner;
class Factiorial
{
   static int factorial(int n){(ア)}
   public static void main(String[] args){(イ)}
}
```

答え□□3
(ア):

```
if(n>0)
   return n * factorial(n-1);
else
   return 1:
```

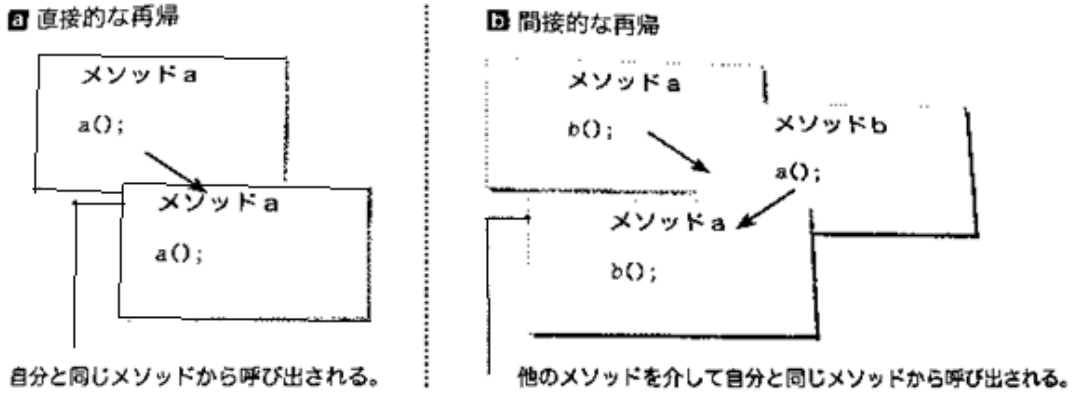
(イ):

```
Scanner stdIn = new Scanner(System.in);
int x = stdIn.nextInt();
System.out.println(x+“!=” + factorial(x));
```

問題□□4
直接的再帰と間接的再帰をそれぞれ説明せよ。図もかけ

答え□□4

直接的再帰は自分と同じメソッドを直接呼び出すこと
間接的再帰は他のメソッドを介して同じメソッドが呼び出されること



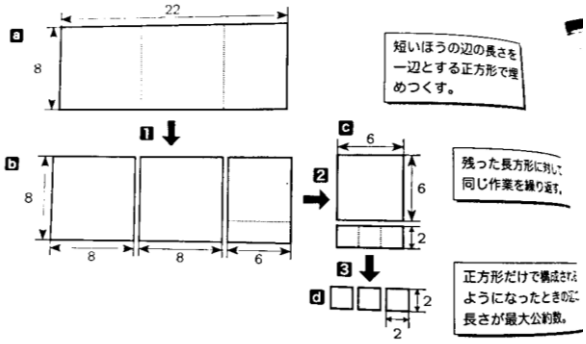
問題□□5
間接的再帰の独特な問題を一つ挙げよ

答え□□5
例えば次の図で、直接的再帰の場合は a()だけ見れば再帰の存在がわかるが、環節的再帰の場合は、a()を見ただけでは再帰の存在に気が付けない。(関わるメソッドがさらに増え、複雑化すると、この問題は急激に深刻化する)

問題□□6
2 自然数の最大公約数を求めることは、「長方形を正方形で埋め尽くす」問題に置き換えることができる。どういうことか具体的に説明せよ。

答え□□6
a と b の最大公約数 gcd(a,b)を求めることは、
縦 a 横 b の長方形を、同じ大きさの正方形で埋め尽くすとき、その正方形の一辺として最大なもの gcd(a,b)を求めることに対応する。

問題□□7
問題□□6 のたとえを用いて、ユーグリットの互除法を説明せよ。



問題□□8
問題□□7 を眺め、「最大公約数」を再帰的に求める手順を考えよ

答え□□8
x>y とする。
「短いほうの辺の長さを...」の手順が
gcd(x,y)で説明されるのであれば、
「残った...同じ作業を繰り返す」の手順は
gcd(y,x%y)
である。
これらは一致するのだから
gcd(x,y)= gcd(y,x%y)
が成り立つ。
また、この**アルゴリズムの止まるタイミング**を考えると(重要)、
gcd(y,x%y)について、x%y が 0 のとき、つまり x が y の倍数になった時である。
このときの y が答えとなる。

問題□□9
ユークリッドの互除法を、問題□□8 の方法で再帰的にやる java プログラムを穴埋めせよ。

```
import java.util.Scanner;
class EuclidGCD
{
   static int gcd(int x, int y){(ア)}
   public static void main(String[] args){(イ)}
}
```

答え□□9
(ア):

```
if(y==0)
   return x;
else
   return gcd(y,x%y);
```

(イ):

```
Scanner stdIn= new Scaner(System.in);
System.out.println(“input 2 Natural Numbers x and y”);
int x=stdIn.nextInt();
int y=stdIn.nextInt();
System.out.println(“gcd(” + x + “,” + y + “)=” + gcd(x,y));
```

問題□□□1

uml とは何か

答え□□□1

問題△1

問題○

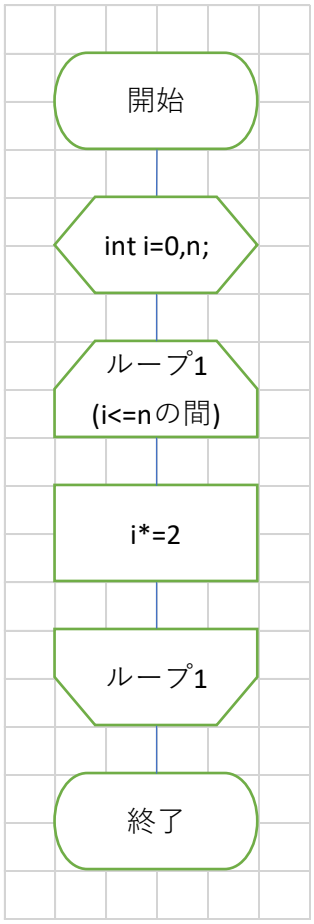
時間計算量とは何か

問題○2

時間計算量の求め方

問題○3

次のフローチャートの時間計算量を求めよ。



問題〇4
2 分探索の時間計算量を求めよ。

問題〇5
入れ子の時間計算量を求めよ。

問題〇6
分岐の時間計算量を求めよ

答え〇
入力サイズの増加に対して、実行時間がどれくらいの割合で増加するかを表す指標。

答え〇2
ステップ数を入力サイズ n の式で表し、最大次数以外の項、および係数を除く。

答え○3
まず、ループ内の処理が繰り返される回数 c_{MAX} を求めよう。(但し c はループ回数を示す変数である)また i の最大値を i_{MAX} とする。
$i_{MAX} = 2^{c_{MAX}} \fallingdotseq n$
という関係が成り立つ。
変形すると
$c_{MAX} \fallingdotseq \log_2 n = \dfrac{\log_a n}{\log_a 2}$ (aは正で、かつ 1 でないならば何でもよい)
となる。従って、時間計算量は仮に$\log_a n$と表される。このことは、時間計算量で log を考えるとき、底はどうでもよいということを意味する。その為、次のように表す。
$\log n$

答え○4
2 分探索では、各探索ごとに、探すべきデータ配列の長さが 2 分の 1 倍になる。
元の配列の長さがnとしよう。この中から探索することは、
nを何回か(ここでは c 回としよう)1/2 倍して、1 にする
ことであるので、これを式に表すと、
$n \cdot \left(\dfrac{1}{2}\right)^c = 1$
変形すると
$c = \log_2 n$
である。
よって時間計算量は$\log n$
(1 回当たりの時間計算量をaとすると、$a \cdot \log n$であるようにみえるが、時間計算量を考えるときは係数は無視されるので、aは時間計算量に無関係である。)

答え○5
内側と外側の積をとる。

答え○6
分岐先の各時間計算量の最大値が分岐の時間計算量となる。

答え 1
先頭から順番に、目的の値が見つかるまで探していく方法。

答え 2

ソート済みの配列において、データ列の中央と大小比較をし続けることで、探していく方法

答え 3
6≤9 より
6,7,8,9,10 の中にあるはず。
8≤9 より
8,9,10 の中にあるはず
同様に
9,10
9

答え 4-1
(ア):class
(イ):public static void main
(ウ):String[] args
(エ):除算
(オ):2
(カ):n/2
解説:n を n/2 越の値で割ったものは 2 未満になる。即ち、n を n/2 越 n 未満の値で割ったものは 1 越 2 未満となるから、割り切れることはない。
(キ):n%i==0
(ク):素数ではない
(ケ):break
(コ):最後まで割り切れなかったなら、素数である
(サ):System.out.println(n)
(シ): System.out
(蛇足:(シ)の処理に本質的な意味はないので、屁理屈を許せば「//」と回答しても不可ではないだろう)

答え 4-2
(答え 4-1 カ)の解説にも示した通り、
n を n/2 越 n 未満の値で割ったものは 1 越 2 未満となるから、割り切れることはない。
同じように、
n を n/3 越 n/2 未満の値で割ったものは 2 越 3 未満となるから、割り切れることはない。
このことから、i の範囲から n/3 越 n/2 未満を除外しても問題なさそうである。
ただし、そのようなアルゴリズムでは、n を何かで割ったものが 2 になる可能性を否定して

いない。その為、n が「整数 n/2」で割れるのか、即ち、n/2 が整数であるかを別途調べる必要がある。さらに、n=2 の場合は、自分自身で割れる分には素数候補から除外できないのだから、例外的に素数となることに注意する必要がある。

その為、次のように書き換える。
・i<=n/2 を i<=n/3 に書き換える
・int i の直前に
if(n!=2 && n%2==0) continue;
を加える
(i に関する for 文内でこれ(但し continue→break)を加えても動作する(だろう)が、わざわざ無駄な処理をさせる必要もない)

答え 4-3
2 と 3 が素数であることは予め示し、以降、過去に発見した素数を区間素数とすればよい。また、区間素数を無限に求めたとき、i の範囲の大きさは 0 となるが、これは素数を求めるときは、それより小さいすべての素数で割って確かめればよく、合成数で割る必要はないということを意味している。

```
class PrimeNumber3
{
    public static void main(String[] args)
    {
        int[] i = new int[500];//2 つに一つは偶数なので、1000 を 2 で割ってよい。
        //Java の仕様で、int 型の配列は 0 で初期化されている。
        System.out.println(i[0]=2);
        System.out.println(i[1]=3);

        int counter = 0;
        int numCounter = 2;
        boolean isPrime;
        int m;
        for(int n=4; n<=1000; n++)
        {
            isPrime = true;
            //最初は、n が素数であるとみなす。
            //(∵i の区間の大きさは原則 0、つまりすべて素数と判定)
```

```
                for(m=0; i[m]!=0; m++)
                //ただし、すべての(過去に見つかった)素数で
                {
                    if(n%i[m]==0)//n を割って、割り切れたことがある場合
                    {
                        isPrime = false;//素数ではないといえる
                        break;
                    }
                    counter++;
                }//割り切れたことがない場合は、素数であるといえるのでなにもしない。

                if(isPrime)//素数だったのなら
                {
                    i[m] = n;//素数として記憶し、
                    System.out.println(n);//表示する。
                    numCounter++;
                }

            }
            System.out.println("除算を行った回数:"+ counter);
            System.out.println("みつかった素数の個数:"+ numCounter);
        }
}
```

答え 5
(ア):ある条件を満たす
(イ):項目
(ウ):キー
(エ):ポインタ
(オ):ポインタでのり付けして繋げた
(カ):配列の先頭を削除したり追加したときに、以降を左や右に 1 つずつずらす手間を省く
(キ):各値(ノード)が枝分かれ(枝分かれ前を「親」、枝分かれ後を「子」ということにする)しているようなデータ構造で、
枝分かれは 0 回(そこで行き止まり。「葉」ともいう)

または 1 回(親から子に進むに際して枝分かれしない)
または 2 回(子はそれぞれ「左」、「右」と表現する)しか起こらないような構造を言う
(ク):ヒープ
(ケ):左の子及びその子孫は親より小さな値を、右の子及びその子孫は親より大きな値を持つような

答え 6
2、30 年前～最近流行りの開発手法。
とにかく簡単なアルゴリズムで、とりあえず動くものをとっとと作って初号機とするやり方。(PDCA サイクルが)速く回転することから「竜巻型」とも。
線形探索でさえも用いられる。
他のやり方としては、ウォーターオール(フロー)型。厳重な吟味の上開発するやり方。

答え 7
データやキーのハッシュ値を求め、それを比較する

答え 8
2 分探索でキーと中央値を比較する際には親を比較すればよく、左の子孫は必ず中央値以下であり、右の子孫は必ず中央値以上であるから、左右どちらかの子孫は注目しなくてもよいから。
解説:
2 分探索木は、任意の 3 値について、それらの中央値が親、最小値が左の子、最大値が右の子となるような構造である。したがって、キーと中央値を比較して、中央値以上のデータ、あるいは中央値以下のデータについて、またキーと中央値を比較することを繰り返す探索方法である 2 分探索では、親とキーを比較して、左右どちらかの子を親とみなし、キーと比較することを繰り返すことになる。
3 つの値を 2 分木の構造にすると、中央値が親となり、左の子は親以下の値、右の子は親以上の値となる。

答え 9
線形探索で、終了判定の処理を削減するための工夫。
最後に key 値(番兵)を、配列の最後に追加する。
こうすることで、線形探索は、終了判定をせずとも必ず key を見つけ、止まることが出来る。
key を見つけたとき、それが番兵なら、元の配列に key はなかったということになる。

答え 10

```
class HaystackNeedle
{
	static int strpos(String haystack, char needle)
	//乾草(haystack)の中から針(needle)を探すメソッド
	{
		//char[] arr = haystack.toCharArray();

		for(int i=0; ; i++)
		{
			if(i==haystack.length())
				return -1;
			if(haystack.toCharArray()[i]==needle)
		  //if(arr[i]==needle)
				return i;
		}
	}

	public static void main(String[] args)
	{
		String univ_name = "TokyoDenkiUniversity";
		char key = 'k';
		System.out.println(strpos(univ_name,key));
	}

}
```

答え 11

“TokyoDenkiUniversityk”

答え 14

(ア):0

(イ):n-1;

(ウ):

```
int pc = (pl+pr)/2;//探索範囲中央のインデックス
if(a[pc]==key)
   return pc;
else if (a[pc]<key)
   pl = pc+1;
else
   pr = pc-1;
```
(ここはできるようになるまで繰り返し)
(エ):(pl<=pr);
(オ):Scanner stdIn = new Scanner(System.in);
(カ):stdIn.nextInt()
(キ):
```
do
{
   System.out.print("x["+i+"]:");
   x[i]=stdIn.nextInt();
}
while(x[i]<x[i-1]);//一つ前の要素より小さければ再入力
```
(ク):x,num,ky
(ケ):要素 not found
(コ):found at" + idx +"

答え 15
合成数のエントロピーが高いからである。
ある問題のある素数で割った余剰に偏りがあることが発見されたとする。
他の素数で割った余剰をキーとしている場合は、この偏りの影響を受けえない。
しかし、合成数を基準にしている場合は、素因数に問題のある素数が含まれていた場合、その合成数でも影響を受けてしまう。このため、合成数は素数に比べて(素因数が多ければ多いほど)エントロピーが大きくなる。そのため、何らかの数字に問題があった場合に影響を受ける可能性や、その影響の大きさが大きくなってしまうのである。

答え 16
チェイン法:同一のハッシュ値を持つ要素を線形リストで管理する。
オープンアドレス法:空きバケット(ハッシュ表の要素のこと)を見つけるまで、ハッシュを繰り返す。

答え 29
スタック:LIFO、プッシュ・ポップ、頂上・底
キュー:FIFO、エンキュー・デキュー、先頭・末尾

答え 30
(ア):配列内の要素をずらさ
(イ):配列の末尾と先頭をつなげてわっかのようにしたもの
(ウ):front
(エ):rear-1
(オ):rear
解説:LIFO を思い出せ
(カ):rear
(キ):イン
(ク):front
(ケ):イン
補足:(カ)~(ケ)は暗記ではなく、LIFO をもとに考えて答えよ