4 Arrays

4.1 Why do programmers need arrays?

Arrays are useful for processing *many* data or generating *many* results at once into a compact contiguous structure. Loosely speaking, array structures allow one to manipulate *many* variables at once. An array is an *indexed* sequence of components. In mathematics, one is familiar with variables bearing indices like, for example, the vector coordinates x_i or matrix elements $m_{i,j}$. In most programming languages, indices start at zero and *not* at one as is often the case in mathematics. This simple 1-or-0 convention actually yields confusion to many novice programmers, and is therefore an important source of bugs to watch for.

4.2 Declaring and initializing arrays

4.2.1 Declaring arrays

In Java, arrays are also *typed* structures: For a given type, say `TYPE`, `TYPE[]` is the type of arrays storing a collection of homogeneous elements of type `TYPE`. Local arrays are declared within the body of functions (delimited by braces) as follows:

```
int [ ] x; // array of integers
```

F. Nielsen, *A Concise and Practical Introduction to Programming Algorithms in Java*, Undergraduate Topics in Computer Science, DOI 10.1007/978-1-84882-339-6_4,

```
boolean [ ] prime; // array of booleans
double [ ] coordinates; // arrays of reals formatted using double precision
```

Similarly, static class arrays are declared inside the body of a class using the keyword `static`:

```
class Example{
static int [ ] x;
static boolean [ ] prime;
...
}
```

These (class) static array variables can then be used and shared by *any* function of the class. In both local/class cases, arrays are allocated into the *global* memory, and not into the function call stack. Only the references of arrays may be stored into the function call stack for locally declared arrays.

4.2.2 Creating and initializing arrays

Arrays are *created* and *initialized* by default using the Java reserved keyword `new`:

```
int [ ] x = new int[5];
boolean [ ] prime = new boolean[16];
```

The size of arrays needs to be specified:

```
x=new int [32];
```

The size of an array can also be given as an integer arithmetic expression like `2*n`, etc:

```
x=new int [2*n+3];
```

Arrays can only be declared once but may eventually be created and initialized several times. This recreation process overrides the former creation/initialization:

```
x=new int [2*n+3];
...
x=new int [4*n-2]; // overrides the former creation and initialization
```

By default, initialization of arrays is performed by Java by filling all its elements with 0, or by casting this 0 to the equivalent array element type: For example, `false` for booleans, `0.0d` for `double`, `0.0f` for `float`, etc. Initialization can also be explicitly done by *enumerating* all its elements separated by comas "," within curly brackets {} as follows:

```
int [ ] prime={2, 3, 5, 7, 11, 13, 17, 19};
boolean prime[]={ false, true, true, true, false, true, false, true};
```

In that case, the size of the array is *deduced* from the number of elements in the set, and should therefore not be explicitly specified. Nor shall there be an explicit creation using the keyword `new`. Here are a few examples illustrating static array declarations, creations and initializations. Arrays can be declared within the body of a function or globally as a static variable of the class. To illustrate global static arrays, consider the following program:

Program 4.1 Static array declarations and creations

```
class ArrayDeclaration{
  static int digit [] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
  static double x [] = {Math.PI, Math.E, 1.0, 0.0};
  static boolean prime[]={ false, true, true, true, false,
     true, false, true, false, false };
  static int y[];
  static void MyFunction(int n)
  {
    // Allocate an array of size n
    y=new int[2*n];
  }
  public static void main (String[] args)
  {
  MyFunction(15);
  // We recreate and initialize array y;
  MyFunction(20); }
}
```

Observe that in this sample program, array `y` is created twice. In any case, arrays are allocated and stored in the global memory, and not stored in the local function call stack. Only the references of arrays are stored in the function stack for arrays declared within functions (without the keyword `static`).

4.2.3 Retrieving the size of arrays: `length`

Arrays in Java carry additional information[1] about themselves: Their types and lengths. The size of an array `array` is accessed by using the keyword `length`, post-appended to the array name with a "." dot:

```
array.length
```

Observe that there are no parenthesis used in conjunction with the keyword `length`.

```
static boolean prime[]={ false, true, true, true, false, true
    , false, true, false, false };
```

[1] Technically speaking, we say that arrays in Java are reflexive since they contain additional information. This is to contrast with arrays in C or C++ that are *non-reflexive* since they contain only their components.

```
System.out.println(prime.length);
```

We cannot change the size of arrays once initialized. For example, trying to force the length of array `array` by setting `array.length=23;` will generate the following compiler message error: `cannot assign a value to final variable length`.

4.2.4 Index range of arrays and out-of-range exceptions

Arrays created with the syntax `array=new TYPE[Expression]` have a fixed length determined at the instruction call time by evaluating the expression `Expression` to its integer value, say l (with `l=array.length`). The elements of that array are accessed by using an index ranging from 0 (lower bound) to $l-1$ (upper bound):

```
array[0]
...
array[l-1]
```

A frequent programming error is to try to access an element that does not belong to the array by giving an inappropriate index falling out of the range $[0, l-1]$. The following program demonstrates that Java raises an exception if we try to use out-of-range indices:

Program 4.2 Arrays and index out of bounds exception

```
class ArrayBound{
  public static void main (String[] args)
  {
    int [ ] v={0,1,2,3,4,5,6,7,8};
    long l=v.length;
    System.out.println("Size of array v:"+l);
    System.out.println(v[4]);
    System.out.println(v[12]);
  }
}
```

Running the above program yields the following console output:

```
Size of array v:9
4
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 12
        at ArrayBound.main(ArrayBound.java:8)
```

That is, index out of bounds cannot be checked by the compiler and may happen at *any* time when executing the program.

A *subarray* is a *consecutive portion* of an array: For example, `array[3..7];`. Java does not provide language support for manipulating subarrays so that one has to explicitly keep track of the lower and upper bounds of subarrays.

4.2.5 Releasing memory and garbage collector

In Java, one does not need to explicitly release memory of unused memory structures such as arrays. The Java virtual machine (JVM) does it *fully automatically* using the *garbage collector*. Once the JVM detects that elements of an array cannot be accessed anymore because the reference of that array has been released by, say, the function call stack, the garbage collector will free that memory. This is a key difference with another popular programming language: C++. The garbage collector checks at any time whether elements of a given array can still be accessed by some variables holding a reference to that array or not. If not, the garbage collector releases that global memory and will perform some memory cleaning operations. Nevertheless, we can also *explicitly* indicate to the JVM that we do not want the array anymore by setting the reference of that array to `null`, meaning that we erase the array reference:

```
int [] array=new int[32];
array=null; // explicitly indicate to the JVM to release the array
```

4.3 The fundamental concept of array references

Whether the array is declared as a local variable or as a global (static/class) variable, all its elements are stored in the program global memory. That is, even if local array variables are declared, created and initialized within a function, their elements may still be accessed by the calling function once the function is completed. This provides an essential mechanism for voluntarily having side-effect phenomena in functions that can therefore potentially change the (global) program environment. An array variable `array` is dealt as a *reference* to that array, a single machine word from which its indexed elements can be accessed. The notion of reference for non-primitive types in Java is essential. It can be quite delicate to grasp at first for novice programmers but nevertheless is essential. The main advantages of handling array variables (whatever their sizes) as references (a single machine word using four bytes[2]) are as follows:

- References provide a mechanism for functions to access and modify elements of arrays that are preserved when functions exit.
- When calling a function with array arguments, Java does not need to allocate the full array on the *function call stack*, but rather pass a single reference to that array. Therefore it is computationally and memory efficient.

[2] That is equivalently 32 bits to reference a given memory location.

Furthermore, this pass-by-reference mechanism limits the risk of function stack overflow.

To illustate the notion of array references, consider the following set of instructions:

Program 4.3 Arrays and references

```
int [ ] v = {0, 1, 2, 3, 4};
// That is, v[0]=0, v[1]=1, v[2]=2, v[3]=3, v[4]=4;
int [ ] t =v;
// Reference of t is assigned to the reference of v so that t
    [i]=v[i]
t[2]++; // Post-incrementation: t[2]=v[2]=3
System.out.println(t[2]++);
// Display 3 and increment t[2]=v[2]=4 now
```

The result displayed in the console is 3. In summary, an array is allocated as a single contiguous memory block. An array variable stores a reference to the array: This reference of the array links to the symbolic memory address of its first element (indexed by 0).

Program 4.4 Assign an array reference to another array: Sharing common elements

```
class ArrayReference{
  public static void main (String[] args)
  {
    int [] v={0,1,2,3,4};
    System.out.println("Reference of array u in memory:"+v);
    System.out.println("Value of the 3rd element of array v:"
       +v[2]);
    // Declare a new array and assign its reference to the
       reference of array v
    int [] t =v;
    System.out.println("Reference of array v in memory:"+v);
       // same as u
    System.out.println(v[2]);
    t[2]++;
    System.out.println(v[2]);
    v[2]++;
    System.out.println(t[2]);
  }
}
```

Running this program, we notice that the reference of the array u coincides with the reference of array v:

```
Reference of array u in memory:[I@3e25a5
Value of the 3rd element of array v:2
Reference of array v in memory:[I@3e25a5
2
```

```
3
4
```

The `I` in `[I@3e25a5` indicates that this is a reference to an array of integers. The forefront letter varies according to the type of array elements as illustrated by the following code:

Program 4.5 Printing the references of various typed arrays

```
class ArrayDisplay{
  public static void main (String[] args)
  {
  int [] x=new int[10]; System.out.println(x);
  double [] y=new double[20]; System.out.println(y);
  float [] z=new float[5]; System.out.println(z);
  boolean [] b=new boolean[7];System.out.println(b);
  }
}
```

We get[3]:

```
[I@3e25a5
[D@19821f
[F@addbf1
[Z@42e816
```

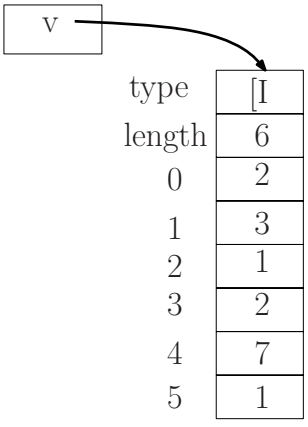

Figure 4.1 A way to visualize arrays in Java, explained for the array declaration and assignment: **int** [] v={2,3,1,2,7,1};.

One way to depict arrays is shown in Figure 4.1. Note that the length and type of the array are indicated in this representation (reflexive arrays).

[3] An array of strings has type `[Ljava.lang.String;`

4.4 Arrays as function arguments

Functions and procedures can have arrays as arguments too. Remember that arrays of element types `TYPE` are themselves of type `TYPE []`, so that the syntax for declaring a function using an array argument and calling it is:

```
static void MyFunction(int [ ] x)
{...}
...
int [] v=new int[10];
// Calling the function with an array argument
MyFunction(v);
```

For example, consider implementing a function that returns the minimum element of an array of integers provided as a function argument:

Program 4.6 Array argument in functions: Minimum element of an array

```
class ArrayMinimum{
  static int minArray(int [] t)
  {
    int m=t[0];
    for(int i=1;i<t.length; ++i)
      if (t[i]<m)
        m=t[i];
        return m;
  }

  public static void main(String[] args)
  {
  int [] v=new int [23];

  for(int i=0;i<23;i++)
      v[i]=(int)(Math.random()*100); // int from 0 to 99

  System.out.println("The minimum of the array is :"+minArray
      (v));
  }
}
```

Since we initialize the array by filling it with random elements using the `Math.random()` function, running the code several times may yield different outputs. For example, running the compiled bytecode three times in a row yields the following results:

```
The minimum of the array is :4
The minimum of the array is :2
The minimum of the array is :1
```

We say that the code is *non-deterministic* because it uses some randomness[4] provided by the function `Math.random()`. The following example demonstrates that only references of arrays are passed by functions:

Program 4.7 Creating and reporting array information using functions

```
class ArrayInFunction{
  public static void MyFunction(int n)
  {
    int array []=new int [n];
    int i;
    InformationArray(array);
  }
  public static void InformationArray(int [] t)
  {System.out.println("Size of array given in argument is:"+t
      .length);}

  public static void main (String[] args)
  {
    MyFunction(2312);
    MyFunction(2008);
    int x []=new int [12];
  }
}
```

Running the program, we get:

```
Size of array given in argument is:2312
Size of array given in argument is:2008
```

Arrays are useful structures for storing coordinates of vectors. Let us consider programming the inner product of two vectors *modeled* as arrays. We end-up with the following code:

Program 4.8 Calculating the inner product of two vectors given as arrays

```
class VectorInnerProduct{

  static double innerproduct(int [] x, int [] y)
  {
  double sum=0.0;
  System.out.println("Dim of vector x:"+x.length+ " Dim of
      vector y:"+y.length);

  for(int i=0;i<x.length; ++i)
        sum=sum+x[i]*y[i];
  return sum;
  }

  public static void main(String[] args)
```

[4] Since the randomness is emulated by some specific algorithms, we prefer to use the term *pseudo-randomness*.

```
  {
  int dimension=30;
  int [] v1,  v2;

  v1=new int[dimension]; v2=new int[dimension];

  for(int i=0;i<dimension;i++)
    {v1[i]=(int)(Math.random()*100);// random int [0,99]
     v2[i]=(int)(Math.random()*100);// random int [0,99]
    }
  System.out.println("The inner product of v1 and v2 is "+
      innerproduct(v1,v2));
  }
}
```

Running this program, we get:

```
Dim of vector x:30 Dim of vector y:30
The inner product of v1 and v2 is 80108.0
```

Static (class) functions may also return an array as a result of their calculation. A typical example is the addition of two vectors that yields *another* vector of the same dimension:

Program 4.9 Function returning an array: Addition of vectors

```
class VectorAddition{

  static int [] VectorAddition(int [] u, int [] v)
  {
    int[] result=new int[u.length];

    for(int i=0;i<u.length;i++)
      result[i]=u[i]+v[i];

      return result;
  }

  public static void main(String[] args)
  {
  int [] x={1, 2, 3}; int [] y={4, 5, 6};
  int [] z= VectorAddition(x,y);

  for(int i=0;i<z.length;i++)
    System.out.print(z[i]+" ");
  }
}
```

The following example demonstrates how one can persistently modify inside a function the contents of an array passed as an argument. That is, this array element swapping program shows that the values of the elements of the array can be changed after exiting the function.

Program 4.10 Swapping array elements by calling a function

```
class ModifyArray{
static void swap(int [] t, int i, int j)
{
  int tmp;
  tmp=t[i];
  t[i]=t[j];
  t[j]=tmp;
}

static void DisplayArray(int [] x)
{ for(int i=0;i<x.length;i++)
    System.out.print(x[i]+" ");
  System.out.println();
}

public static void main(String[] args)
{
int [] t={1,2,3,4,5,6,7,8,9};
DisplayArray(t);
swap(t,2,3);
DisplayArray(t);
  }
}
```

We observe that the third and fourth element (corresponding respectively to index 2 and 3) have indeed been swapped:

```
1 2 3 4 5 6 7 8 9
1 2 4 3 5 6 7 8 9
```

4.5 Multi-dimensional arrays: Arrays of arrays

4.5.1 Multi-dimensional regular arrays

We have so far considered linear arrays (also called 1D arrays). These 1D arrays have proved useful for storing vector coordinates and processing arithmetic operations on them (see, for example, the former scalar product and vector addition programs). What about manipulating 2D matrices $M = [m_{i,j}]$ with n rows and m columns? Of course, once the dimensions n and m are known, we can map the elements $m_{i,j}$ of a 2D matrix to a 1D vector in $\mathbb{R}^{n \times m}$ by linearizing the matrix and using the following index correspondence:

$$(i, j) \Leftrightarrow i \times m + j.$$

This index remapping[5] is quite cumbersome to use in practice and may yield various insidious bugs. Fortunately in Java, we can also create *multi-dimensional* arrays easily; Java will perform the necessary index remapping accordingly. A regular bi-dimensional array consists of n lines, each line being itself an array of m elements. A 2D matrix of integers has type `int [] []` and is declared, created and initialized as follows:

```
int [ ] [ ] matrix;
matrix=new int[n][m];
```

By default at the initialization stage, the array `matrix` is filled up with all zero elements. We can change the contents of this 2D array using two *nested* loops, as follows:

```
for(int i=0; i<n; i++)
  for(int j=0; j<m;j++)
      matrix[i][j]=i*j+1;
```

These constructions extend to arbitrary array dimensions. For example, a 3D array may be defined as follows:

```
int [ ] [ ] [] volume;
volume=new double[depth][height][width];
```

Let us illustrate the manipulations of linear and bi-dimensional arrays by implementing the matrix vector product operation. Observe the declaration/creation and initialization of a 2D array by enumerating all its elements:

```
int [][] M={{1, 2, 3}, {4,5,6}, {7,8,9}};
```

Program 4.11 Matrix-vector product function

```
class MatrixVectorProduct{
  static int [] MultiplyMatrixVector(int [][] mat, int [] v)
  {
    int[] result;
    result=new int[mat.length];

    for(int i=0;i<result.length;i++)
    {
      result[i]=0;
      for(int j=0;j<v.length;j++)
        result[i]+= mat[i][j]*v[j];
    }
      return result;
  }
  public static void main(String[] args)
  {
    int [][] M={{1, 2, 3}, {4,5,6}, {7,8,9}};
    int [] v={1,2,3};
```

[5] We arbitrarily chose row major order. We can also choose the column major order.

```
    int [] z= MultiplyMatrixVector(M,v);

  for(int i=0;i<z.length;i++)
    System.out.print(z[i]+" ");
  }
}
```

Thus it is quite easy to write basic functions of *linear algebra*. Note that in Java, it is not necessary[6] to provide the function with the array dimensions since we can retrieve these dimensions with the `length` keyword, as shown below:

Program 4.12 Creating multidimensional arrays and retrieving their dimensions

```
class MultidimArrays
{
static void f2D(double [] [] tab)
{
  System.out.println("Number of lines:"+tab.length);
  System.out.println("Number of columns:"+tab[0].length);
}
static void f3D(double [] [] [] tab)
{
  System.out.println("Number of lines X:"+tab.length);
  System.out.println("Number of columns Y:"+tab[0].length);
  System.out.println("Number of depths Z:"+tab[0][0].length);
}
public static void main(String[] args)
{
  double [] [] var=new double[3][4];
  f2D(var);
  double [] [] [] tmp=new double[4][5][7];
  f3D(tmp);
}
}
```

Running this program, we see that we correctly retrieved the 2D and 3D array dimensions given as function arguments:

```
Number of lines:3
Number of columns:4
Number of lines X:4
Number of columns Y:5
Number of depths Z:7
```

4.5.2 Multi-dimensional ragged arrays **

Multi-dimensional arrays need not be regular: They can be completely irregular. That is, a multi-dimensional array can also be defined as a 1D array of arrays,

[6] In the C programming language, one *has* to pass these dimensions as arguments.

each array "element" being itself an array with its own dimensions. However, these arrays should all store the same type of elements. To create such *ragged* arrays, we first need to declare the 1D array of arrays, and then proceed by declaring each individual array using a loop statement. For example, to declare and create a 2D ragged array of integers, we write the following statements:

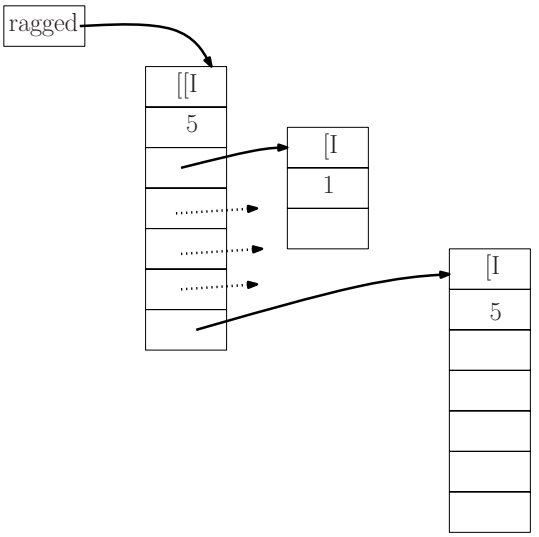

Figure 4.2 Visualizing the structure of a ragged array in the global memory

```
int ragged[][] = new int[5][];
for (int i = 0; i < 5; i++)
 {ragged[i] = new int[i + 1];}
```

The elements of the ragged arrays are either initialized by default (value zero) or by using nested loops as follows:

```
for (int i = 0; i < 5; i++)
{for (int j = 0; j < ragged[i].length; j++)
{        ragged[i][j] = (int)(10*Math.random());  // random init.
 }
}
```

Note that `ragged[i]` stores *references*[7] to linear arrays of integers. To visualize the entries of the ragged array, consider the following instructions:

```
System.out.println("type:"+ragged+" "+ragged.length);
 for (int i = 0; i < 5; i++)
  System.out.println("type:"+ragged[i]+" "+ragged[i].length);
```

[7] In general, the type of elements contained in the ragged array may be retrieved using `array.getClass();`

We get the following output:

```
type:[[I@addbf1 5
type:[I@42e816 1
type:[I@9304b1 2
type:[I@190d11 3
type:[I@a90653 4
type:[I@de6ced 5
```

Observe that array `ragged` is printed as a bi-dimensional array of integers using the "[[" notational convention: `[[I@addbf1`. Similarly to Figure 4.1, we can visualize ragged arrays as depicted in Figure 4.2.

4.6 Arrays of strings and `main` function

Strings of type `String` are *not* primitive types of Java. Though they can be constructed from literals and are immutable, strings are considered as special Java objects. Strings are *not* arrays of characters `char`. In other words,

$$\texttt{String} \neq \texttt{char []}.$$

These object notions shall be explained in the next chapter. We can also build arrays of strings that are of type `String []`, and functions may have string arrays as arguments. Actually, we are already very familiar with the `main` function of all Java programs that take as argument an array of strings:

```
class ProgramName
{
public static void main(String[ ] args)
{
...
}
}
```

For example, the following program lists all string arguments given in the line command when invoking the java virtual machine on the bytecode:

Program 4.13 Writing to the output the arguments of the invoked `main` function

```
class ParsingMain
{
  public static void main(String[] args)
  {
  for(int i=0;i<args.length;i++)
    System.out.println(i+":"+args[i]);
  }
}
```

After compiling this code, let us execute the bytecode using `java` as follows:

```
prompt%java ParsingMain Once upon a time there was a programming language
named Java!

0:Once
1:upon
2:a
3:time
4:there
5:was
6:a
7:programming
8:language
9:named
10:Java!
```

We can use the string array passed as argument of the `main` function of programs, to process inputs. Since these elementary inputs are considered as strings, we may eventually need to re-interpret them into the appropriate type before processing them. For example, consider the following program that seeks for the smallest integer entered in the arguments of the command line:

Program 4.14 Array of strings in `main`

```
class ParseArgumentsMin{
  public static void main(String[] args)
  {
    int indexMin=0;
    for(int i=1; i<args.length;i++)
    if (Integer.parseInt(args[indexMin])>Integer.parseInt(
        args[i]))
        indexMin=i;

    System.out.println("Maximum argument found at index:"+
        indexMin+" :"+args[indexMin]);
  }
}
```

Compiling and running this program with argument strings “345”, “546”, “234”, “2” and “45”, we get:

```
prompt%javac ParseArgumentsMinInt.java

prompt%java ParseArgumentsMin 345 546 234 2 45
Maximum argument found at index:3 :2
```

Once the strings are converted into corresponding integers using the library function `Integer.parseInt`, we get the index of the smallest argument: 2. Indeed `args[3]` corresponds to the string “2.”

4.7 A basic application of arrays: Searching **

Consider the following simple search problem encountered very day by in programmers: We are given a set $\mathcal{E}$ of n integers $\mathcal{E} = \{E_1, ..., E_n\}$, and we would like to know whether a given query element E belongs to that set or not: That is mathematically for short, $E \in \mathcal{E}$?. This search task is essential to decide whether we should add this element to the set or not. Let the data-structure for storing the n elements of $\mathcal{E}$ be an array named `array`.
The *sequential search* inspects in turn all the array elements `array[i]` and performs a comparison with the query element `E` to check for equality or not. If for a given index position i the query element matches the array element (that is, predicate `array[i]==E` is evaluated to `true`) then the element is found and the index of its position in the array is reported. Otherwise, we need to browse the full array before answering that `E` was not found in the array. This sequential search approach is summarized by the following program:

Program 4.15 Sequential search: Exhaustive search on arrays

```
class SequentialSearch{

static int SequentialSearch(int [] array, int key)
{int i;
for(i=0;i<array.length;i++)
  if (array[i]==key)
      return i;

return -1;
}

public static void main (String[] args)
{
int [] v={1,6,9 ,12 ,45, 67, 76, 80, 95};

System.out.println("Seeking for element 6: Position "+
    SequentialSearch(v,6));
System.out.println("Seeking for element 80: Position "+
    SequentialSearch(v,80));
System.out.println("Seeking for element 33: Position "+
    SequentialSearch(v,33));
  }
}
```

Running the program, we get the following output:

```
Seeking for element 6: Position 1
Seeking for element 80: Position 7
Seeking for element 33: Position -1
```

For query elements that are not present inside the array, we have to wait to reach the end of array to return -1. This explains why this sequential search is

also called the *linear* search since it takes time *proportional* to the array length. The algorithmic question raised is to know whether there exists or not a faster method? Observe that in the above program the array elements were ordered in increasing order. We should try to use this *extra* property to speed-up the search algorithm. The idea is to skip browsing some portions of the arrays for which we know that the query element cannot be found for sure. Start with a search interval [`left`, `right`] initialized with the extremal array indices: `left`$= 0$ and `right`$= n-1$ where n denote the array length `array.length`. Let `m` denote the index of the *middle* element of this range: $m = (\text{left} + \text{right})/2$. Then execute *recursively* the following steps:

- If `array[m]==E` then we are done, and we return index `m`,
- If `array[m] <E`, then if the element is inside the array, it is necessarily within range $[m+1, \text{right}]$,
- If `array[m] >E`, then if the element is inside the array, it is necessarily within range $[\text{left}, m+1]$.

The search algorithm terminates whenever we find the element, or if at some point `left>right`. In that latter case, we return index -1 for reporting that we did not find the query element. Thus the dichotomic search (also called binary search) is a provably *fast* method for searching whether or not a query element is inside a *sorted* array by successively halving the index range. The number of steps required to answer an element membership is thus proportional to $\log_2 n$. The dichotomic search is said to have *logarithmic time complexity*. These time complexity notions will be further explained in Chapter 6. We summarize the bisection search by the following code:

Program 4.16 Binary search: Fast dichotomic search on sorted arrays

```
class BinarySearch{
static int Dichotomy(int [] array, int left, int right, int
    key)
{
if (left>right)
    return -1;
int m=(left+right)/2;
if (array[m]==key)
  {return m;}
  else
  {
  if (array[m]<key) return Dichotomy(array,m+1, right, key);
    else  return Dichotomy(array,left,m-1, key);
  }
}
static int DichotomicSearch(int [] array, int key)
{return Dichotomy(array,0,array.length-1, key);}
public static void main (String[] args)
```

```
{
int [] v={1,6,9 ,12 ,45, 67, 76, 80, 95};
System.out.println("Seeking for element 6: Position "+
    DichotomicSearch(v,6));
System.out.println("Seeking for element 80: Position "+
    DichotomicSearch(v,80));
System.out.println("Seeking for element 33: Position "+
    DichotomicSearch(v,33));
  }
}
```

We get the following console output:

```
Seeking for element 6: Position 1
Seeking for element 80: Position 7
Seeking for element 33: Position -1
```

4.8 Exercises

Exercise 4.1 (Array of strings)

Write a static function `DisplayArray` that reports the number of elements in an array of strings, and displays in the console output all string elements. Give another function `DisplayReverseArray` that displays the array in reverse order.

Exercise 4.2 (Pass-by-value array arguments)

Explain why the following `permute` function does *not* work:

Program 4.17 Permuting strings and Java's pass-by-reference

```
class ExoArray{

static void permute(String s1, String s2)
{
String tmp=s1;
s1=s2;
s2=tmp;
}
public static void main(String args[])
{
String [] array={"shark", "dog", "cat", "crocodile"};
permute(array[0],array[1]);
System.out.println(array[0]+" "+array[1]);
}
}
```

Give a static function `static void permute(String [] tab, int i, int j)` that allows one to permute the element at index position i with

the element at index position j. Explain the fundamental differences with the former `permute` function.

Exercise 4.3 (Searching for words in dictionary)

Consider a dictionary of words stored in a plain array of strings: `String [] dictionary`. Write a function `static boolean isInDictionary` that takes as argument a given word stored in a `String` variable, and report whether the word is already defined inside the dictionary or not. Explain your choice for performing equality tests of words.

Exercise 4.4 (Cumulative sums: Sequential and recursive)

Write a function that takes a *single* array argument of elements of type `double`, and returns its cumulative sum by iteratively adding the elements altogether. Computing the cumulative sum of an array can also be done *recursively* by using, for example, the following function prototype `CumulativeSumRec(double array, int left, int right)`. Implement this function and test it using `CumulativeSumRec(array, 0, array.length-1);`

Exercise 4.5 (Chasing bugs)

The following program when executed yields the following exception:

```
Exception in thread "main" java.lang.NullPointerException
        at BugArrayDeclaration.main(BugArrayDeclaration.java:8)
```

Program 4.18 Bug in array declaration

```
class BugArrayDeclaration
{
public static void main(String [] t)
{
int [] array;
int [] array2=null;
array=array2;
array[0]=1;
}
}
```

Find the bug and correct the program so that it runs without any bug.

Exercise 4.6 (Sieve of Eratosthenes)

One wants to compute all prime integers falling within range $[2, N]$ for a prescribed integer $N \in \mathbb{N}$. The sieve of Eratosthenes algorithm uses a boolean array to mark prime numbers, and proceeds as follows:

– First, the smallest prime integer is 2. Strike off 2 and all multiples of 2 in the array (setting the array elements to `false`),
– Retrieve the smallest remaining prime number p in the array (marked with boolean `true`), and strike off all multiples of p,
– Repeat the former step until we reach at some stage $p > \sqrt{N}$, and list all prime integers.

Design a function `static int[] Eratosthene(int N)` that returns in an integer array all prime numbers falling in range $[2, N]$.

Exercise 4.7 (Image histogram)

Consider that an image with grey level ranging in $[0, 255]$ has been created and stored in the regular bi-dimensional data-structure `byte [] [] img;`. How do we retrieve the image dimensions (width and height) from this array? Give a procedure that calculates the histogram distribution of the image. (*Hint:* Do not forget to perform the histogram normalization so that the cumulative distribution of grey colors sums up to 1.)

Exercise 4.8 (Ragged array for symmetric matrices)

A d-dimensional symmetric matrix M is such that $M_{i,j} = M_{j,i}$ for all $1 \leq i, j \leq d$. That is, matrix M equals its transpose matrix: $M^T = M$. Consider storing only the elements $M_{i,j}$ with $d \geq i \geq j \geq 1$ into a *ragged* array: `double [] [] symMatrix=new double [d][];`. Write the array allocation instructions that create a 1D array of length i for each row of the `symMatrix`. Provides a static function that allows one to multiply two such symmetric matrices stored in "triangular" bi-dimensional ragged arrays.

*Exercise 4.9 (Birthday paradox **)*

In probability theory, the birthday paradox is a mathematically well-explained phenomenon that states that the probability of having at least two people in a group of n people having the same birthday is above $\frac{1}{2}$ for $n \geq 23$. For $n = 57$ the probability goes above 99%. Using the `Math.random()` function and a `boolean` array for modeling the 365 days, simulate the birthday paradox experiment of having at least two people having the same birthday among a set of n people. Run this birthday experiment many times to get empirical probabilities for various values of n. Then show mathematically that the probability of having at least two person's birthdays falling the same day among a group of n people is exactly $1 - \frac{365!}{365^n(365-n)!}$.