Data Structures & Algorithms 1

Topic 4 – Arrays and Array Algorithms

Arrays

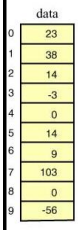
Say that you are writing a program that reads in 100 numbers for data.

Would you like to declare 100 variables and write 100 input statements to read in the data?

Even if it was 6 numbers, it's not too efficient to declare 6 separate variables, provided they are of the same type.

The solution to grouping large numbers of variables together is to use arrays

Arrays



An **array** is an object that is used to store a list of values

It is made out of a contiguous block of memory that is divided into a number of "slots"

Each slot holds a value, and all the values are of the same type. In the example array here, each slot holds an `int`

Arrays have names, for example this one is called `data`

The slots are indexed 0 through 9. Each slot can be accessed by using its **index**. For example, `data[0]` is the slot which is indexed by zero (which contains the value 23). `data[5]` is the slot which is indexed by 5 (which contains the value 14)

Arrays

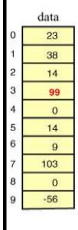
Important:

The slots are numbered sequentially starting at zero.

If there are N slots in an array, the indexes will be 0 through N-1

If you write a for loop cycling through all of the slots in an array, make sure it stops at N-1

Using Arrays



Every slot of an array holds a value of the same type.

For example, you can have an array of `int`, an array of `double`, and so on.

This array holds data of type `int`. Every slot may contain only an `int`.

A slot of this array can be used anywhere a variable of type `int` can be used.

```
data[3] = 99 ;
```

Using Arrays

Any of the array entries (or *elements*) can be used exactly the same way as a standard variable, including arithmetic expressions.

For example, if x contains a 10, then

```
(x + data[2]) / 4
```

evaluates to

```
(10+14) / 4 == 6
```

Using Arrays

Here are some other legal statements:

```
data[0] = (x + data[2]) / 4 ;

data[2] = data[2] + 1;

x = data[3]++ ;
// data in slot 3 is incremented

data[4] = data[1] / data[6];
```

Declaring Arrays

Array declarations look like this:

```
type[] arrayName = new type[ length ];
```

This names the type of data in each slot and the number of slots.

Once an array has been constructed, the number of slots it has does **not** change.

Declaring Arrays

Examples:

```
int[] myArray = new int[20];

double[] theArray = new double[5];

String[] words = new String[17];

char[] charArray = new char[256];
```

Array Boundary Checking

The **length** of an array is how many slots it has. An array of length N has slots indexed 0..(N-1)

Indexes must be an integer type. It is OK to have spaces around the index of an array

For example `data[1]` and `data[ 1 ]` are exactly the same as far as the compiler is concerned

It is *not legal* to refer to a slot that does not exist

Array Boundary Checking

Say that an array was declared:

```
int[] data = new int[10];
```

Here are some elements of this array, are they valid?

```
data[ -1 ]
data[ 10 ]
data[ 1.5 ]
data[ 0 ]
data[ 9 ]
```

Array Boundary Checking

```
int[] data = new int[10];

data[ -1 ] always illegal
data[ 10 ] illegal (given the above declaration)
data[ 1.5 ] always illegal
data[ 0 ] always OK
data[ 9 ] OK (given the above declaration)
```

Array Boundary Checking

```
Error line 17: ArrayIndexOutOfBoundsExceptionError
```

This means you've overstepped the boundaries of the array

You have used either an index less than 0, or greater than N-1, where N is the length of the array.

This problem is only revealed when you run the program, not when you compile

Variables as Index Values

The index of an array is always an integer type

This means it can be any expression that evaluates to an integer. For example, the following are legal:

```
int[] values = new int[7];
int index = 0;
values[ index ] = 71; // put 71 into slot 0
index = 5;
values[ index ] = 23; // put 23 into slot 5
index = 3;
values[ 2+2 ] = values[ index-3 ];
//same as values[ 4 ] = values[ 0 ];
```

Variables as Index Values

Using an expression for an array index is a very powerful tool

Often a problem is solved by organizing the data into arrays, and then processing that data in a systematic way using variables as indexes. Here are further examples:

```
double[] val = new double[4];
val[0] = 0.12;
val[1] = 1.43;
val[2] = 2.98;
int j = 2;
System.out.println("slot 2:" + val[j] );
System.out.println("slot 1:" + val[j-1] );
j = j-2;
System.out.println("slot 0:" + val[j] );
```

Initial Values

- When array is created, all values are initialized depending on array type:
  - Numbers: 0
  - Boolean: `false`
  - Object References: `null`

Array initialisation as a list

You can declare, construct, and initializese the array all in one statement:

```
int[] data = {23,38,14,-3,0,14,9,103,0,-56};
```

This declares an array of `int` which is named data. Then it constructs an `int` array of 10 slots (indexed 0..9)

Finally it puts the designated values into the slots. The first value in the **list** corresponds to index 0, the second value corresponds to index 1, and so on

So in this example, `data[0]` gets the 23

Copying Arrays

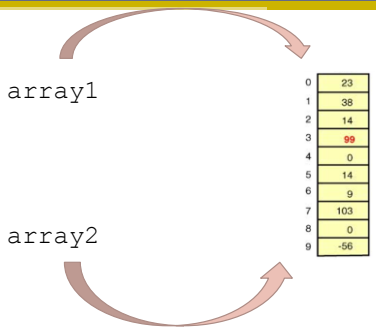
Say we have two arrays:

```
int[] array1 = {17,12,32,103,5};
int[] array2 = {22,57,13,203,15};
```

How do we copy the contents of array1 into array2?

Can we just do this?

```
array2 = array1;
```



We just get two references to the same array

Copying Arrays

This does not work!

~~array2 = array1;~~

Never do this! Worst of all, it does not cause an error, so remember it!!!

Arrays must be dealt with on an element by element basis

Copying Arrays

You must copy all the elements one by one

How about...

```
array2[0] = array1[0];
array2[1] = array1[1];
array2[2] = array1[2];
array2[3] = array1[3];
array2[4] = array1[4];
```

This will work, but it's a little inefficient, isn't it?
We can produce the same effect using a loop

Copying Arrays

Arrays must be of the same type...

```
double[] array1 = {9,8,7,6,5,4,3,2,1,0};
double[] array2 = new double[10];

for(int i = 0; i < array1.length; i++){

      array2[i] = array1[i];

}
```

Printing Arrays

To print any array, it's just the same...

```
for(int j = 0; j < array.length; j++)
{
      System.out.println(array[j]);

}
```

Arrays and Loops

THINK OF FOR LOOPS!

Why? Because `for` loops execute for an exact number of times, no more, no less

This is tailor made for arrays which are always of a definite size

Array Length

If we are uncertain about the size of an array, we can use array.length to get it

Because arrays are a fundamental data type, we get the length using the statement

```
int length = array.length;
```

Because Strings are a class, when we get the length of a String we are calling a method and must provide brackets

```
int length = message.length();
```

Exercise

Write a program that:

- takes the array size as input from the user,
- creates an `int` array of that size,
- populates it with values, prompting the reader for each value.

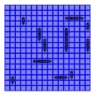
Program

```
import java.util.Scanner;

public class loop {
  public static void main(String[] args) {
    Scanner kbinput = new Scanner(System.in);

    System.out.println("Please enter array size");
    int size = kbinput.nextInt();
    int[] array = new int[size];

    for(int i = 0; i < size; i++){
      System.out.println("Enter array value" +i);
      array[i]= kbinput.nextInt();
    }
  }
}
```

Nested Loops

- This code uses nested for loops to print out each name in each slot, one character at a time
  - The outer loop selects a name in a particular slot
  - The inner loop prints out each character of that name, one at a time

```
String[] names = {"Peter", "Susan", "Keith"...};
for(int i = 0; i < names.length; i++){
  for(int j = 0; j < names[i].length(); j++){
    System.out.print(names[i].charAt(j) + " ");
  }
  System.out.println();
}
```

2D Arrays

Often data comes naturally in a two dimensional form.

For example, maps are two dimensional, the layout of a printed page is two dimensional, a computer-generated image (such as on your computer screen) is two dimensional, and so on.

Think Battleships, or chess in a newspaper, or reading a map. It's always just rows by columns, x by y, whatever way you want to think of it...

So, instead of one value to specify an array element or slot, we now need two

Two Dimensional Arrays

- A single dimensional stores data as a list

[1 2 3 4 5 6 7 8 9]

- A two dimensional array stores data using two separate indices – like a rectangle

[1 2 3
4 5 6
7 8 9]

2D Arrays

Student	Week				
	0	1	2	3	4
0	99	42	74	83	100
1	90	91	72	88	95
2	88	61	74	89	96
3	61	89	82	98	93
4	93	73	75	78	99
5	50	65	92	87	94
6	43	98	78	56	99

```
int[][] gradeTable= new int[7][5]
```

```
gradeTable[0][1] //holds 42
gradeTable[3][4] //holds 93
gradeTable[6][2] //holds 78
```

2D Arrays

```
int[][] myArray = new int[3][5];
```

Will result in an array the same size as if we declared it as

```
int[][] myArray = {{8,1,2,2,9},{1,9,4,0,3},
{0,3,0,0,7}};
```

`myArray[2][4]` holds the value 7
`myArray[1][0]` holds the value 1

Remember, row first, then column

Initializing 2D Arrays

- Usually, the number of **rows** and **columns** will be stored in variables
- Sometimes you will want to fill an array with default values
- Sometimes you will want to search through the whole array for a particular value
- It is common to use two nested loops when filling or searching a two-dimensional array:

```
for (int i = 0; i < rows; i++)
   for (int j = 0; j < columns; j++)
      board[i][j] = " ";
   }
}
```

Initializing 2D Arrays

```
for (int i = 0; i < rows; i++)
   for (int j = 0; j < columns; j++)
      board[i][j] = " ";
   }
}
```

Lets say rows = 3 and columns = 3. Then this happens:

```
board[0][0] = " ";
board[0][1] = " ";
board[0][2] = " ";
board[1][0] = " ";
board[1][1] = " ";
board[1][2] = " ";
board[2][0] = " ";
board[2][1] = " ";
board[2][2] = " ";
```

Random Numbers

`Math.random()` provides a random number between 0.0 and 1.0

```
System.out.println("Here's one random number: " +
Math.random());

System.out.println("Here's another random number:
" + Math.random());
```

The random number that is generated is of type double. If you need an `int`, you have to cast it by putting `(int)` in front

Random Numbers

```
//how about an int in the range of 0 to 99?

int number = (int)(Math.random()*100.0);


//how about an int in the range 0 to 76?

int number2 = ((int)(Math.random()*77);
```

Fill an Array with Random Numbers

```
int[] randArray = new int [100];

for(int i = 0; i < randArray.length; i++)
{
      randArray[i] = (int)(Math.random()*100.0);
}

//Loops through 100 times and fills it in!
```

The Sieve of Eratosthenes

- The Sieve of Eratosthenes is a famous method for obtaining prime numbers
- Eratosthenes was a famous Greek mathematician (276 BC – 194 BC) born in Libya
- A prime integer is any integer that is only divisible by itself and 1
  - *2, 3, 5, 7, 11, 13, 17, 19, 23…*
- There is no simple way to predict which numbers are going to be prime without testing them using an algorithm

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Prime numbers

The Algorithm

- First step: create a boolean array with a size which corresponds to the range of numbers you want to check:

```
boolean sieve = new boolean[12];
```

- Put all of the values equal to true from 2 onwards

0	1	2	3	4	5	6	7	8	9	10	11
false	false	true	true	true	true	true	true	true	true	true	true

```
for(int i= 2; i<12; i++){
      myArray[i]=true;
}
```

The Algorithm

- Starting with the element in slot 2, check if the value in that slot is true – if not skip it and go onto the next number
- If so, it is a prime number – print it out...

0	1	2	3	4	5	6	7	8	9	10	11
false	false	true	true	false	true	false	true	false	true	false	true

2 is prime

- Now loop through the remainder of the array and set to 'false' every element whose slot number is a multiple of that slot number

The Algorithm

- For example, for the element in slot 2, all elements beyond 2 in the array that are multiples of 2 will be set to 'false' (e.g. slot numbers 4, 6, 8, 10 etc.)
- For slot number 3, all elements beyond 3 in the array that are multiples of 3 will be set to 'false' (e.g. slot numbers 6, 9, 12, 15 etc.)
- When you are finished any slot which still contains 'true' must be a prime number

0	1	2	3	4	5	6	7	8	9	10	11
false	false	true	true	false	true	false	true	false	false	false	true

Algorithm Structure

- Use a nested `for` loop
- The outer loop loops through all the numbers from two onwards checking if they are true or false
- The inner loop figures out all of the multiples and sets the contents of those slot numbers to false
- The inner loop goes to the end of the array and goes up in jumps of the multiple

Randomize an Array

- Say we want to mix up all the elements in an array
- How can we swap all of the elements around in a random order?
  - e.g. shuffling a deck of cards

0	1	2	3	4

- Java lets us select a random number in a range
- Math.random( ) returns a random floating point number between 0 and 1
- We can cast it into an `int` and then multiply by 5 to give us a number between 0 and 4

```
int randomnumber = (int)(Math.random()*5);
```

Random Swaps

- A good way to randomize the array is actually to go through each item and swap it with another random item
- [0 1 2 3 4]
- [4 1 2 3 0]
- [4 0 2 3 1]

- Advantages:
  - We don't need to create another array and waste space
  - We only need to make one copy of a variable at a time in order to swap it
  - The final ordering will be completely random
- Disadvantages:
  - Some numbers may be swapped multiple times

Swapping

- Lets swap slot 1 with slot 4

1. Backup slot 1 into temp [0 1 2 3 4] 1
2. Copy slot 4 into slot 1 [0 4 2 3 4] 1
3. Copy temp into slot 4 [0 4 2 3 1] 1

Swapping

- In order to swap one variable with another in an array
  - Back-up variable #1 (the one that will be overwritten first) into a temporary variable
  - Overwrite variable #1 with the value of variable #2
  - Use the temporary variable to overwrite the value of variable #2 with the original value of variable #1

```
int temp = array[i];
array[i]=array[random];
array[random]=temp;
```

Monte Carlo method

- Often it is complicated to work out precise mathematical formulae which describe how a system works
- The lazier solution is simply to model the system and run the simulation many times randomly
- You base the probability on what you observe, letting the simulation do the hard work

Monte Carlo method

- Even the value for PI can be calculated using a Monte Carlo method
- Draw a square on the ground and inscribe a circle in it
- Scatter some grains of rice randomly throughout the square (or count rain drops falling into it etc.)
- The ratio of grains of rice in the square to grains of rice in the circle will be the ratio of their areas, or PI/4

Array Linear Search

- To find an item in an array, start at the beginning and check every item

```
public int search (int searchKey){

  for(int j=0; j<array.length; j++){
     if(array[j] == searchKey){
          return j;
     }
  }
  return null;
}
```

Counting matches

- Count the number of items with a searchKey greater than a specified threshold

```
public int countMatches (int threshold){
  int count=0;
  for(j=0; j<array.length; j++){
     if(array[j] > threshold){
          count++;
     }
  }
  return count;
}
```

Finding the Maximum or Minimum

- Algorithm:
  - Initialize a candidate with the starting element
  - Compare candidate with remaining elements
  - Update it if you find a larger or smaller value

Find Biggest

- Find the biggest value in the array
- Go through every element and track biggest found so far

```
public int findMax(){
  int biggestSoFar = array[0];
  for(int j=1; j<array.length; j++){
     if(array[j] > biggestSoFar){
          biggestSoFar=array[j];
     }
  }
  return biggestSoFar;
}
```

Inserting into an Array

- Arrays have fixed size and will usually will not be filled to capacity
- Some slots will be filled whereas others will be empty

[4 6 2 7 9 8 _ _ _ _ _]

- When a new element is added, it makes sense to add it to the next available free slot

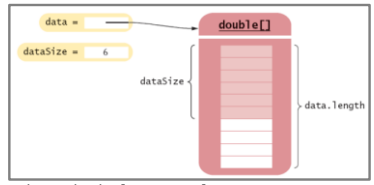
Inserting into an Array

- If we know how many elements are in the array then we know what the next available slot number is
- We use a variable to track how many elements are currently in the array
- For example, if dataSize = 6 this means there are six elements in the array and the next available slot will be the seventh slot, namely data[6]

```
final int LENGTH = 100;
double[] data = new double[LENGTH];
int dataSize = 0;
```

Inserting into an Array

A Partially Filled Array



- Next element inserted goes in slot [dataSize]
- Update dataSize as array is filled:

```
public void insert (int value){
       data[dataSize] = value;
       dataSize++;
}
```

Growing an Array

- If the array is full and you need more space, you can grow the array:
  - Create a new, larger array

```
double[] newData = new double[2 * data.length];
```

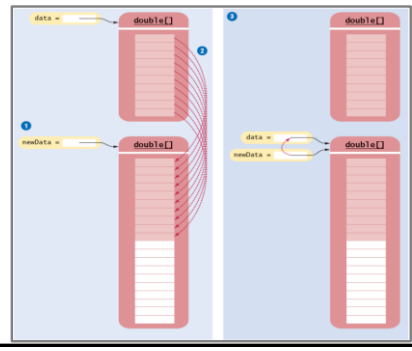
  - Copy all elements into the new array

```
for(int i=0;i<data.length;i++){
  newData[i]=data[i];
}
```

  - Change the reference to point to the new array

```
data = newData;
```

Growing an Array



Problems with Arrays

- You need to keep track of what slot in the array is the next free one
- You need to write special code to search and delete a particular element
- Every time you want to find an item, you have to check EVERY item
- Every time you want to delete an item you have to check EVERY item
- As the array gets bigger and bigger it will take longer and longer to find what you want
- Imagine looking for a word in a dictionary and having to check every word!

Ordered Arrays

- When you have to check every item this is known as *linear search*
- We notice that dictionaries and telephone directories are *ordered*
- This makes it easier to find stuff we're looking for
- If information is ordered then you can use a *binary search*

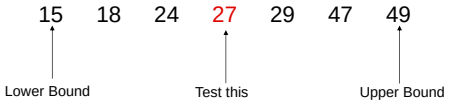
Ordered Arrays

- If an array is in order and we want to search for a particular entry then we just play the guessing game
- We try the middle element first (like guessing 50 for a number between 1 and 100)
- If the middle element is smaller than the one we're looking for, we know that the element must be in the second half
- If the middle element is bigger then the one we're searching for must be in the first half

Ordered Arrays

Computer: Guess my number between 1 and 100
You: 50
Computer: Too low!
You: Aha, the number must be between 50 and 100. I guess 75
Computer: Too high!
You: Aha, the number must be between 50 and 75. I guess 63 which is in the middle again.
Computer: Too low!
You: Now I'm getting close. The number must be between 63 and 75. How about 69?
Computer: Too high!
You: 66
Computer: You got the correct answer! There were 100 numbers but you guessed in only 5 guesses!

What do we need?

- We keep dividing our search space and therefore need to keep track of the limits
  - Upperbound
  - Lowerbound
- If the number is bigger than 27 then 27 is the new lower bound
- 

Algorithm Description

- Objective : find the array slot in which a certain searchKey value is contained
- Identify the range of slots in which the value could possibly turn up
- Keep doing this while the search space is bigger than 0:
  - Check the slot in the middle of the range
  - If this slot holds the correct value, return that slot number
  - If not, adjust the search range and repeat the process

Code

- In the following code we use the following variables:
  - `searchKey` is the number we're looking for
  - `nElems` is the number of elements in the array (it might not be full)
  - `lowerBound` and `upperBound` are used to track the range of our search
  - `check` is used to store the slot number we are currently checking
  - `myArray` is the array we're searching through

```
public int find(int searchKey){

     int lowerBound = 0;
     int upperBound = nElems-1;
     int check;

     while(true){
        check = (lowerBound + upperBound ) / 2;
        if(myArray[check]==searchKey){
           return check;              // found it
        }else if(lowerBound > upperBound){
           return -1;              // can't find it
        }else{                        // divide range

           if(myArray[check] < searchKey){
              lowerBound = check + 1; // it's in upper half
           }else{
              upperBound = check - 1; // it's in lower half
           }

       }  // end else divide range
    }  // end while
}  // end find()
```

Keeping things ordered

- In order to be able to run a binary search like this, the array we're working with has to be sorted
- Now we need new algorithms to keep our array sorted
- Whenever a new number is inserted, it has to be inserted into the correct place
- Whenever a number is removed, the gap it leaves behind has to be filled

Inserting an element

- We need to insert an element according to its order
- This means we will have to move all the other elements up to make room

[2 4 6 7 8 9 _ _ _ _ _]

- Say we want to insert the number 5
- This should go in the third slot (between 4 and 6)
- We need an algorithm that is going to shuffle all the elements from slot 2 onwards one space to leave a gap

Inserting an Element

Make a gap in the array by shifting everything up

2 4 6 7 8 9 _ _ _ _

```
//lets make space to insert something into slot 2

for(int j=dataSize;j>2;j--){
  data[j]=data[j-1];
}
```

2 4 _ 6 7 8 9 _ _ _

Full insertion method

```
public void insert(int value){

   int j=0;
  while(array[j] < value &&j<nElems){  //find where it goes
     j++;                               //linear search
   }
  for(int k=nElems; k>j; k--){       // move bigger ones up
      a[k] = a[k-1];
   }
   a[j] = value;                     // insert it
   nElems++;                         // increment size
}                                    // end insert()
```

Removing an element

- Say we want to remove a particular element in our array
- Once we delete it there will be a gap left in our array
- If we don't keep track of these gaps, then the array will just fill up with holes like a Swiss cheese
- We need an algorithm that will move all the elements down to fill the gap that is created after one is removed

Removing an Element

Removing – squishes an existing element by shifting everything down

2 4 6 7 8 9 _ _ _ _

```
//delete something from slot 2

for(int j=2;j<dataSize;j++){
  data[j]=data[j+1];
}
```

2 4 7 8 9 9 _ _ _ _

Removing an element

```
public boolean delete(int value){

      int j=0;
      while(array[j] != value&&j<nElems){
       j++;
      }                                        // linear search
      if(j==nElems){                           // can't find it
         return false;
      }else{                                          // found it

         for(int k=j; k<nElems-1; k++){        // move values down
            array[k] = array[k+1];
        }
         nElems--;                                    // decrement
  size
         return true;
      }
}                                              // end delete()
```

Evaluation of Ordered Arrays

- Search process is much shorter (we can run a binary search)
- Insertion takes longer because we have to move elements up to make room rather than just sticking a new element at the end
- Deletion is slow for both ordered and unordered arrays since you have to move items down to get rid of gaps
- Ordered arrays are useful in situations where searches are frequent and insertions are not
  - Good for a shelf of books in a library
  - Not useful for a book jumble sale

How good is binary search?

- As arrays get bigger, using a binary search becomes more important
- A linear search would take ages!

Size of Array	Comparisons Needed
10	4
100	7
1,000	10
10,000	14
100,000	17
1,000,000	20
10,000,000	24
100,000,000	27
1,000,000,000	30

Mathematically

- The number of steps needed to perform a binary search on an array of size N is the number of times that N can be halved
- If N is 16 then 4 steps will be needed
  - Step 1: narrow search space down to 8 slots
  - Step 2: narrow search space down to 4 slots
  - Step 3: narrow search space down to 2 slots
  - Step 4: narrow search space down to 1 slot
- Each iteration of the binary search algorithm halves the search space that needs to be considered
- In other words, each extra iteration allowed doubles the range you can search through

A log relationship

- Each step halves the size, so the number of iterations needed to search through an array using a binary search is the number of times the size of the array can be halved

$$size = 2^{iterations}$$

- The opposite of raising something to a power is to take its log

$$iterations = \log_2(size)$$

- Number of steps required increases very slowly compared to increases in size – logarithmically as opposed to linearly
- We express this log type relationship between array size and number of steps required by saying that the complexity of binary search is O(log n)