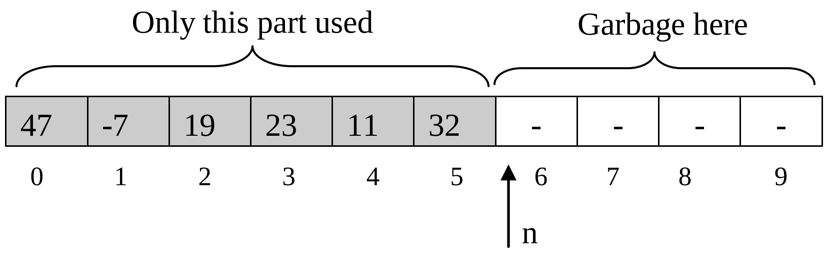
15 Array Algorithms

1 Partially filled arrays

Suppose we want to write a program which reads a list of integers and displays it in reverse order. Obviously we will store the integers in an array until we are ready to print them, but how big should we make the array? The problem is that don't know the ideal array size until after we have read in all the integers! This is a frequently occurring difficulty in programming, and the solution is important. (i) We choose some maximum size for the array. (ii) We keep the array partially unused at the right-hand end. (iii) If the array turns out to be too small, we typically abort the program. (iv) We employ an ancillary "count variable" to keep track of the number of elements in the array. The following is a typical declaration of an array for storing a list of integers of unknown size, together with the count variable:

```
int[] b = new int[10]; // allow for a max of 10 integers
int n = 0;
```

The count variable n is initially zero because b is empty. After 6 values have been read, say, b and n can be pictured thus:



Variable n has value 6, the values read are in the shaded part of b, and the rightmost 4 components of b contain garbage. Observe that the next available component in b is b[n], and that n must be incremented as each successive value is inserted.

The program to reverse the input follows.

```
class RevInts {
    public static void main(String[] args) {
        int[] b = new int[100]; // allow for a max of 100 integers
        int n = 0; // first n elements of b significant
        //Read list of numbers
        while (!Console.endOfFile()) {
            if (n==b.length) {
                System.out.println("Too many values");
                return; //abort execution
            }
            b[n] = Console.readInt();
            n++;
        }
        // Print in reverse order
        int i = n; // i = no. of values remaining to be printed
        while (i!=0) {
            i--;
            System.out.print(b[i] + " ");
        }
    }
}
```

Observe that we write b.length even though we know its value is 100. The advantage is that if we subsequently decide to change the size of b, only the declaration of b needs to be changed.

When an array parameter is allowed to be partially-filled, its count variable must also be a parameter – otherwise the method will not know which elements are significant and which are garbage. We illustrate this with an alternative coding of the preceding program:

```
class RevInts {

    static void putRev(int[] w, int k) { // print first k elements of w in reverse
        int i = k; // i = no. of values remaining to be printed
        while (i!=0) {
            i--;
            System.out.print(w[i] + " ");
        }
    }

    public static void main(String[] args) {
        int[] b = new int[100]; // allow for a max of 100 integers
        int n = 0; // first n elements of b significant
        //Read list of numbers
            .... as previously ...
        // Print in reverse order
        putRev(b, n); // print first n elements of b in reverse order
    }
}
```

The no-count-variable trap

Let variable b be an array of integers that may be partially filled. *You must use an associated count variable*. If you don't, you're writing a program that can't work. Let n be the count variable associated with b. You should be careful to distinguish between the capacity of b, which is b.length, and the current size of b, which is n (n varies of course as the computation progresses). Always, the first n elements of b are significant (i.e. we care about their values), and the remaining b.length-n elements are garbage. Many beginners fail to understand that the garbage values at the end of the array are just that – garbage. We do not care about them, we do not inspect or print them, and they play no part in the computation. Some beginners pointlessly (and misleadingly) initialise all the elements of the array (typically to 0 or –1, although all values are equally without merit). Although initialising the array elements to 0 is merely pointless, it is downright erroneous to believe that you can dispense with the count variable by testing array elements for 0 (or –1, or whatever initial value was used). It is erroneous, because significant values may well be 0. You *must* use a count variable. If you write a method which processes a partially-filled array supplied as a parameter, then the parameters must also include an associated count variable.

2 Linear search

Linear search is a technique for finding the first occurrence of a item satisfying some property in a list of items, or determining that no such item occurs in the list. Some examples of problems which are solved by linear search are:

(i) Find the first 0 in an integer array b.

(ii) Find the first upper case word in an array of strings

(iii) Determine whether an item satisfying some property occurs in an array.

The linear search algorithm is simple: begin at the beginning, and step through each element of the array one by one until either (i) the sought-for element is found or (ii) the array is exhausted. The coding is typified in the following which searches an integer array b for the location of the first zero:

```
int i = 0;
while (i<b.length && b[i]!=0)
        i++;
if (i<b.length) System.out.print(i);
else System.out.print("No zeros");
```

Do not use a for-loop to code linear search. Note that the loop guard has two terms, each capturing one of the two reasons for stopping the search. Note also that the loop body is just a simple increment. It is strongly recommended not to write linear search in any other way – it is easy to write a wrong linear search.

Here is another example: determine if an array b of positive integers contains a prime number.

```
int i = 0;
while (i<b.length && !isPrime(b[i]))
        i++;
if (i<b.length) System.out.println("Has a prime");
else System.out.println("No primes");
```

where isPrime is:

```
static boolean isPrime(int n) { // is n prime, n>0
        if (n==1) return false; // 1 is not prime
        else if (n%2==0) return n==2; // 2 is only even prime
        else { // try dividing n by 3, 5, 7, ..., √n
                int k = 3;
                while (k*k<=n && n%k!=0)
                        k = k+2;
                return k*k>n;
            }
        }
```

Observe that the loop in isPrime is itself an example of linear search – we search the sequence 3, 5, 7, ... for the first item that satisfies the property of being an exact divisor of n. Why does the search stop at the square root of n?

As a larger example, we write a program which reads a sequence of words and for each one determines whether it's the name of a day, month, or season. The program is terminated by entering a blank line.

```
class LinearSearch {

        public static void main(String[] args) {
           String[] ws =
                {"sunday", "monday", "tuesday", "wednesday",
                "thursday", "friday", "saturday", "january", "february",
                "march", "april", "may", "june", "july", "august", "september",
                "october", "november", "december", "spring", "summer", "autumn",
                "winter"};
           System.out.println("Enter a blank line to terminate.");
           System.out.print("Enter a word: ");
           String s = Console.readString();
           while (s.length()>0) {
              s = s.toLowerCase();
              // This is a linear search
              int i = 0;
              while (i<ws.length && !s.equals(ws[i])) {
                 i++;
              }
              if (i<ws.length) // found
                 System.out.println("That's a day, month, or season.");
              else // not found
                 System.out.println("That's not a day, month, or season.");
              System.out.print("Enter a word: ");
              s = Console.readString();
```

```
            }
        }
}
```

Observe the use of a common coding trick to handle the case of input words that use a mixture of upper and lower case letters. The words in the table are in lower case, and each word read is converted to lower case before searching for it.

3 Binary search

Binary search, for searching *sorted* arrays, has been described previously. The complete coding for an array of integers, say, follows (when searching and array of Strings or objects in general, .equals and .compareTo should be used in place of == and <):

```
int j = 0; int k = ws.length-1; int mid = (j+k)/2;
// array  segment to be searched is from index j to k, incl.
while (j<=k && !key==ws[mid]) {
    if (key<ws[mid]) // key < middle value
        k = mid-1; // eliminate top of segment
    else // key > middle value
        j = mid+1; // eliminate bottom of segment
    mid = (j+k)/2; // ready for next time around
}
if (j<=k)
    System.out.println(key + "present (at position " + mid + ")");
else
    System.out.println(key " not present");
```

4 Selection sort

Sorting by selection sort was described previously. The complete coding (for an array of strings) is:

```
// Sort array ws using selection sort
int j = 0; // ws[j..] remains to be sorted
while (j<ws.length-1) {
    // find minimum in ws[j..]
    int min = j;  int i = j+1; // ws[min] is minimum in ws[j..i-1]
    while (i<ws.length) {
        if (ws[i].compareTo(ws[min])<0) {
            min = i;
        }
        i++;
    }
    // ws[min] is minimum in ws[j..];  swap ws[j] and ws[min]
    String temp = ws[j]; ws[j] = ws[min]; ws[min] = temp;
    j++;
}
```