C Pointers

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using **ampersand (&)** operator, which denotes an address in memory.

Consider the following example, which will print the address of the variables defined:

```
#include <stdio.h>

int main ()
{
   int  var1;
   char var2[10];

   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var1 variable: bff5a400

Address of var2 variable: bff5a3f6
```

So you understood what is memory address and how to access it, so base of the concept is over. Now let us see what is a pointer.

What Are Pointers?

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk * you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to use Pointers?

There are few important operations, which we will do with the help of pointers very frequently. **(a)** we define a pointer variable **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

```
#include <stdio.h>

int main ()
{
   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */

   ip = &var;  /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var variable: bffd8b3c

Address stored in ip variable: bffd8b3c

Value of *ip variable: 20
```

NULL Pointers in C

It is always a good practice to assign a **NULL** value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned **NULL** is called a **null** pointer.

The **NULL** pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

```
#include <stdio.h>

int main ()
{
   int  *ptr = NULL;

   printf("The value of ptr is : %x\n", &ptr  );

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
The value of ptr is 0
```

On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer you can use an if statement as follows:

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

Pointer arithmetic

As explained in main chapter, C pointer is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer:

```
ptr++
```

Now, after the above operation, the **ptr** will point to the location 1004 because each time **ptr** is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to next memory location without impacting actual value at the memory location. If **ptr** points to a character whose address is 1000, then above operation will point to the location 1001 because next character will be available at 1001.

Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array:

```
#include <stdio.h>

const int MAX = 3;

int main ()
{
   int  var[] = {10, 100, 200};
   int  i, *ptr;

   /* let us have array address in pointer */
   ptr = var;
   for ( i = 0; i < MAX; i++)
   {

      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n", i, *ptr );

      /* move to the next location */
      ptr++;
   }
   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var[0] = bf882b30

Value of var[0] = 10

Address of var[1] = bf882b34

Value of var[1] = 100

Address of var[2] = bf882b38

Value of var[2] = 200
```

Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below:

```
#include <stdio.h>

const int MAX = 3;

int main ()
{
   int  var[] = {10, 100, 200};
   int  i, *ptr;

   /* let us have array address in pointer */
   ptr = &var[MAX-1];
   for ( i = MAX; i > 0; i--)
   {

      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n", i, *ptr );

      /* move to the previous location */
      ptr--;
   }
   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var[3] = bfedbcd8

Value of var[3] = 200

Address of var[2] = bfedbcd4

Value of var[2] = 100

Address of var[1] = bfedbcd0

Value of var[1] = 10
```

Pointer Comparisons

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

The following program modifies the previous example one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1]:

```
#include <stdio.h>
```

```
const int MAX = 3;

int main ()
{
   int  var[] = {10, 100, 200};
   int  i, *ptr;

   /* let us have address of the first element in pointer */
   ptr = var;
   i = 0;
   while ( ptr <= &var[MAX - 1] )
   {

      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n", i, *ptr );

      /* point to the previous location */
      ptr++;
      i++;
   }
   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var[0] = bfdbcb20
Value of var[0] = 10
Address of var[1] = bfdbcb24
Value of var[1] = 100
Address of var[2] = bfdbcb28
Value of var[2] = 200
```

Array of pointers

Before we understand the concept of **arrays of pointers**, let us consider the following example, which makes use of an array of 3 integers:

```
#include <stdio.h>

const int MAX = 3;

int main ()
{
   int  var[] = {10, 100, 200};
   int i;

   for (i = 0; i < MAX; i++)
   {
      printf("Value of var[%d] = %d\n", i, var[i] );
   }
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of var[0] = 10

Value of var[1] = 100

Value of var[2] = 200
```

There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer:

```
int *ptr[MAX];
```

This declares ptr as an array of MAX integer pointers. Thus, each element in ptr, now holds a pointer to an int value. Following example makes use of three integers, which will be stored in an array of pointers as follows:

```
#include <stdio.h>

const int MAX = 3;

int main ()
{
   int  var[] = {10, 100, 200};
   int i, *ptr[MAX];

   for ( i = 0; i < MAX; i++)
   {
      ptr[i] = &var[i]; /* assign the address of integer. */
   }
   for ( i = 0; i < MAX; i++)
   {
      printf("Value of var[%d] = %d\n", i, *ptr[i] );
   }
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of var[0] = 10

Value of var[1] = 100

Value of var[2] = 200
```

You can also use an array of pointers to character to store a list of strings as follows:

```
#include <stdio.h>

const int MAX = 4;

int main ()
{
   char *names[] = {
```

```
                    "Zara Ali",
                    "Hina Ali",
                    "Nuha Ali",
                    "Sara Ali",
   };
   int i = 0;

   for ( i = 0; i < MAX; i++)
   {
      printf("Value of names[%d] = %s\n", i, names[i] );
   }
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of names[0] = Zara Ali

Value of names[1] = Hina Ali

Value of names[2] = Nuha Ali

Value of names[3] = Sara Ali
```

Pointer to Pointer

A pointer to a pointer is a form of **multiple indirection**, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.

A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, following is the declaration to declare a pointer to a pointer of type int:

```
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example:

```
#include <stdio.h>

int main ()
{
   int  var;
   int  *ptr;
   int  **pptr;

   var = 3000;
```

```
   /* take the address of var */
   ptr = &var;

   /* take the address of ptr using address of operator & */
   pptr = &ptr;

   /* take the value using pptr */
   printf("Value of var = %d\n", var );
   printf("Value available at *ptr = %d\n", *ptr );
   printf("Value available at **pptr = %d\n", **pptr);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of var = 3000

Value available at *ptr = 3000

Value available at **pptr = 3000
```

Passing pointers to functions

C programming language allows you to pass a pointer to a function. To do so, simply declare the function parameter as a pointer type.

Following a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function:

```
#include <stdio.h>
#include <time.h>

void getSeconds(unsigned long *par);

int main ()
{
   unsigned long sec;


   getSeconds( &sec );

   /* print the actual value */
   printf("Number of seconds: %ld\n", sec );

   return 0;
}

void getSeconds(unsigned long *par)
{
   /* get the current number of seconds */
   *par = time( NULL );
   return;
}
```

When the above code is compiled and executed, it produces the following result:

```
Number of seconds :1294450468
```

The function, which can accept a pointer, can also accept an array as shown in the following example:

```
#include <stdio.h>

/* function declaration */
double getAverage(int *arr, int size);

int main ()
{
   /* an int array with 5 elements */
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;

   /* pass pointer to the array as an argument */
   avg = getAverage( balance, 5 ) ;

   /* output the returned value  */
   printf("Average value is: %f\n", avg );

   return 0;
}

double getAverage(int *arr, int size)
{
  int    i, sum = 0;
  double avg;

  for (i = 0; i < size; ++i)
  {
    sum += arr[i];
  }

  avg = (double)sum / size;

  return avg;
}
```

When the above code is compiled together and executed, it produces the following result:

```
Average value is: 214.40000
```

Return pointer from functions

As we have seen in last chapter how C programming language allows to return an array from a function, similar way C allows you to **return a pointer** from a function. To do so, you would have to declare a function returning a pointer as in the following example:

```
int * myFunction()
{
```

```
.
.
.
}
```

Second point to remember is that, it is not good idea to return the address of a local variable to outside of the function so you would have to define the local variable as static variable.

Now, consider the following function, which will generate 10 random numbers and returns them using an array name which represents a pointer, i.e., address of first array element.

```
#include <stdio.h>
#include <time.h>

/* function to generate and retrun random numbers. */
int * getRandom( )
{
   static int  r[10];
   int i;

   /* set the seed */
   srand( (unsigned)time( NULL ) );
   for ( i = 0; i < 10; ++i)
   {
      r[i] = rand();
      printf("%d\n", r[i] );
   }

   return r;
}

/* main function to call above defined function */
int main ()
{
   /* a pointer to an int */
   int *p;
   int i;

   p = getRandom();
   for ( i = 0; i < 10; i++ )
   {
       printf("*(p + [%d]) : %d\n", i, *(p + i) );
   }

   return 0;
}
```

When the above code is compiled together and executed, it produces result something as follows:

```
1523198053

1187214107

1108300978

430494959
```

```
1421301276
930971084
123250484
106932140
1604461820
149169022
*(p + [0]) : 1523198053
*(p + [1]) : 1187214107
*(p + [2]) : 1108300978
*(p + [3]) : 430494959
*(p + [4]) : 1421301276
*(p + [5]) : 930971084
*(p + [6]) : 123250484
*(p + [7]) : 106932140
*(p + [8]) : 1604461820
*(p + [9]) : 149169022
```

C Strings

The string in C programming language is actually a one-dimensional array of characters which is terminated by a null character '\0'. Thus a **null-terminated** string contains the characters that comprise the string followed by a null.

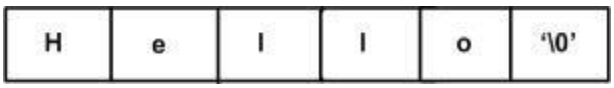
The following declaration and initialization create a string consisting of the word **"Hello"**. To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word **"Hello"**.

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization then you can write the above statement as follows:

```
char greeting[] = "Hello";
```

Following is the memory presentation of above-defined string in C/C++:



Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above mentioned string:

```
#include <stdio.h>

int main ()
{
   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

   printf("Greeting message: %s\n", greeting );

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Greeting message: Hello
```

C supports a wide range of functions that manipulate null-terminated strings:

S.N.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

Following example makes use of few of the above-mentioned functions:

```
#include <stdio.h>
#include <string.h>

int main ()
{
   char str1[12] = "Hello";
   char str2[12] = "World";
   char str3[12];
   int  len ;

   /* copy str1 into str3 */
   strcpy(str3, str1);
   printf("strcpy( str3, str1) :  %s\n", str3 );

   /* concatenates str1 and str2 */
   strcat( str1, str2);
   printf("strcat( str1, str2):   %s\n", str1 );

   /* total lenghth of str1 after concatenation */
   len = strlen(str1);
   printf("strlen(str1) :  %d\n", len );

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
strcpy( str3, str1) :  Hello
```

```
strcat( str1, str2):   HelloWorld
strlen(str1) :  10
```

You can find a complete list of C string related functions in C Standard Library.

C Structures

C arrays allow you to define type of variables that can hold several data items of the same kind but structure is another user defined data type available in C programming, which allows you to combine data items of different kinds.

Structures are used to represent a record, suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

- Title
- Author
- Subject
- Book ID

Defining a Structure

To define a structure, you must use the **struct** statement. The **struct statement** defines a new data type, with more than one member for your program. The format of the **struct statement** is this:

```
struct [structure tag]
{
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

The structure tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure:

```
struct Books
{
```

```
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

Accessing Structure Members

To access any member of a structure, we use the member access operator (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use **struct** keyword to define variables of structure type. Following is the example to explain usage of structure:

```
#include <stdio.h>
#include <string.h>

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( )
{
   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info */
   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);

   /* print Book2 info */
   printf( "Book 2 title : %s\n", Book2.title);
   printf( "Book 2 author : %s\n", Book2.author);
   printf( "Book 2 subject : %s\n", Book2.subject);
   printf( "Book 2 book_id : %d\n", Book2.book_id);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Book 1 title : C Programming

Book 1 author : Nuha Ali

Book 1 subject : C Programming Tutorial

Book 1 book_id : 6495407

Book 2 title : Telecom Billing

Book 2 author : Zara Ali

Book 2 subject : Telecom Billing Tutorial

Book 2 book_id : 6495700
```

Structures as Function Arguments

You can pass a structure as a function argument in very similar way as you pass any other variable or pointer. You would access structure variables in the similar way as you have accessed in the above example:

```c
#include <stdio.h>
#include <string.h>

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

/* function declaration */
void printBook( struct Books book );
int main( )
{
   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info */
   printBook( Book1 );

   /* Print Book2 info */
```

```
   printBook( Book2 );

   return 0;
}
void printBook( struct Books book )
{
   printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
   printf( "Book book_id : %d\n", book.book_id);
}
```

When the above code is compiled and executed, it produces the following result:

```
Book title : C Programming

Book author : Nuha Ali

Book subject : C Programming Tutorial

Book book_id : 6495407

Book title : Telecom Billing

Book author : Zara Ali

Book subject : Telecom Billing Tutorial

Book book_id : 6495700
```

Pointers to Structures

You can define pointers to structures in very similar way as you define pointer to any other variable as follows:

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows:

```
struct_pointer = &Book1;
```

To access the members of a structure using a **pointer** to that **structure**, you must use the **->** operator as follows:

```
struct_pointer->title;
```

Let us re-write above example using structure pointer, hope this will be easy for you to understand the concept:

```
#include <stdio.h>
```

```c
#include <string.h>

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

/* function declaration */
void printBook( struct Books *book );
int main( )
{
   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info by passing address of Book1 */
   printBook( &Book1 );

   /* print Book2 info by passing address of Book2 */
   printBook( &Book2 );

   return 0;
}
void printBook( struct Books *book )
{
   printf( "Book title : %s\n", book->title);
   printf( "Book author : %s\n", book->author);
   printf( "Book subject : %s\n", book->subject);
   printf( "Book book_id : %d\n", book->book_id);
}
```

When the above code is compiled and executed, it produces the following result:

```
Book title : C Programming

Book author : Nuha Ali

Book subject : C Programming Tutorial

Book book_id : 6495407

Book title : Telecom Billing

Book author : Zara Ali
```

```
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

C Unions

A union is a special data type available in C that enables you to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multi-purpose.

Defining a Union

To define a **union**, you must use the union statement in very similar was as you did while defining structure. The **union** statement defines a new data type, with more than one member for your program. The format of the **union statement** is as follows:

```
union [union tag]
{
   member definition;
   member definition;
   ...
   member definition;
} [one or more union variables];
```

The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final **semicolon**, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data which has the three members i, f, and str:

```
union Data
{
   int i;
   float f;
   char  str[20];
} data;
```

Now, a variable of Data type can store an integer, a floating-point number, or a string of characters. This means that a single variable i.e. same memory location can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in above example Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by character string. Following is the example which will display total memory size occupied by the above union:

```
#include <stdio.h>
#include <string.h>

union Data
{
   int i;
   float f;
   char  str[20];
};

int main( )
{
   union Data data;

   printf( "Memory size occupied by data : %d\n", sizeof(data));

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Memory size occupied by data : 20
```

Accessing Union Members

To access any member of a union, we use the member access operator **(.)**. The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use **union** keyword to define variables of union type. Following is the example to explain usage of union:

```
#include <stdio.h>
#include <string.h>

union Data
{
   int i;
   float f;
   char  str[20];
};

int main( )
{
   union Data data;

   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
```

```
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
data.i : 1917853763

data.f : 4122360580327794860452759994368.000000

data.str : C Programming
```

Here, we can see that values of **i** and f members of union got corrupted because final value assigned to the variable has occupied the memory location and this is the reason that the value if **str** member is getting printed very well. Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having union:

```
#include <stdio.h>
#include <string.h>

union Data
{
   int i;
   float f;
   char  str[20];
};

int main( )
{
   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);

   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
data.i : 10

data.f : 220.500000

data.str : C Programming
```

Here, all the members are getting printed very well because one member is being used at a time.

Searching

An important and recurring problem in computing is that of *locating information*. More succinctly, this problem is known as *searching*. This is a good topic to use for a preliminary exploration of the various issues involved in algorithm design.

1 Requirements for searching

Clearly, the information to be searched has to first be *represented* (or *encoded*) somehow. This is where *data structures* come in. Of course, in a computer, everything is ultimately represented as sequences of binary digits (bits), but this is too low level for most purposes. We need to develop and study useful data structures that are closer to the way humans think, or at least more structured than mere sequences of bits. This is because it is humans who have to develop and maintain the software systems – computers merely run them.

After we have chosen a suitable representation, the represented information has to be processed somehow. This is what leads to the need for *algorithms*. In this case, the process of interest is that of searching. In order to simplify matters, let us assume that we want to search a collection of integer numbers (though we could equally well deal with strings of characters, or any other data type of interest). To begin with, let us consider:

1. The most obvious and simple representation.
2. Two potential algorithms for processing with that representation.

As we have already noted, *arrays* are one of the simplest possible ways of representing collections of numbers (or strings, or whatever), so we shall use that to store the information to be searched. Later we shall look at more complex data structures that may make storing and searching more efficient.

Suppose, for example, that the set of integers we wish to search is {1,4,17,3,90,79,4,6,81}. We can write them in an array a as

$$a = [1, 4, 17, 3, 90, 79, 4, 6, 81]$$

If we ask where 17 is in this array, the answer is 2, the index of that element. If we ask where 91 is, the answer is *nowhere*. It is useful to be able to represent *nowhere* by a number that is not used as a possible index. Since we start our index counting from 0, any negative number would do. We shall follow the convention of using the number -1 to represent *nowhere*. Other (perhaps better) conventions are possible, but we will stick to this here.

2 Specification of the search problem

We can now formulate a *specification* of our search problem using that data structure:

> *Given an array a and integer x, find an integer i such that*
>
> 1. *if there is no j such that $a[j]$ is x, then i is -1,*
> 2. *otherwise, i is any j for which $a[j]$ is x.*

The first clause says that if x does not occur in the array a then i should be -1, and the second says that if it does occur then i should be a position where it occurs. If there is more than one position where x occurs, then this specification allows you to return any of them – for example, this would be the case if a were $[17, 13, 17]$ and x were 17. Thus, the specification is ambiguous. Hence different algorithms with different behaviours can satisfy the same specification – for example, one algorithm may return the smallest position at which x occurs, and another may return the largest. There is nothing wrong with ambiguous specifications. In fact, in practice, they occur quite often.

3 A simple algorithm: Linear Search

We can conveniently express the simplest possible *algorithm* in a form of *pseudocode* which reads like English, but resembles a computer program without some of the precision or detail that a computer usually requires:

```
// This assumes we are given an array a of size n and a key x.
For i = 0,1,...,n-1,
    if a[i] is equal to x,
        then we have a suitable i and can terminate returning i.
If we reach this point,
    then x is not in a and hence we must terminate returning -1.
```

Some aspects, such as the ellipsis "...", are potentially ambiguous, but we, as human beings, know exactly what is meant, so we do not need to worry about them. In a programming language such as *C* or *Java*, one would write something that is more precise like:

```
for ( i = 0 ; i < n ; i++ ) {
    if ( a[i] == x ) return i;
}
return -1;
```

In the case of *Java*, this would be within a method of a class, and more details are needed, such as the parameter a for the method and a declaration of the auxiliary variable i. In the case of *C*, this would be within a function, and similar missing details are needed. In either, there would need to be additional code to output the result in a suitable format.

In this case, it is easy to see that the algorithm satisfies the specification (assuming n is the correct size of the array) – we just have to observe that, because we start counting from zero, the last position of the array is its size minus one. If we forget this, and let i run from 0 to n instead, we get an incorrect algorithm. The practical effect of this mistake is that the execution of this algorithm gives rise to an error when the item to be located in the array is

actually not there, because a non-existing location is attempted to be accessed. Depending on the particular language, operating system and machine you are using, the actual effect of this error will be different. For example, in *C* running under Unix, you may get execution aborted followed by the message "segmentation fault", or you may be given the wrong answer as the output. In *Java*, you will always get an *error message*.

4 A more efficient algorithm: Binary Search

One always needs to consider whether it is possible to improve upon the performance of a particular algorithm, such as the one we have just created. In the worst case, searching an array of size n takes n steps. On average, it will take $n/2$ steps. For large collections of data, such as all web-pages on the internet, this will be unacceptable in practice. Thus, we should try to organize the collection in such a way that a more efficient algorithm is possible. As we shall see later, there are many possibilities, and the more we demand in terms of efficiency, the more complicated the data structures representing the collections tend to become. Here we shall consider one of the simplest – we still represent the collections by arrays, but now we enumerate the elements in ascending order. The problem of obtaining an ordered list from any given list is known as *sorting* and will be studied in detail in a later chapter.

Thus, instead of working with the previous array $[1, 4, 17, 3, 90, 79, 4, 6, 81]$, we would work with $[1, 3, 4, 4, 6, 17, 79, 81, 90]$, which has the same items but listed in ascending order. Then we can use an improved *algorithm*, which in English-like pseudocode form is:

```
// This assumes we are given a sorted array a of size n and a key x.
// Use integers left and right (initially set to 0 and n-1) and mid.
While left is less than right,
    set mid to the integer part of (left+right)/2, and
    if x is greater than a[mid],
        then       set left to mid+1,
        otherwise  set right to mid.
If a[left] is equal to x,
    then       terminate returning left,
    otherwise  terminate returning -1.
```

and would correspond to a segment of *C* or *Java* code like:

```
/* DATA */
int a = [1,3,4,4,6,17,79,81,90];
int n = 9;
int x = 79;
/* PROGRAM */
int left = 0, right = n-1, mid;
while ( left < right ) {
    mid = ( left + right ) / 2;
    if ( x > a[mid] ) left = mid+1;
    else right = mid;
}
if ( a[left] == x ) return left;
else return -1;
```

This algorithm works by repeatedly splitting the array into two segments, one going from $left$ to mid, and the other going from $mid + 1$ to $right$, where mid is the position half way from $left$ to $right$, and where, initially, $left$ and $right$ are the leftmost and rightmost positions of the array. Because the array is sorted, it is easy to see which of each pair of segments the searched-for item x is in, and the search can then be restricted to that segment. Moreover, because the size of the sub-array going from locations $left$ to $right$ is halved at each iteration of the while-loop, we only need $\log_2 n$ steps in either the average or worst case. To see that this runtime behaviour is a big improvement, in practice, over the earlier linear-search algorithm, notice that $\log_2 1000000$ is approximately 20, so that for an array of size 1000000 only 20 iterations are needed in the worst case of the binary-search algorithm, whereas 1000000 are needed in the worst case of the linear-search algorithm.

With the binary search algorithm, it is not so obvious that we have taken proper care of the boundary condition in the while loop. Also, strictly speaking, this algorithm is not correct because it does not work for the empty array (that has size zero), but that can easily be fixed. Apart from that, is it correct? Try to convince yourself that it is, and then try to explain your argument-for-correctness to a colleague. Having done that, try to *write down* some convincing arguments, maybe one that involves a *loop invariant* and one that doesn't. Most algorithm developers stop at the first stage, but experience shows that it is only when we attempt to write down seemingly convincing arguments that we actually find all the subtle mistakes. Moreover, it is not unusual to end up with a better/clearer algorithm after it has been modified to make its correctness easier to argue.

It is worth considering whether linked-list versions of our two algorithms would work, or offer any advantages. It is fairly clear that we could perform a linear search through a linked list in essentially the same way as with an array, with the relevant pointer returned rather than an index. Converting the binary search to linked list form is problematic, because there is no efficient way to split a linked list into two segments. It seems that our array-based approach is the best we can do with the data structures we have studied so far. However, we shall see later how more complex data structures (trees) can be used to formulate efficient recursive search algorithms.

Notice that we have not yet taken into account how much effort will be required to sort the array so that the binary search algorithm can work on it. Until we know that, we cannot be sure that using the binary search algorithm really is more efficient overall than using the linear search algorithm on the original unsorted array. That may also depend on further details, such as how many times we need to performa a search on the set of n items – just once, or as many as n times. We shall return to these issues later. First we need to consider in more detail how to compare algorithm efficiency in a reliable manner.

Efficiency and Complexity

We have already noted that, when developing algorithms, it is important to consider how *efficient* they are, so we can make informed choices about which are best to use in particular circumstances. So, before moving on to study increasingly complex data structures and algorithms, we first look in more detail at how to measure and describe their efficiency.

1 Time versus space complexity

When creating software for serious applications, there is usually a need to judge how quickly an algorithm or program can complete the given tasks. For example, if you are programming a flight booking system, it will not be considered acceptable if the travel agent and customer have to wait for half an hour for a transaction to complete. It certainly has to be ensured that the waiting time is reasonable for the size of the problem, and normally faster execution is better. We talk about the *time complexity* of the algorithm as an indicator of how the execution time depends on the *size* of the data structure.

Another important efficiency consideration is how much memory a given program will require for a particular task, though with modern computers this tends to be less of an issue than it used to be. Here we talk about the *space complexity* as how the memory requirement depends on the size of the data structure.

For a given task, there are often algorithms which trade time for space, and vice versa. For example, we will see that, as a data storage device, *hash tables* have a very good time complexity at the expense of using more memory than is needed by other algorithms. It is usually up to the algorithm/program designer to decide how best to balance the *trade-off* for the application they are designing.

2 Worst versus average complexity

Another thing that has to be decided when making efficiency considerations is whether it is the *average case* performance of an algorithm/program that is important, or whether it is more important to guarantee that even in the *worst case* the performance obeys certain rules. For many applications, the average case is more important, because saving time overall is usually more important than guaranteeing good behaviour in the worst case. However, for time-critical problems, such as keeping track of aeroplanes in certain sectors of air space, it may be totally unacceptable for the software to take too long if the worst case arises.

Again, algorithms/programs often trade-off efficiency of the average case against efficiency of the worst case. For example, the most efficient algorithm on average might have a particularly bad worst case efficiency. We will see particular examples of this when we consider efficient algorithms for sorting and searching.

3 Concrete measures for performance

These days, we are mostly interested in *time complexity*. For this, we first have to decide how to measure it. Something one might try to do is to just implement the algorithm and run it, and see how long it takes to run, but that approach has a number of problems. For one, if it is a big application and there are several potential algorithms, they would all have to be programmed first before they can be compared. So a considerable amount of time would be wasted on writing programs which will not get used in the final product. Also, the machine on which the program is run, or even the compiler used, might influence the running time. You would also have to make sure that the *data* with which you tested your program is typical for the application it is created for. Again, particularly with big applications, this is not really feasible. This empirical method has another disadvantage: it will not tell you anything useful about the next time you are considering a similar problem.

Therefore complexity is usually best measured in a different way. First, in order to not be bound to a particular programming language or machine architecture, it is better to measure the efficiency of the *algorithm* rather than that of its *implementation*. For this to be possible, however, the algorithm has to be described in a way which very much *looks* like the program to be implemented, which is why algorithms are usually best expressed in a form of *pseudocode* that comes close to the implementation language.

What we need to do to determine the time complexity of an algorithm is count the number of times each operation will occur, which will usually depend on the *size* of the problem. The size of a problem is typically expressed as an integer, and that is typically the number of items that are manipulated. For example, when describing a search algorithm, it is the number of items amongst which we are searching, and when describing a sorting algorithm, it is the number of items to be sorted. So the *complexity* of an algorithm will be given by a function which maps the number of items to the (usually approximate) number of time steps the algorithm will take when performed on that many items.

In the early days of computers, the various operations were each counted in proportion to their particular 'time cost', and added up, with multiplication of integers typically considered much more expensive than their addition. In today's world, where computers have become much faster, and often have dedicated floating-point hardware, the differences in time costs have become less important. However, we still we need to be careful when deciding to consider all operations as being equally costly – applying some function, for example, can take much longer than simply adding two numbers, and swaps generally take many times longer than comparisons. Just counting the most costly operations is often a good strategy.

4 Big-O notation for complexity class

Very often, we are not interested in the actual function $C(n)$ that describes the time complexity of an algorithm in terms of the problem size n, but just its *complexity class*. This ignores any constant overheads and small constant factors, and just tells us about the principal growth

of the complexity function with problem size, and hence something about the performance of the algorithm on large numbers of items.

If an algorithm is such that we may consider all steps equally costly, then usually the complexity class of the algorithm is simply determined by the number of loops and how often the content of those loops are being executed. The reason for this is that adding a constant number of instructions which does not change with the size of the problem has no significant effect on the overall complexity for large problems.

There is a standard notation, called the *Big-O notation*, for expressing the fact that constant factors and other insignificant details are being ignored. For example, we saw that the procedure `last(l)` on a list `l` had time complexity that depended linearly on the size n of the list, so we would say that the time complexity of that algorithm is $O(n)$. Similarly, linear search is $O(n)$. For binary search, however, the time complexity is $O(\log_2 n)$.

Before we define complexity classes in a more formal manner, it is worth trying to gain some intuition about what they actually mean. For this purpose, it is useful to choose one function as a representative of each of the classes we wish to consider. Recall that we are considering functions which map natural numbers (the size of the problem) to the set of non-negative real numbers $\mathbb{R}^+$, so the classes will correspond to common mathematical functions such as powers and logarithms. We shall consider later to what degree a representative can be considered 'typical' for its class.

The most common complexity classes (in increasing order) are the following:

- $O(1)$, pronounced 'Oh of one', or *constant* complexity;
- $O(\log_2 \log_2 n)$, 'Oh of log log en';
- $O(\log_2 n)$, 'Oh of log en', or *logarithmic* complexity;
- $O(n)$, 'Oh of en', or *linear* complexity;
- $O(n\log_2 n)$, 'Oh of en log en';
- $O(n^2)$, 'Oh of en squared', or *quadratic* complexity;
- $O(n^3)$, 'Oh of en cubed', or *cubic* complexity;
- $O(2^n)$, 'Oh of two to the en', or *exponential* complexity.

As a representative, we choose the function which gives the class its name – e.g. for $O(n)$ we choose the function $f(n) = n$, for $O(\log_2 n)$ we choose $f(n) = \log_2 n$, and so on. So assume we have algorithms with these functions describing their complexity. The following table lists how many operations it will take them to deal with a problem of a given size:

$f(n)$	$n = 4$	$n = 16$	$n = 256$	$n = 1024$	$n = 1048576$
1	1	1	1	1.00×10^0	1.00×10^0
$\log_2 \log_2 n$	1	2	3	3.32×10^0	4.32×10^0
$\log_2 n$	2	4	8	1.00×10^1	2.00×10^1
n	4	16	2.56×10^2	1.02×10^3	1.05×10^6
$n\log_2 n$	8	64	2.05×10^3	1.02×10^4	2.10×10^7
n^2	16	256	6.55×10^4	1.05×10^6	1.10×10^{12}
n^3	64	4096	1.68×10^7	1.07×10^9	1.15×10^{18}
2^n	16	65536	1.16×10^{77}	1.80×10^{308}	6.74×10^{315652}

Some of these numbers are so large that it is rather difficult to imagine just how long a time span they describe. Hence the following table gives time spans rather than instruction counts, based on the assumption that we have a computer which can operate at a speed of 1 MIP, where one MIP = a million instructions per second:

$f(n)$	$n = 4$	$n = 16$	$n = 256$	$n = 1024$	$n = 1048576$
1	1 μsec	1 μsec	1 μsec	1 μsec	1 μsec
$\log_2 \log_2 n$	1 μsec	2 μsec	3 μsec	3.32 μsec	4.32 μsec
$\log_2 n$	2 μsec	4 μsec	8 μsec	10 μsec	20 μsec
n	4 μsec	16 μsec	256 μsec	1.02 msec	1.05 sec
$n\log_2 n$	8 μsec	64 μsec	2.05 msec	1.02 msec	21 sec
n^2	16 μsec	256 μsec	65.5 msec	1.05 sec	1.8 wk
n^3	64 μsec	4.1 msec	16.8 sec	17.9 min	36,559 yr
2^n	16 μsec	65.5 msec	3.7×10^{63} yr	5.7×10^{294} yr	2.1×10^{315639} yr

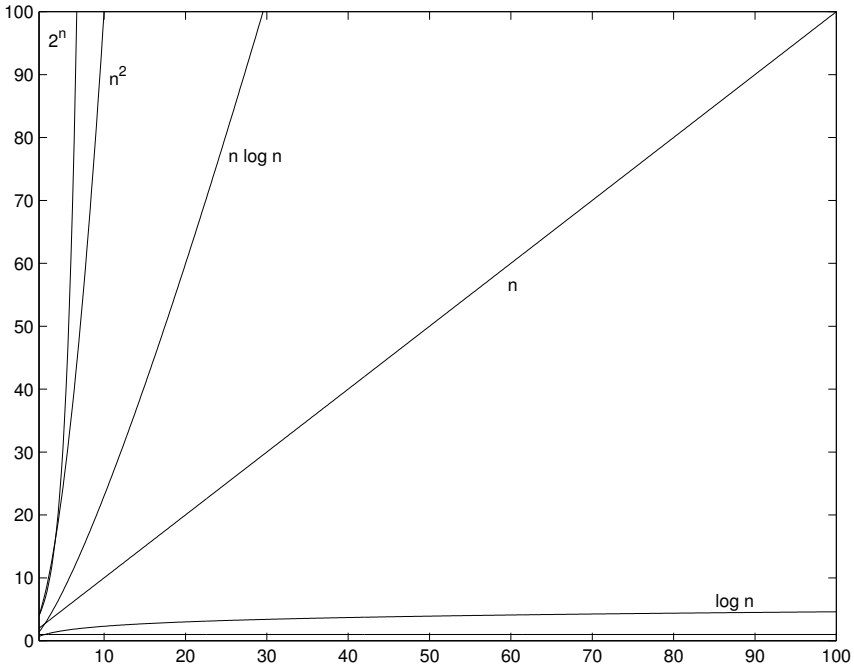
It is clear that, as the sizes of the problems get really big, there can be huge differences in the time it takes to run algorithms from different complexity classes. For algorithms with exponential complexity, $O(2^n)$, even modest sized problems have run times that are greater than the age of the universe (about 1.4×10^{10} yr), and current computers rarely run uninterrupted for more than a few years. This is why complexity classes are so important – they tell us how feasible it is likely to be to run a program with a particular large number of data items. Typically, people do not worry much about complexity for sizes below 10, or maybe 20, but the above numbers make it clear why it is worth thinking about complexity classes where bigger applications are concerned.

Another useful way of thinking about growth classes involves considering how the compute time will vary if the problem size doubles. The following table shows what happens for the various complexity classes:

$f(n)$	If the size of the problem doubles then $f(n)$ will be	
1	the same,	$f(2n) = f(n)$
$\log_2 \log_2 n$	almost the same,	$\log_2 (\log_2 (2n)) = \log_2 (\log_2 (n) + 1)$
$\log_2 n$	more by $1 = \log_2 2$,	$f(2n) = f(n) + 1$
n	twice as big as before,	$f(2n) = 2f(n)$
$n\log_2 n$	a bit more than twice as big as before,	$2n\log_2 (2n) = 2(n\log_2 n) + 2n$
n^2	four times as big as before,	$f(2n) = 4f(n)$
n^3	eight times as big as before,	$f(2n) = 8f(n)$
2^n	the square of what it was before,	$f(2n) = (f(n))^2$

This kind of information can be very useful in practice. We can test our program on a problem that is a half or quarter or one eighth of the full size, and have a good idea of how long we will have to wait for the full size problem to finish. Moreover, that estimate won't be affected by any constant factors ignored in computing the growth class, or the speed of the particular computer it is run on.

The following graph plots some of the complexity class functions from the table. Note that although these functions are only defined on natural numbers, they are drawn as though they were defined for all real numbers, because that makes it easier to take in the information presented.



It is clear from these plots why the non-principal growth terms can be safely ignored when computing algorithm complexity.

5 Formal definition of complexity classes

We have noted that complexity classes are concerned with *growth*, and the tables and graph above have provided an idea of what different behaviours mean when it comes to growth. There we have chosen a representative for each of the complexity classes considered, but we have not said anything about just how 'representative' such an element is. Let us now consider a more formal definition of a 'big O' class:

Definition. A function g belongs to the *complexity class* $O(f)$ if there is a number $n_0 \in \mathbb{N}$ and a constant $c > 0$ such that for all $n \geq n_0$, we have that $g(n) \leq c * f(n)$. We say that the function g is 'eventually smaller' than the function $c * f$.

It is not totally obvious what this implies. First, we do not need to know exactly *when* g becomes smaller than $c * f$. We are only interested in the *existence* of n_0 such that, from then on, g is smaller than $c * f$. Second, we wish to consider the efficiency of an algorithm independently of the speed of the computer that is going to execute it. This is why f is multiplied by a constant c. The idea is that when we measure the time of the steps of a particular algorithm, we are not sure how long each of them takes. By definition, $g \in O(f)$ means that eventually (namely beyond the point n_0), the growth of g will be at most as much as the growth of $c * f$. This definition also makes it clear that *constant factors* do not change the growth class (or O-class) of a function. Hence $C(n) = n^2$ is in the same growth class as $C(n) = 1/1000000 * n^2$ or $C(n) = 1000000 * n^2$. So we can write $O(n^2) = O(1000000 * n^2) = O(1/1000000 * n^2)$. Typically, however, we choose the *simplest* representative, as we did in the tables above. In this case it is $O(n^2)$.

The various classes we mentioned above are related as follows:

$$O(1) \subseteq O(\log_2 \log_2 n) \subseteq O(\log_2 (n)) \subseteq O(n) \subseteq O(n\log_2 n) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(2^n)$$

We only consider the principal growth class, so when adding functions from different growth classes, their sum will always be in the larger growth class. This allows us to simplify terms. For example, the growth class of $C(n) = 500000\log_2 n + 4n^2 + 0.3n + 100$ can be determined as follows. The summand with the largest growth class is $4n^2$ (we say that this is the 'principal sub-term' or 'dominating sub-term' of the function), and we are allowed to drop constant factors, so this function is in the class $O(n^2)$.

When we say that an algorithm 'belongs to' some class $O(f)$, we mean that it is *at most* as fast growing as f. We have seen that 'linear searching' (where one searches in a collection of data items which is unsorted) has linear complexity, i.e. it is in growth class $O(n)$. This holds for the *average case* as well as the *worst case*. The operations needed are comparisons of the item we are searching for with all the items appearing in the data collection. In the worst case, we have to check all n entries until we find the right one, which means we make n comparisons. On average, however, we will only have to check $n/2$ entries until we hit the correct one, leaving us with $n/2$ operations. Both those functions, $C(n) = n$ and $C(n) = n/2$ belong to the same complexity class, namely $O(n)$. However, it would be equally correct to say that the algorithm belongs to $O(n^2)$, since that class contains all of $O(n)$. But this would be *less informative*, and we would not say that an algorithm has quadratic complexity if we know that, in fact, it is linear. Sometimes it is difficult to be sure what the exact complexity is (as is the case with the famous NP = P problem), in which case one might say that an algorithm is 'at most', say, quadratic.

The issue of efficiency and complexity class, and their computation, will be a recurring feature throughout the chapters to come. We shall see that concentrating only on the complexity class, rather than finding exact complexity functions, can render the whole process of considering efficiency much easier. In most cases, we can determine the time complexity by a simple counting of the loops and tree heights. However, we will also see at least one case where that results in an overestimate, and a more exact computation is required.

Sorting

1 The problem of sorting

In computer science, 'sorting' usually refers to bringing a set of items into some well-defined order. To be able to do this, we first need to specify the notion of *order* on the items we are considering. For example, for numbers we can use the usual numerical order (that is, defined by the mathematical 'less than' or '$<$' relation) and for strings the so-called *lexicographic* or *alphabetic* order, which is the one dictionaries and encyclopedias use.

Usually, what is meant by *sorting* is that once the sorting process is finished, there is a simple way of 'visiting' all the items in order, for example to print out the contents of a database. This may well mean different things depending on how the data is being stored. For example, if all the objects are sorted and stored in an array a of size n, then

```
for i = 0,...,n-1
   print(a[i])
```

would print the items in ascending order. If the objects are stored in a linked list, we would expect that the first entry is the smallest, the next the second-smallest, and so on. Often, more complicated structures such as binary search trees or heap trees are used to sort the items, which can then be printed, or written into an array or linked list, as desired.

Sorting is important because having the items in order makes it much easier to *find* a given item, such as the cheapest item or the file corresponding to a particular student. It is thus closely related to the problem of *search*, as we saw with the discussion of binary search tress. If the sorting can be done beforehand (*off-line*), this enables faster *access* to the required item, which is important because that often has to be done on the fly (*on-line*). We have already seen that, by having the data items stored in a sorted array or binary search tree, we can reduce the average (and worst case) complexity of searching for a particular item to $O(\log_2 n)$ steps, whereas it would be $O(n)$ steps without sorting. So, if we often have to look up items, it is worth the effort to sort the whole collection first. Imagine using a dictionary or phone book in which the entries do not appear in some known logical order.

It follows that sorting algorithms are important tools for program designers. Different algorithms are suited to different situations, and we shall see that there is no 'best' sorting algorithm for everything, and therefore a number of them will be introduced in these notes. It is worth noting that we will be far from covering *all* existing sorting algorithms – in fact, the field is still very much alive, and new developments are taking place all the time. However,

the general strategies can now be considered to be well-understood, and most of the latest new algorithms tend to be derived by simply tweaking existing principles, although we still do not have accurate measures of performance for some sorting algorithms.

2 Common sorting strategies

One way of organizing the various sorting algorithms is by classifying the underlying idea, or 'strategy'. Some of the key strategies are:

enumeration sorting Consider all items. If we know that there are N items which are smaller than the one we are currently considering, then its final position will be at number $N + 1$.

exchange sorting If two items are found to be out of order, exchange them. Repeat till all items are in order.

selection sorting Find the smallest item, put it in the first position, find the smallest of the remaining items, put it in the second position ...

insertion sorting Take the items one at a time and insert them into an initially empty data structure such that the data structure continues to be sorted at each stage.

divide and conquer Recursively split the problem into smaller sub-problems till you just have single items that are trivial to sort. Then put the sorted 'parts' back together in a way that preserves the sorting.

All these strategies are based on *comparing* items and then rearranging them accordingly. These are known as comparison-based sorting algorithms. We will later consider other non-comparison-based algorithms which are possible when we have specific prior knowledge about the items that can occur, or restrictions on the range of items that can occur.

The ideas above are based on the assumption that all the items to be sorted will fit into the computer's internal memory, which is why they are often referred to as being *internal sorting algorithms*. If the whole set of items cannot be stored in the internal memory at one time, different techniques have to be used. These days, given the growing power and memory of computers, external storage is becoming much less commonly needed when sorting, so we will not consider *external sorting algorithms* in detail. Suffice to say, they generally work by splitting the set of items into subsets containing as many items as can be handled at one time, sorting each subset in turn, and then carefully merging the results.

3 How many comparisons must it take?

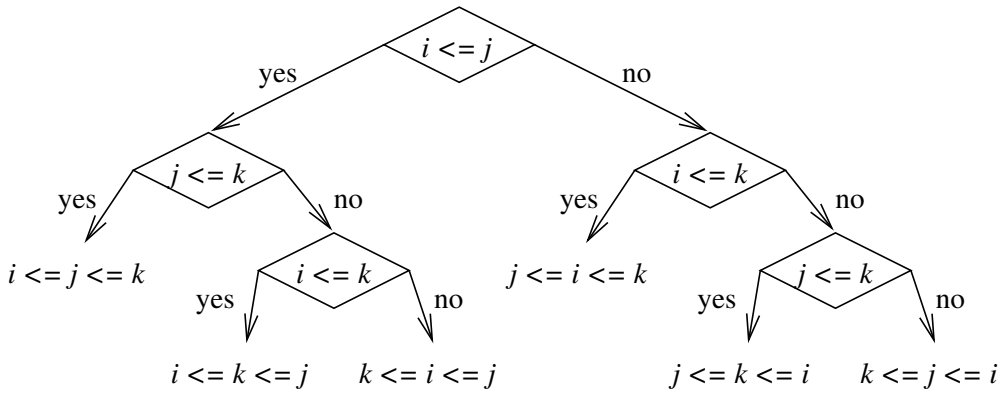
An obvious way to compute the *time complexity* of sorting algorithms is to count the number of comparisons they need to carry out, as a function of the number of items to be sorted. There is clearly no general *upper bound* on the number of comparisons used, since a particularly stupid algorithm might compare the same two items indefinitely. We are more interested in having a *lower bound* for the number of comparisons needed for the best algorithm in the worst case. In other words, we want to know the minimum number of comparisons required

to have all the information needed to sort an arbitrary collection of items. Then we can see how well particular sorting algorithms compare against that theoretical lower bound.

In general, questions of this kind are rather hard, because of the need to consider *all* possible algorithms. In fact, for some problems, optimal lower bounds are not yet known. One important example is the so-called *Travelling Salesman Problem* (TSP), for which all algorithms, which are known to give the correct shortest route solution, are extremely inefficient in the worst case (many to the extent of being useless in practice). In these cases, one generally has to relax the problem to find solutions which are *probably approximately correct*. For the TSP, it is still an open problem whether there exists a feasible algorithm that is guaranteed to give the exact shortest route.

For sorting algorithms based on comparisons, however, it turns out that a tight lower bound does exist. Clearly, even if the given collection of items is already sorted, we must still check all the items one at a time to see whether they are in the correct order. Thus, the lower bound must be at least n, the number of items to be sorted, since we need at least n steps to examine every element. If we already knew a sorting algorithm that works in n steps, then we could stop looking for a better algorithm: n would be both a lower bound and an upper bound to the minimum number of steps, and hence an *exact bound*. However, as we shall shortly see, no algorithm can actually take fewer than $O(n\log_2 n)$ comparisons in the worst case. If, in addition, we can design an algorithm that works in $O(n\log_2 n)$ steps, then we will have obtained an exact bound. We shall start by demonstrating that every algorithm needs at least $O(n\log_2 n)$ comparisons.

To begin with, let us assume that we only have three items, i, j, and k. If we have found that $i \leq j$ and $j \leq k$, then we know that the sorted order is: i, j, k. So it took us two comparisons to find this out. In some cases, however, it is clear that we will need as many as three comparisons. For example, if the first two comparisons tell us that $i > j$ and $j \leq k$, then we know that j is the smallest of the three items, but we cannot say from this information how i and k relate. A third comparison is needed. So what is the *average* and *worst* number of comparisons that are needed? This can best be determined from the so-called *decision tree*, where we keep track of the information gathered so far and count the number of comparisons needed. The decision tree for the three item example we were discussing is:



So what can we deduce from this about the general case? The decision tree will obviously always be a binary tree. It is also clear that its *height* will tell us how many comparisons will be needed in the worst case, and that the average length of a path from the root to a leaf will give us the average number of comparisons required. The leaves of the decision tree are

all the possible *outcomes*. These are given by the different possible orders we can have on n items, so we are asking how many ways there are of arranging n items. The first item can be any of the n items, the second can be any of the remaining $n-1$ items, and so forth, so their total number is $n(n-1)(n-2)\cdots 3\cdot 2\cdot 1 = n!$. Thus we want to know the height h of a binary tree that can accommodate as many as $n!$ leaves. The number of leaves of a tree of height h is at most 2^h, so we want to find h such that

$$2^h \geq n! \quad \text{or} \quad h \geq \log_2(n!)$$

There are numerous approximate expressions that have been derived for $\log_2(n!)$ for large n, but they all have the same dominant term, namely $n\log_2 n$. (Remember that, when talking about time complexity, we ignore any sub-dominant terns and constant factors.) Hence, no sorting algorithm based on comparing items can have a better average or worst case performance than using a number of comparisons that is approximately $n\log_2 n$ for large n. It remains to be seen whether this $O(n\log_2 n)$ complexity can actually be achieved in practice. To do this, we would have to exhibit at least one algorithm with this performance behaviour (and convince ourselves that it really does have this behaviour). In fact, we shall shortly see that there are several algorithms with this behaviour.

We shall proceed now by looking in turn at a number of sorting algorithms of increasing sophistication, that involve the various strategies listed above. The way they work depends on what kind of data structure contains the items we wish to sort. We start with approaches that work with simple arrays, and then move on to using more complex data structures that lead to more efficient algorithms.

4 Bubble Sort

Bubble Sort follows the *exchange sort* approach. It is very easy to implement, but tends to be particularly slow to run. Assume we have array `a` of size `n` that we wish to sort. Bubble Sort starts by comparing `a[n-1]` with `a[n-2]` and swaps them if they are in the wrong order. It then compares `a[n-2]` and `a[n-3]` and swaps those if need be, and so on. This means that once it reaches `a[0]`, the smallest entry will be in the correct place. It then starts from the back again, comparing pairs of 'neighbours', but leaving the zeroth entry alone (which is known to be correct). After it has reached the front again, the second-smallest entry will be in place. It keeps making 'passes' over the array until it is sorted. More generally, at the ith stage Bubble Sort compares neighbouring entries 'from the back', swapping them as needed. The item with the lowest index that is compared to its right neighbour is `a[i-1]`. After the ith stage, the entries `a[0],...,a[i-1]` are in their final position.

At this point it is worth introducing a simple 'test-case' of size `n = 4` to demonstrate how the various sorting algorithms work:

4	1	3	2

Bubble Sort starts by comparing `a[3]=2` with `a[2]=3`. Since they are not in order, it swaps them, giving | 4 | 1 | 2 | 3 |. It then compares `a[2]=2` with `a[1]=1`. Since those are in order, it leaves them where they are. Then it compares `a[1]=1` with `a[0]=4`, and those are not in order once again, so they have to be swapped. We get | 1 | 4 | 2 | 3 |. Note that the smallest entry has reached its final place. This will *always* happen after Bubble Sort has done its first 'pass' over the array.

Now that the algorithm has reached the zeroth entry, it starts at the back again, comparing `a[3]=3` with `a[2]=2`. These entries are in order, so nothing happens. (Note that these numbers have been compared before – there is nothing in Bubble Sort that prevents it from repeating comparisons, which is why it tends to be pretty slow!) Then it compares `a[2]=2` and `a[1]=4`. These are not in order, so they have to be swapped, giving | 1 | 2 | 4 | 3 |. Since we already know that `a[0]` contains the smallest item, we leave it alone, and the second pass is finished. Note that now the second-smallest entry is in place, too.

The algorithm now starts the third and final pass, comparing `a[3]=3` and `a[2]=4`. Again these are out of order and have to be swapped, giving | 1 | 2 | 3 | 4 |. Since it is known that `a[0]` and `a[1]` contain the correct items already, they are not touched. Furthermore, the third-smallest item is in place now, which means that the fourth-smallest *has to be* correct, too. Thus the whole array is sorted.

It is now clear that Bubble Sort can be implemented as follows:

```
for ( i = 1 ; i < n ; i++ )
   for ( j = n-1 ; j >= i ; j-- )
      if ( a[j] < a[j-1] )
         swap a[j] and a[j-1]
```

The outer loop goes over all `n` $-$ `1` positions that may still need to be swapped to the left, and the inner loop goes from the end of the array back to that position.

As is usual for comparison-based sorting algorithms, the time complexity will be measured by counting the number of comparisons that are being made. The outer loop is carried out $n-1$ times. The inner loop is carried out $(n-1)-(i-1) = n-i$ times. So the number of comparisons is the same in each case, namely

$$\begin{aligned}\sum_{i=1}^{n-1}\sum_{j=i}^{n-1} 1 &= \sum_{i=1}^{n-1}(n-i) \\ &= (n-1)+(n-2)+\cdots+1 \\ &= \frac{n(n-1)}{2}.\end{aligned}$$

Thus the worst case and average case number of comparisons are both proportional to n^2, and hence the average and worst case time complexities are $O(n^2)$.

5 Insertion Sort

Insertion Sort is (not surprisingly) a form of *insertion sorting*. It starts by treating the first entry `a[0]` as an already sorted array, then checks the second entry `a[1]` and compares it with the first. If they are in the wrong order, it swaps the two. That leaves `a[0],a[1]` sorted. Then it takes the third entry and positions it in the right place, leaving `a[0],a[1],a[2]` sorted, and so on. More generally, at the beginning of the ith stage, Insertion Sort has the entries `a[0],..., a[i-1]` sorted and inserts `a[i]`, giving sorted entries `a[0],...,a[i]`.

For the example starting array | 4 | 1 | 3 | 2 |, Insertion Sort starts by considering `a[0]=4` as sorted, then picks up `a[1]` and 'inserts it' into the already sorted array, increasing the size of it by 1. Since `a[1]=1` is smaller than `a[0]=4`, it has to be inserted in the zeroth slot,

but that slot is holding a value already. So we first move `a[0]` 'up' one slot into `a[1]` (care being taken to remember `a[1]` first!), and then we can move the old `a[1]` to `a[0]`, giving

1	4	3	2

.

At the next step, the algorithm treats `a[0],a[1]` as an already sorted array and tries to insert `a[2]=3`. This value obviously has to fit between `a[0]=1` and `a[1]=4`. This is achieved by moving `a[1]` 'up' one slot to `a[2]` (the value of which we assume we have remembered), allowing us to move the current value into `a[1]`, giving

1	3	4	2

.

Finally, `a[3]=2` has to be inserted into the sorted array `a[0],...,a[2]`. Since `a[2]=4` is bigger than 2, it is moved 'up' one slot, and the same happens for `a[1]=3`. Comparison with `a[0]=1` shows that `a[1]` was the slot we were looking for, giving

1	2	3	4

.

The general algorithm for Insertion Sort can therefore be written:

```
for ( i = 1 ; i < n ; i++ ) {
   for( j = i ; j > 0 ; j-- )
      if ( a[j] < a[j-1] )
         swap a[j] and a[j-1]
      else break
}
```

The outer loop goes over the `n - 1` items to be inserted, and the inner loop takes each next item and swaps it back through the currently sorted portion till it reaches its correct position. However, this typically involves swapping each next item many times to get it into its right position, so it is more efficient to store each next item in a temporary variable `t` and only insert it into its correct position when that has been found and its content moved:

```
for ( i = 1 ; i < n ; i++ ) {
   j = i
   t = a[j]
   while ( j > 0 && t < a[j-1] ) {
      a[j] = a[j-1]
      j--
   }
   a[j] = t
}
```

The outer loop again goes over `n-1` items, and the inner loop goes back through the currently sorted portion till it finds the correct position for the next item to be inserted.

The time complexity is again taken to be the number of comparisons performed. The outer loop is always carried out $n-1$ times. How many times the inner loop is carried out depends on the items being sorted. In the worst case, it will be carried out i times; on average, it will be half that often. Hence the number of comparison in the worst case is:

$$\begin{aligned}\sum_{i=1}^{n-1}\sum_{j=1}^{i} 1 &= \sum_{i=1}^{n-1} i \\ &= 1+2+\cdots+(n-1) \\ &= \frac{n(n-1)}{2};\end{aligned}$$

and in the average case it is half that, namely $n(n-1)/4$. Thus average and worst case number of steps for of Insertion Sort are both proportional to n^2, and hence the average and worst case time complexities are both $O(n^2)$.

6 Selection Sort

Selection Sort is (not surprisingly) a form of *selection sorting*. It first finds the smallest item and puts it into `a[0]` by exchanging it with whichever item is in that position already. Then it finds the second-smallest item and exchanges it with the item in `a[1]`. It continues this way until the whole array is sorted. More generally, at the ith stage, Selection Sort finds the ith-smallest item and swaps it with the item in `a[i-1]`. Obviously there is no need to check for the ith-smallest item in the first $i-1$ elements of the array.

For the example starting array | 4 | 1 | 3 | 2 |, Selection Sort first finds the smallest item in the whole array, which is `a[1]=1`, and swaps this value with that in `a[0]` giving | 1 | 4 | 3 | 2 |. Then, for the second step, it finds the smallest item in the reduced array `a[1],a[2],a[3]`, that is `a[3]=2`, and swaps that into `a[1]`, giving | 1 | 2 | 3 | 4 |. Finally, it finds the smallest of the reduced array `a[2],a[3]`, that is `a[2]=3`, and swaps that into `a[2]`, or recognizes that a swap is not needed, giving | 1 | 2 | 3 | 4 |.

The general algorithm for Selection Sort can be written:

```
for ( i = 0 ; i < n-1 ; i++ ) {
   k = i
   for ( j = i+1 ; j < n ; j++ )
      if ( a[j] < a[k] )
         k = j
   swap a[i] and a[k]
}
```

The outer loop goes over the first `n-1` positions to be filled, and the inner loop goes through the currently unsorted portion to find the next smallest item to fill the next position. Note that, unlike with Bubble Sort and Insertion Sort, there is exactly one swap for each iteration of the outer loop,

The time complexity is again the number of comparisons carried out. The outer loop is carried out $n-1$ times. In the inner loop, which is carried out $(n-1)-i = n-1-i$ times, one comparison occurs. Hence the total number of comparisons is:

$$\begin{aligned}\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 &= \sum_{i=0}^{n-2}(n-1-i)\\ &= (n-1)+\cdots+2+1\\ &= \frac{n(n-1)}{2}.\end{aligned}$$

Therefore the number of comparisons for Selection Sort is proportional to n^2, in the worst case as well as in the average case, and hence the average and worst case time complexities are both $O(n^2)$.

Note that Bubblesort, Insertion Sort and Selection Sort all involve two nested for loops over $O(n)$ items, so it is easy to see that their overall complexities will be $O(n^2)$ without having to compute the exact number of comparisons.

7 Comparison of O(n) sorting algorithms

We have now seen three different array based sorting algorithms, all based on different sorting strategies, and all with $O(n^2)$ time complexity. So one might imagine that it does not make much difference which of these algorithms is used. However, in practice, it can actually make a big difference which algorithm is chosen. The following table shows the measured running times of the three algorithms applied to arrays of integers of the size given in the top row:

Algorithm	128	256	512	1024	O1024	R1024	2048
Bubble Sort	54	221	881	3621	1285	5627	14497
Insertion Sort	15	69	276	1137	6	2200	4536
Selection Sort	12	45	164	634	643	833	2497

Here O1024 denotes an array with 1024 entries which are already sorted, and R1024 is an array which is sorted in the *reverse* order, that is, from biggest to smallest. All the other arrays were filled randomly. Warning: tables of measurements like this are *always* dependent on the random ordering used, the implementation of the programming language involved, and on the machine it was run on, and so will never be exactly the same.

So where exactly do these differences come from? For a start, Selection Sort always makes $n(n-1)/2$ comparisons, but carries out *at most* $n-1$ swaps. Each swap requires three assignments and takes, in fact, more time than a comparison. Bubble Sort, on the other hand, does a lot of swaps. Insertion Sort does particularly well on data which is sorted already – in such a case, it only makes $n-1$ comparisons. It is worth bearing this in mind for some applications, because if only a few entries are out of place, Insertion Sort can be very quick. These comparisons serve to show that complexity considerations can be rather delicate, and require good judgement concerning what operations to count. It is often a good idea to run some experiments to test the theoretical considerations and see whether any simplifications made are realistic in practice. For instance, we have assumed here that all comparisons cost the same, but that may not be true for big numbers or strings of characters.

What exactly to count when considering the complexity of a particular algorithm is always a judgement call. You will have to gain experience before you feel comfortable with making such decisions yourself. Furthermore, when you want to improve the performance of an algorithm, you may want to determine the biggest user of computing *resources* and focus on improving that. Something else to be aware of when making these calculations is that it is not a bad idea to keep track of any constant factors, in particular those that go with the dominating sub-term. In the above examples, the factor applied to the dominating sub-term, namely n^2, varies. It is $1/2$ for the average case of Bubble Sort and Selection Sort, but only $1/4$ for Insertion Sort. It is certainly useful to know that an algorithm that is linear will perform better than a quadratic one *provided the size of the problem is large enough*, but if you know that your problem has a size of, say, at most 100, then a complexity of $(1/20)n^2$ will be preferable to one of $20n$. Or if you know that your program is only ever used on fairly small samples, then using the simplest algorithm you can find might be beneficial overall – it is easier to program, and there is not a lot of compute time to be saved.

Finally, the above numbers give you some idea why, for program designers, the general rule is to *never* use Bubble Sort. It is certainly easy to program, but that is about all it has going for it. You are better off avoiding it altogether.

8 Sorting algorithm stability

One often wants to sort items which might have identical keys (e.g., ages in years) in such a way that items with identical keys are kept in their original order, particularly if the items have already been sorted according to a different criteria (e.g., alphabetical). So, if we denote the original order of an array of items by subscripts, we want the subscripts to end up in order for each set of items with identical keys. For example, if we start out with the array $[5_1, 4_2, 6_3, 5_4, 6_5, 7_6, 5_7, 2_8, 9_9]$, it should be sorted to $[2_8, 4_2, 5_1, 5_4, 5_7, 6_3, 6_5, 7_6, 9_9]$ and not to $[2_8, 4_2, 5_4, 5_1, 5_7, 6_3, 6_5, 7_6, 9_9]$. Sorting algorithms which satisfy this useful property are said to be *stable*.

The easiest way to determine whether a given algorithm is stable is to consider whether the algorithm can ever swap identical items past each other. In this way, the stability of the sorting algorithms studied so far can easily be established:

Bubble Sort This is stable because no item is swapped past another unless they are in the wrong order. So items with identical keys will have their original order preserved.

Insertion Sort This is stable because no item is swapped past another unless it has a smaller key. So items with identical keys will have their original order preserved.

Selection Sort This is not stable, because there is nothing to stop an item being swapped past another item that has an identical key. For example, the array $[2_1, 2_2, 1_3]$ would be sorted to $[1_3, 2_2, 2_1]$ which has items 2_2 and 2_1 in the wrong order.

The issue of sorting stability needs to be considered when developing more complex sorting algorithms. Often there are stable and non-stable versions of the algorithms, and one has to consider whether the extra cost of maintaining stability is worth the effort.

9 Treesort

Let us now consider a way of implementing an *insertion sorting* algorithm using a data structure better suited to the problem. The idea here, which we have already seen before, involves inserting the items to be sorted into an initially empty binary search tree. Then, when all items have been inserted, we know that we can traverse the binary search tree to visit all the items in the right order. This sorting algorithm is called *Treesort*, and for the basic version, we require that all the search keys be different.

Obviously, the tree must be kept balanced in order to minimize the number of comparisons, since that depends on the height of the tree. For a balanced tree that is $O(\log_2 n)$. If the tree is not kept balanced, it will be more than that, and potentially $O(n)$.

Treesort can be difficult to compare with other sorting algorithms, since it returns a tree, rather than an array, as the sorted data structure. It should be chosen if it is desirable to have the items stored in a binary search tree anyway. This is usually the case if items are frequently deleted or inserted, since a binary search tree allows these operations to be implemented efficiently, with time complexity $O(\log_2 n)$ per item. Moreover, as we have seen before, searching for items is also efficient, again with time complexity $O(\log_2 n)$.

Even if we have an array of items to start with, and want to finish with a sorted array, we can still use Treesort. However, to output the sorted items into the original array, we will need another procedure `fillArray(tree t, array a, int j)` to traverse the tree `t` and fill the array `a`. That is easiest done by passing and returning an index `j` that keeps track of the next array position to be filled. This results in the complete Treesort algorithm:

```
treeSort(array a) {
   t = EmptyTree
   for ( i = 0 ; i < size(a) ; i++ )
      t = insert(a[i],t)
   fillArray(t,a,0)
}

fillArray(tree t, array a, int j) {
   if ( not isEmpty(t) ) {
      j = fillArray(left(t),a,j)
      a[j++] = root(t)
      j = fillArray(right(t),a,j)
   }
   return j
}
```

which assumes that `a` is a pointer to the array location and that its elements can be accessed and updated given that and the relevant array index.

Since there are n items to insert into the tree, and each insertion has time complexity $O(\log_2 n)$, Treesort has an overall average time complexity of $O(n\log_2 n)$. So, we already have one algorithm that achieves the theoretical best average case time complexity of $O(n\log_2 n)$. Note, however, that if the tree is not kept balanced while the items are being inserted, and the items are already sorted, the height of the tree and number of comparisons per insertion will be $O(n)$, leading to a worst case time complexity of $O(n^2)$, which is no better than the simpler array-based algorithms we have already considered.

Exercise: We have assumed so far that the items stored in a Binary Search Tree must not contain any duplicates. Find the simplest ways to relax that restriction and determine how the choice of approach affects the *stability* of the associated Treesort algorithm.

10 Heapsort

We now consider another way of implementing a *selection sorting* algorithm using a more efficient data structure we have already studied. The underlying idea here is that it would help if we could pre-arrange the data so that selecting the smallest/biggest entry becomes easier. For that, remember the idea of a *priority queue* discussed earlier. We can take the value of each item to be its priority and then queue the items accordingly. Then, if we remove the item with the highest priority at each step we can fill an array in order 'from the rear', starting with the biggest item.

Priority queues can be implemented in a number of different ways, and we have already studied a straightforward implementation using *binary heap trees* in Chapter 8. However, there may be a better way, so it is worth considering the other possibilities.

An obvious way of implementing them would be using a sorted array, so that the entry with the highest priority appears in `a[n]`. Removing this item would be very simple, but inserting a new item would always involve finding the right position and shifting a number of items to the right to make room for it. For example, inserting a `3` into the queue $[1, 2, 4]$:

n	0	1	2	3	4	5
`a[n]`	1	2	4			

n	0	1	2	3	4	5
`a[n]`	1	2		4		

n	0	1	2	3	4	5
`a[n]`	1	2	3	4		

That kind of item insertion is effectively insertion sort and clearly inefficient in general, of $O(n)$ complexity rather than $O(\log_2 n)$ with a binary heap tree.

Another approach would be to use an unsorted array. In this case, a new item would be inserted by just putting it into `a[n+1]`, but to delete the entry with the highest priority would involve having to find it first. Then, after that, the last item would have to be swapped into the gap, or all items with a higher index 'shifted down'. Again, that kind of item deletion is clearly inefficient in general, of $O(n)$ complexity rather than $O(\log_2 n)$ with a heap tree.

Thus, of those three representations, only one is of use in carrying out the above idea efficiently. An unsorted array is what we started from, so that is not any help, and ordering the array is what we are trying to achieve, so heaps are the way forward.

To make use of binary heap trees, we first have to take the unsorted array and re-arrange it so that it satisfies the heap tree priority ordering. We have already studied the *heapify* algorithm which can do that with $O(n)$ time complexity. Then we need to extract the sorted array from it. In the heap tree, the item with the highest priority, that is the largest item, is always in `a[1]`. In the sorted array, it should be in the last position `a[n]`. If we simply swap the two, we will have that item at the right position of the array, and also have begun the standard procedure of removing the root of the heap-tree, since `a[n]` is precisely the item that would be moved into the root position at the next step. Since `a[n]` now contains the correct item, we will never have to look at it again. Instead, we just take the items `a[1],...,a[n-1]` and bring them back into a heap-tree form using the *bubble down* procedure on the new root, which we know to have complexity $O(\log_2 n)$.

Now the second largest item is in position `a[1]`, and its final position should be `a[n-1]`, so we now swap these two items. Then we rearrange `a[1],...,a[n-2]` back into a heap tree using the bubble down procedure on the new root. And so on.

When the ith step has been completed, the items `a[n-i+1],...,a[n]` will have the correct entries, and there will be a heap tree for the items `a[1],...,a[n-i]`. Note that the size, and therefore the height, of the heap tree decreases at each step. As a part of the ith step, we have to bubble down the new root. This will take at most twice as many comparisons as the height of the original heap tree, which is $\log_2 n$. So overall there are $n-1$ steps, with at most $2\log_2 n$ comparisons, totalling $2(n-1)\log_2 n$. The number of comparisons will actually be less than that, because the number of bubble down steps will usually be less than the full height of the tree, but usually not much less, so the time complexity is still $O(n\log_2 n)$.

The full Heapsort algorithm can thus be written in a very simple form, using the bubble down and heapify procedures we already have from Chapter 8. First `heapify` converts the

array into a binary heap tree, and then the for loop moves each successive root one item at a time into the correct position in the sorted array:

```
heapSort(array a, int n) {
   heapify(a,n)
   for( j = n ; j > 1 ; j-- ) {
      swap a[1] and a[j]
      bubbleDown(1,a,j-1)
   }
}
```

It is clear from the swap step that the order of identical items can easily be reversed, so there is no way to render the Heapsort algorithm *stable*.

The average and worst-case time complexities of the entire Heapsort algorithm are given by the *sum* of two complexity functions, first that of `heapify` rearranging the original unsorted array into a heap tree which is $O(n)$, and then that of making the sorted array out of the heap tree which is $O(n\log_2 n)$ coming from the $O(n)$ bubble-downs each of which has $O(\log_2 n)$ complexity. Thus the overall average and worst-case complexities are both $O(n\log_2 n)$, and we now have a sorting algorithm that achieves the theoretical best worst-case time complexity. Using more sophisticated priority queues, such as Binomial or Fibonacci heaps, cannot improve on this because they have the same delete time complexity.

A useful feature of Heapsort is that if only the largest $m \ll n$ items need to be found and sorted, rather than all n, the complexity of the second stage is only $O(m\log_2 n)$, which can easily be less than $O(n)$ and thus render the whole algorithm only $O(n)$.

11 Divide and conquer algorithms

All the sorting algorithms considered so far work on the whole set of items together. Instead, *divide and conquer* algorithms recursively split the sorting problem into more manageable sub-problems. The idea is that it will usually be easier to sort many smaller collections of items than one big one, and sorting single items is trivial. So we repeatedly split the given collection into two smaller parts until we reach the 'base case' of one-item collections, which require no effort to sort, and then merge them back together again. There are two main approaches for doing this:

Assuming we are working on an array `a` of size n with entries `a[0],...,a[n-1]`, then the obvious approach is to simply split the set of indices. That is, we split the array at item $n/2$ and consider the two sub-arrays `a[0],...,a[(n-1)/2]` and `a[(n+1)/2],...,a[n-1]`. This method has the advantage that the splitting of the collection into two collections of equal (or nearly equal) size at each stage is easy. However, the two sorted arrays that result from each split have to be *merged* together carefully to maintain the ordering. This is the underlying idea for a sorting algorithm called *mergesort*.

Another approach would be to split the array in such a way that, at each stage, all the items in the first collection are no bigger than all the items in the second collection. The splitting here is obviously more complex, but all we have to do to put the pieces back together again at each stage is to take the first sorted array followed by the second sorted array. This is the underlying idea for a sorting algorithm called *Quicksort*.

We shall now look in detail at how these two approaches work in practice.

12 Quicksort

The general idea here is to repeatedly split (or partition) the given array in such a way that all the items in the first sub-array are smaller than all the items in the second sub-array, and then concatenate all the sub-arrays to give the sorted full array.

How to partition. The important question is how to perform this kind of splitting most efficiently. If the array is very simple, for example [4,3,7,8,1,6], then a good split would be to put all the items smaller than 5 into one part, giving [4,3,1], and all the items bigger than or equal to 5 into the other, that is [7,8,6]. Indeed, moving all items with a smaller key than some given value into one sub-array, and all entries with a bigger or equal key into the other sub-array is the standard *Quicksort* strategy. The value that defines the split is called the *pivot*. However, it is not obvious what is the best way to choose the pivot value.

One situation that we absolutely have to avoid is splitting the array into an *empty* sub-array and the whole array again. If we do this, the algorithm will not just perform badly, it will not even terminate. However, if the pivot is chosen to be an item in the array, and the pivot is kept in between and separate from both sub-arrays, then the sub-arrays being sorted at each recursion will always be at least one item shorter than the previous array, and the algorithm is guaranteed to terminate.

Thus, it proves convenient to split the array at each stage into the sub-array of values smaller than or equal to some chosen pivot item, followed by that chosen pivot item, followed by the sub-array of values greater than or equal to the chosen pivot item. Moreover, to save space, we do not actually split the array into smaller arrays. Instead, we simply *rearrange* the whole array to reflect the splitting. We say that we *partition* the array, and the *Quicksort* algorithm is then applied to the sub-arrays of this partitioned array.

In order for the algorithm to be called recursively, to sort ever smaller parts of the original array, we need to tell it *which part* of the array is currently under consideration. Therefore, Quicksort is called giving the lowest index (`left`) and highest index (`right`) of the sub-array it must work on. Thus the algorithm takes the form:

```
quicksort(array a, int left, int right) {
   if ( left < right ) {
      pivotindex = partition(a,left,right)
      quicksort(a,left,pivotindex-1)
      quicksort(a,pivotindex+1,right)
   }
}
```

for which the initial call would be `quicksort(a,0,n-1)` and the array `a` at the end is sorted. The crucial part of this is clearly the `partition`(`a`, `left`, `right`) procedure that rearranges the array so that it can be split around an appropriate pivot `a`[`pivotindex`].

If we were to split off only one item at a time, Quicksort would have n recursive calls, where n is the number of items in the array. If, on the other hand, we halve the array at each stage, it would only need $\log_2 n$ recursive calls. This can be made clear by drawing a binary tree whose nodes are labelled by the sub-arrays that have been split off at each stage, and measuring its height. Ideally then, we would like to get two sub-arrays of roughly equal size (namely half of the given array) since that is the most efficient way of doing this. Of course, that depends on choosing a good pivot.

Choosing the pivot. If we get the pivot 'just right' (e.g., choosing 5 in the above example), then the split will be as even as possible. Unfortunately, there is no quick guaranteed way of finding the optimal pivot. If the keys are integers, one could take the average value of all the keys, but that requires visiting *all* the entries to sample their key, adding considerable overhead to the algorithm, and if the keys are more complicated, such as strings, you cannot do this at all. More importantly, it would not necessarily give a pivot that is a value in the array. Some sensible *heuristic* pivot choice strategies are:

- Use a random number generator to produce an index `k` and then use `a[k]`.
- Take a key from 'the middle' of the array, that is `a[(n-1)/2]`.
- Take a small sample (e.g., 3 or 5 items) and take the 'middle' key of those.

Note that one should *never* simply choose the first or last key in the array as the pivot, because if the array is almost sorted already, that will lead to the particularly bad choice mentioned above, and this situation is actually quite common in practice.

Since there are so many reasonable possibilities, and they are all fairly straightforward, we will not give a specific implementation for any of these pivot choosing strategies, but just assume that we have a `choosePivot(a,left,right)` procedure that returns the index of the pivot for a particular sub-array (rather than the pivot value itself).

The partitioning. In order to carry out the partitioning within the given array, some thought is required as to how this may be best achieved. This is more easily demonstrated by an example than put into words. For a change, we will consider an array of strings, namely the programming languages: [c, fortran, java, ada, pascal, basic, haskell, ocaml]. The ordering we choose is the standard lexicographic one, and let the chosen pivot be "fortran".

We will use markers | to denote a partition of the array. To the left of the left marker, there will be items we know to have a key smaller than or equal to the pivot. To the right of the right marker, there will be items we know to have a key bigger than or equal to the pivot. In the middle, there will be the items we have not yet considered. Note that this algorithm proceeds to investigate the items in the array *from two sides*.

We begin by swapping the pivot value to the end of the array where it can easily be kept separate from the sub-array creation process, so we have the array: [|c, ocaml, java, ada, pascal, basic, haskell | fortran]. Starting from the left, we find "c" is less than "fortran", so we move the left marker one step to the right to give [c | ocaml, java, ada, pascal, basic, haskell | fortran]. Now "ocaml" is greater than "fortran", so we stop on the left and proceed from the right instead, without moving the left marker. We then find "haskell" is bigger than "fortran", so we move the right marker to the left by one, giving [c | ocaml, java, ada, pascal, basic, | haskell, fortran]. Now "basic" is smaller than "fortran", so we have two keys, "ocaml" and "basic", which are 'on the wrong side'. We therefore swap them, which allows us to move both the left and the right marker one step further towards the middle. This brings us to [c, basic | java, ada, pascal | ocaml, haskell, fortran]. Now we proceed from the left once again, but "java" is bigger than "fortran", so we stop there and switch to the right. Then "pascal" is bigger than "fortran", so we move the right marker again. We then find "ada", which is smaller than the pivot, so we stop. We have now got [c, basic | java, ada, | pascal, ocaml, haskell, fortran]. As before, we want to swap "java" and "ada", which leaves the left and the right markers in the same place: [c, basic, ada, java | | pascal, ocaml, haskell, fortran], so we

stop. Finally, we swap the pivot back from the last position into the position immediately after the markers to give [c, basic, ada, java | | fortran, ocaml, haskell, pascal].

Since we obviously cannot have the marker indices 'between' array entries, we will assume the left marker is on the left of `a[leftmark]` and the right marker is to the right of `a[rightmark]`. The markers are therefore 'in the same place' once `rightmark` becomes smaller than `leftmark`, which is when we stop. If we assume that the keys are integers, we can write the partitioning procedure, that needs to return the final pivot position, as:

```
partition(array a, int left, int right) {
   pivotindex = choosePivot(a, left, right)
   pivot = a[pivotindex]
   swap a[pivotindex] and a[right]
   leftmark = left
   rightmark = right - 1
   while (leftmark <= rightmark) {
      while (leftmark <= rightmark  and  a[leftmark] <= pivot)
          leftmark++
      while (leftmark <= rightmark  and  a[rightmark] >= pivot)
          rightmark--
      if (leftmark < rightmark)
         swap a[leftmark++] and a[rightmark--]
   }
   swap a[leftmark] and a[right]
   return leftmark
}
```

This achieves a partitioning that ends with the same items in the array, but in a different order, with all items to the left of the returned pivot position smaller or equal to the pivot value, and all items to the right greater or equal to the pivot value.

Note that this algorithm doesn't require any extra memory – it just swaps the items in the original array. However, the swapping of items means the algorithm is not *stable*. To render quicksort stable, the partitioning must be done in such a way that the order of identical items can never be reversed. A conceptually simple approach that does this, but requires more memory and copying, is to simply go systematically through the whole array, re-filling the array `a` with items less than or equal to the pivot, and filling a second array `b` with items greater or equal to the pivot, and finally copying the array `b` into the end of `a`:

```
partition2(array a, int left, int right) {
   create new array b of size right-left+1
   pivotindex = choosePivot(a, left, right)
   pivot = a[pivotindex]
   acount = left
   bcount = 1
   for ( i = left ; i <= right ; i++ ) {
      if ( i == pivotindex )
         b[0] = a[i]
      else if ( a[i] < pivot || (a[i] == pivot && i < pivotindex) )
```

```
            a[acount++] = a[i]
         else
            b[bcount++] = a[i]
      }
      for ( i = 0 ; i < bcount ; i++ )
         a[acount++] = b[i]
      return right-bcount+1
   }
```

Like the first partition procedure, this also achieves a partitioning with the same items in the array, but in a different order, with all items to the left of the returned pivot position smaller or equal to the pivot value, and all items to the right greater or equal to the pivot value.

Complexity of Quicksort. Once again we shall determine complexity based on the number of comparisons performed. The partitioning step compares each of n items against the pivot, and therefore has complexity $O(n)$. Clearly, some partition and pivot choice algorithms are less efficient than others, like `partition2` involving more copying of items than `partition`, but that does not generally affect the overall complexity class.

In the *worst case*, when an array is partitioned, we have one empty sub-array. If this happens at each step, we apply the partitioning method to arrays of size n, then $n-1$, then $n-2$, until we reach 1. Those complexity functions then add up to

$$n + (n-1) + (n-2) + \cdots 2 + 1 = n(n+1)/2$$

Ignoring the constant factor and the non-dominant term $n/2$, this shows that, in the worst case, the number of comparisons performed by Quicksort is $O(n^2)$.

In the *best case*, whenever we partition the array, the resulting sub-arrays will differ in size by at most one. Then we have n comparisons in the first case, two lots of $\lfloor n/2 \rfloor$ comparisons for the two sub-arrays, four times $\lfloor n/4 \rfloor$, eight times $\lfloor n/8 \rfloor$, and so on, down to $2^{\log_2 n - 1}$ times $\lfloor n/2^{\log_2 n - 1} \rfloor = \lfloor 2 \rfloor$. That gives the total number of comparisons as

$$n + 2^1\lfloor n/2^1 \rfloor + 2^2\lfloor n/2^2 \rfloor + 2^3\lfloor n/2^3 \rfloor + \cdots + 2^{\log_2 n - 1}\lfloor n/2^{\log_2 n - 1} \rfloor \approx n\log_2 n$$

which matches the theoretical best possible time complexity of $O(n\log_2 n)$.

More interesting and important is how well Quicksort does in the *average case*. However, that is much harder to analyze exactly. The strategy for choosing a pivot at each stage affects that, though as long as it avoids the problems outlined above, that does not change the complexity class. It also makes a difference whether there can be duplicate values, but again that doesn't change the complexity class. In the end, *all* reasonable variations involve comparing $O(n)$ items against a pivot, for each of $O(\log_2 n)$ recursions, so the total number of comparisons, and hence the overall time complexity, in the average case is $O(n\log_2 n)$.

Like Heapsort, when only the largest $m \ll n$ items need to be found and sorted, rather than all n, Quicksort can be modified to result in reduced time complexity. In this case, only the first sub-array needs to be processed at each stage, until the sub-array sizes exceed m. In that situation, for the best case, the total number of comparisons is reduced to

$$n + 1\lfloor n/2^1 \rfloor + 1\lfloor n/2^2 \rfloor + 1\lfloor n/2^3 \rfloor + \cdots + m\log_2 m \approx 2n.$$

rendering the time complexity of the whole modified algorithm only $O(n)$. For the average case, the computation is again more difficult, but as long as the key problems outlined above are avoided, the average-case complexity of this special case is also $O(n)$.

Improving Quicksort. It is always worthwhile spending some time optimizing the strategy for defining the pivot, since the particular problem in question might well allow for a more refined approach. Generally, the pivot will be better if more items are sampled before it is being chosen. For example, one could check several randomly chosen items and take the 'middle' one of those, the so called *median*. Note that in order to find the median of all the items, without sorting them first, we would end up having to make n^2 comparisons, so we cannot do that without making Quicksort unattractively slow.

Quicksort is rarely the most suitable algorithm if the problem size is small. The reason for this is all the overheads from the recursion (e.g., storing all the return addresses and formal parameters). Hence once the sub-problem become 'small' (a size of 16 is often suggested in the literature), Quicksort should stop calling itself and instead sort the remaining sub-arrays using a simpler algorithm such as Selection Sort.

13 Mergesort

The other divide and conquer sorting strategy based on repeatedly splitting the array of items into two sub-arrays, mentioned in Section 9.11, is called *mergesort*. This simply splits the array at each stage into its first and last half, without any reordering of the items in it. However, that will obviously not result in a set of sorted sub-arrays that we can just append to each other at the end. So mergesort needs another procedure `merge` that merges two sorted sub-arrays into another sorted array. As with binary search in Section 4.4, integer variables `left` and `right` can be used to refer to the lower and upper index of the relevant array, and `mid` refers to the end of its left sub-array. Thus a suitable mergesort algorithm is:

```
mergesort(array a, int left, int right) {
   if ( left < right ) {
      mid = (left + right) / 2
      mergesort(a, left, mid)
      mergesort(a, mid+1, right)
      merge(a, left, mid, right)
   }
}
```

Note that it would be relatively simple to modify this mergesort algorithm to operate on linked lists (of known length) rather than arrays. To 'split' such a list into two, all one has to do is set the pointer of the $\lfloor n/2 \rfloor$th list entry to null, and use the previously-pointed-to next entry as the head of the new second list. Of course, care needs to be taken to keep the list size information intact, and effort is required to find the crucial pointer for each split.

The `merge` algorithm. The principle of merging two sorted collections (whether they be lists, arrays, or something else) is quite simple: Since they are sorted, it is clear that the smallest item overall must be either the smallest item in the first collection or the smallest item in the second collection. Let us assume it is the smallest key in the first collection. Now the second smallest item overall must be either the second-smallest item in the first collection, or the smallest item in the second collection, and so on. In other words, we just work through both collections and at each stage, the 'next' item is the current item in either the first or the second collection.

The implementation will be quite different, however, depending on which data structure we are using. When arrays are used, it is actually necessary for the `merge` algorithm to create a new array to hold the result of the operation at least temporarily. In contrast, when using linked lists, it would be possible for `merge` to work by just changing the reference to the next node. This does make for somewhat more confusing code, however.

For arrays, a suitable *merge* algorithm would start by creating a new array `b` to store the results, then repeatedly add the next smallest item into it until one sub-array is finished, then copy the remainder of the unfinished sub-array, and finally copy `b` back into `a`:

```
merge(array a, int left, int mid, int right) {
   create new array b of size right-left+1
   bcount = 0
   lcount = left
   rcount = mid+1
   while ( (lcount <= mid) and (rcount <= right) ) {
      if ( a[lcount] <= a[rcount] )
         b[bcount++] = a[lcount++]
      else
         b[bcount++] = a[rcount++]
   }
   if ( lcount > mid )
      while ( rcount <= right )
         b[bcount++] = a[rcount++]
   else
      while ( lcount <= mid )
         b[bcount++] = a[lcount++]
   for ( bcount = 0 ; bcount < right-left+1 ; bcount++ )
      a[left+bcount] = b[bcount]
}
```

It is instructive to compare this with the `partition2` algorithm for Quicksort to see exactly where the two sort algorithms differ. As with `partition2`, the merge algorithm never swaps identical items past each other, and the splitting does not change the ordering at all, so the whole Mergesort algorithm is *stable*.

Complexity of Mergesort. The total number of comparisons needed at each recursion level of mergesort is the number of items needing merging which is $O(n)$, and the number of recursions needed to get to the single item level is $O(\log_2 n)$, so the total number of comparisons and its time complexity are $O(n\log_2 n)$. This holds for the worst case as well as the average case. Like Quicksort, it is possible to speed up mergesort by abandoning the recursive algorithm when the sizes of the sub-collections become small. For arrays, 16 would once again be a suitable size to switch to an algorithm like Selection Sort.

Note that, with Mergesort, for the special case when only the largest/smallest $m \ll n$ items need to be found and sorted, rather than all n, there is no way to reduce the time complexity in the way it was possible with Heapsort and Quicksort. This is because the ordering of the required items only emerges at the very last stage after the large majority of the comparisons have already been carried out.

14 Summary of comparison-based sorting algorithms

The following table summarizes the key properties of all the comparison-based sorting algorithms we have considered:

Sorting Algorithm	Strategy employed	Objects manipulated	Worst case complexity	Average case complexity	Stable
Bubble Sort	Exchange	arrays	$O(n^2)$	$O(n^2)$	Yes
Selection Sort	Selection	arrays	$O(n^2)$	$O(n^2)$	No
Insertion Sort	Insertion	arrays/lists	$O(n^2)$	$O(n^2)$	Yes
Treesort	Insertion	trees/lists	$O(n^2)$	$O(n\log_2 n)$	Yes
Heapsort	Selection	arrays	$O(n\log_2 n)$	$O(n\log_2 n)$	No
Quicksort	D & C	arrays	$O(n^2)$	$O(n\log_2 n)$	Maybe
Mergesort	D & C	arrays/lists	$O(n\log_2 n)$	$O(n\log_2 n)$	Yes

To see what the time complexities mean in practice, the following table compares the typical run times of those of the above algorithms that operate directly on arrays:

Algorithm	128	256	512	1024	O1024	R1024	2048
Bubble Sort	54	221	881	3621	1285	5627	14497
Selection Sort	12	45	164	634	643	833	2497
Insertion Sort	15	69	276	1137	6	2200	4536
Heapsort	21	45	103	236	215	249	527
Quicksort	12	27	55	112	1131	1200	230
Quicksort2	6	12	24	57	1115	1191	134
Mergesort	18	36	88	188	166	170	409
Mergesort2	6	22	48	112	94	93	254

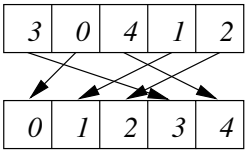
As before, arrays of the stated sizes are filled randomly, except O1024 that denotes an array with 1024 entries which are already sorted, and R1024 that denotes an array which is sorted in the *reverse* order. Quicksort2 and Mergesort2 are algorithms where the recursive procedure is abandoned in favour of Selection Sort once the size of the array falls to 16 or below. It should be emphasized again that these numbers are of limited accuracy, since they vary somewhat depending on machine and language implementation.

What has to be stressed here is that there is no 'best sorting algorithm' in general, but that there are usually good and bad choices of sorting algorithms *for particular circumstances*. It is up to the program designer to make sure that an appropriate one is picked, depending on the properties of the data to be sorted, how it is best stored, whether all the sorted items are required rather than some sub-set, and so on.

15 Non-comparison-based sorts

All the above sorting algorithms have been based on comparisons of the items to be sorted, and we have seen that we can't get time complexity better than $O(n\log_2 n)$ with comparison based algorithms. However, in some circumstances it is possible to do better than that with sorting algorithms that are not based on comparisons.

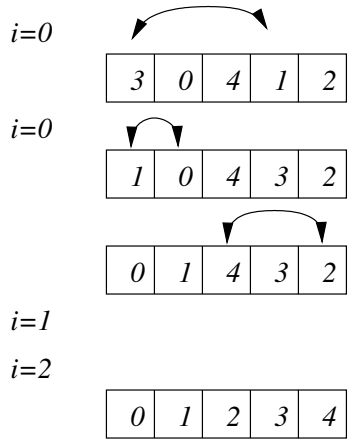
It is always worth thinking about the data that needs to be sorted, and whether comparisons really are required. For example, suppose you know the items to be sorted are the numbers from 0 to $n-1$. How would you sort those? The answer is surprisingly simple. We know that we have n entries in the array and we know *exactly which items should go there and in which order.* This is a very unusual situation as far as general sorting is concerned, yet this kind of thing often comes up in every-day life. For example, when a hotel needs to sort the room keys for its 100 rooms. Rather than employing one of the comparison-based sorting algorithms, in this situation we can do something much simpler. We can simply put the items directly in the appropriate places, using an algorithm such as that as shown in Figure 9.1:




```
create array b of size n
for ( i = 0 ; i < n ; i++ )
    b[a[i]] = a[i]
copy array b into array a
```

Figure 9.1: Simply put the items in the right order using their values.

This algorithm uses a second array `b` to hold the results, which is clearly not very memory efficient, but it is possible to do without that. One can use a series of swaps within array `a` to get the items in the right positions as shown in Figure 9.2:




```
for ( i = 0 ; i < n; i++ ) {
   while ( a[i] != i )
      swap a[a[i]] and a[i]
}
```

Figure 9.2: Swapping the items into the right order without using a new array.

As far as time complexity is concerned, it is obviously not appropriate here to count the number of comparisons. Instead, it is the number of swaps or copies that is important. The algorithm of Figure 9.1 performs n copies to fill array `b` and then another n to return the result to array a, so the overall time complexity is $O(n)$. The time complexity of the algorithm of Figure 9.2 looks worse than it really is. This algorithm performs at most $n-1$ swaps, since one item, namely `a[a[i]]` is always swapped into its final position. So at worst, this has time complexity $O(n)$ too.

This example should make it clear that in particular situations, sorting might be performed by much simpler (and quicker) means than the standard comparison sorts, though most realistic situations will not be quite as simple as the case here. Once again, it is the responsibility of the program designer to take this possibility into account.

16 Bin, Bucket, Radix Sorts

Bin, Bucket, and Radix Sorts are all names for essentially the same non-comparison-based sorting algorithm that works well when the items are labelled by small sets of values. For example, suppose you are given a number of dates, by day and month, and need to sort them into order. One way of doing this would be to create a queue for each day, and place the items (dates) one at a time into the right queue according to their day (without sorting them further). Then form one big queue out of these, by concatenating all the day queues starting with day 1 and continuing up to day 31. Then for the second phase, create a queue for each month, and place the dates into the right queues *in the order that they appear in the queue created by the first phase*. Again form a big queue by concatenating these month queues in order. This final queue is sorted in the intended order.

This may seem surprising at first sight, so let us consider a simple example:

[25/12, 28/08, 29/05, 01/05, 24/04, 03/01, 04/01, 25/04, 26/12, 26/04, 05/01, 20/04].

We first create and fill queues for the days as follows:

01: [01/05]
03: [03/01]
04: [04/01]
05: [05/01]
20: [20/04]
24: [24/04]
25: [25/12, 25/04]
26: [26/12, 26/04]
28: [28/08]
29: [29/05]

The empty queues are not shown – there is no need to create queues before we hit an item that belongs to them. Then concatenation of the queues gives:

[01/05, 03/01, 04/01, 05/01, 20/04, 24/04, 25/12, 25/04, 26/12, 26/04,28/08, 29/05].

Next we create and fill queues for the months that are present, giving:

01: [03/01, 04/01, 05/01]
04: [20/04, 24/04, 25/04, 26/04]
05: [01/05, 29/05]
08: [28/08]
12: [25/12, 26/12]

Finally, concatenating all these queues gives the items in the required order:

[03/01, 04/01, 05/01, 20/04, 24/04, 25/04, 26/04, 01/05, 29/05, 28/08, 25/12, 26/12].

This is called *Two-phase Radix Sorting*, since there are clearly two phases to it.

The extension of this idea to give a general sorting algorithm should be obvious: For each phase, create an ordered set of queues corresponding to the possible values, then add each item in the order they appear to the end of the relevant queue, and finally concatenate the the queues in order. Repeat this process for each sorting criterion. The crucial additional detail is that the queuing phases must be performed in the order of the significance of each criteria, with the *least significant* criteria first.

For example, if you know that your items to be sorted are all (at most) two-digit integers, you can use Radix Sort to sort them. First create and fill queues for the last digit, concatenate, then create and fill queues for the first digit, and concatenate to leave the items in sorted order. Similarly, if you know that your keys are all strings consisting of three characters, you can again apply Radix Sort. You would first queue according to the third character, then the second, and finally the first, giving a *Three phase* Radix Sort.

Note that *at no point*, does the the algorithm actually *compare* any items at all. This kind of algorithm makes use of the fact that for each phase the items are *from a strictly restricted set*, or, in other words, the items are of a particular form which is known *a priori*. The complexity class of this algorithm is $O(n)$, since at every phase, each item is dealt with precisely once, and the number of phases is assumed to be small and constant. If the *restricted sets* are small, the number of operations involved in finding the right queue for each item and placing it at the end of it will be small, but this could become significant if the sets are large. The concatenation of the queues will involve some overheads, of course, but these will be small if the sets are small and linked lists, rather than arrays, are used. One has to be careful, however, because if the total number of operations for each item exceeds $\log_2 n$, then the overall complexity is likely to be greater than the $O(n\log_2 n)$ complexity of the more efficient comparison-based algorithms. Also, if the *restricted sets* are not known in advance, and potentially large, the overheads of finding and sorting them could render Radix sort worse than using a comparison-based approach. Once again, it is the responsibility of the program designer to decide whether a given problem can be solved more efficiently with Radix Sort rather than a comparison-based sort.