DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CS13002 ; Programming and Data Structures
Semester II 2004-05
Brief Lecture Notes Instructor **Sandeep Sen**

1 Lecture 1, Jan 3, 2005

1.1 The notion of computing

In our everyday life, we are involved in numerous activities that require numerical calculation, starting from our grocery bills, keeping cricket scores to handling budget. It is not unusual to find shopkeepers punching in numbers to produce bills for customers. In bigger stores, life is made simpler where there are check-out machines where one has to only punch in the item codes (rather than the item prices) for producing the bills, thus reducing the chances of error in punching the prices. This also provides additional flexibility to the shopowners who can modify the prices of items very easily by changing the "table of prices." The person at the check-out counter doesn't have to remember the prices and he/she can punch the same keys for the same items. But this requires for sophisticated computing machines than calculators.

1.2 An abstract model of computer

A machine that executes instructions faithfully and very quickly. There are differences between computers with regards to speed and instructions but the *final answer* will be same. The computers are all more or less equally capable in their ability to solve problems and *possibly there is nothing better* (Church-Turing thesis).

1.3 What is a programming language

A formal language that contains a set of instructions that will produce the same (modulo some numerical precision) answers even if executed in diferent machines. Therefore, we need not bother about the internals of a computer when we write a program.

All programming languages are again equally capable of solving problems, namely that they have rich enough instruction set. (It is actually a very small set by which we can carry out all computations.)

1.4 Can you compute everything

A very deep philosophical question that was answered in the negative by a German Mathematical called Goedel, who *proved* that one cannot compute everything. This was done in setting of proving theorems mechanically in an axiomatic system. Fortunately not many naturally occuring problems fall in the non-computatble category.

Another paradox is that all this was done even before the first real computer was built. So the notion of computation had existed much before computers were built.

2 Lecture 2, Jan 5, 2005

2.1 What is interesting computation

A fixed sequence of instruction, for example, how to prepare cake is not particularly interesting since it has nothing unpredictable about it. Or say, how to go from RP hall to Bhatnagar is equally boring once someone has figured it out. However, given a campus map, how do we get from point A to point B is a more challenging exercise.

If the number of source-destinations is fixed then again you can do a *one-time* computation and store the routes. Just look up the table (as you do in case of a railway time table).

The problem becomes more interesting, if you are not told in advance which map you will be searching, i.e. the map is also now part of the input.

2.2 An algorithm

Before you write a program, which is ready to be fed into a computer, we must *reason out* a strategy to solve the problem. The strategy should be

- Finite (we can't have infinite instructions)
- Correct for **all possible inputs**
- Terminate (in finite time, infact the sooner the better)

The last two properties must be formally argued/proved before we translate it into a program. In fact the more time we spend on this, the chances of a successful program is higher. Any strategy that satisfies the above properties is called an *Algorithm* for a given problem (and there ican be more than one). With experience, the process of translating an algorithm into a program becomes more mechanical. One may ask now, how do we describe an algorithm ? It is usually done using less formal means although one can claim that a program itself describes an algorithm. There are many messy details that are usually left out in describing an algorithm.

2.3 Goodness of a program

Usually one argues at the level of the algorithm regarding use of resources like execution time and memory consumption. Scaling is often used to avoid absolute measurements in terms of a specific computer.

How many elementary operations do we do to multiply two 100 digit numbers ? How much paper do we consume ?

3 Lectures 3, 4 – Jan 10,11 2005

3.1 Designing a program

As mentioned in the previous lecture, the process of designing a program begins with development of an algorithm for the problem. It is a process of successive refinement starting with a rough sketch and filling in more details gradually till you have a program. For instance, in order to find solution of simultaneous linear equations, we first think in terms of which of the mathematical methods we want to pursue. Since there are some well-known methods (like say Gaussian elimination) [1], we focus on how to implement the the main steps of this strategy, namely the pivot step (eliminating a variable). Therefore the first sketch of the algorithm can be

```
Read input
Pivot $x_1$ , $x_2$ ..
Solve the triangular system
Print results
```

Of these the first and last are very standard routines that are part of most programs and doesn't require much thought, so we can ignore them now. The next step is to fill in details about how to implement steps 2 and 3 using some standard instruction set. Also we have to worry about how do we keep track of the coefficients of the equations ? We will come back to this example later after we look at some simpler problems.

3.2 Finding the median of 3 numbers

The problem is as follows - given 3 numbers x, y, z, we want to identify the second largest number breaking ties arbitrarily. In general if you are given $2n + 1$ numbers, we want to find out out the $n + 1$-th largest number. Among 6, 1, 9 we have 6 as the median. Although 3 is a small number of variables that we can solve by inspection, it is a good starting point for the understanding some of the basics of algorithm design.

Essentially if x is median it must satisfy $z < x < y$ or $y < x < z$. For y and z to be median we have symmetric cases. (I claim that distinctness can be assumed - how ?). This immediately gives us a strategy to solve this problem. We must figure out which of the above (six) situations holds.

To write a program we clearly require some instruction like "If <condition< then <carry out some instructions> otherwise <carry out alternate instructions>". This is a fundamental construct in all programming languages although the form (syntax) may be different. A condition is a *Boolean* expression that we will discuss soon. The following is a C program for the above algorithm.

[1] We could have alternatively chosen a method based on computing determinant

```
#include <stdio.h>

int i, j, k;

main ()
{
  printf("Supply three integers \n");
  scanf("%d %d %d",&i , &j, &k);

  if (( i < j) && (j < k)) printf("%d\n", j) ;
  if (( k < j) && (j < i)) printf("%d\n", j) ;

  if (( j < k) && (k < i)) printf("%d\n", k) ;
  if (( i < k) && (k < j)) printf("%d\n", k) ;

  if (( j < i) && (i < k)) printf("%d\n", i) ;
  if (( k < i) && (i < j)) printf("%d\n", i) ;


}
```

Here is another variation - which one will you prefer and why ?

```
#include <stdio.h>
int i, j, k;
main ()
{
  printf("Supply three integers \n");
  scanf("%d %d %d",&i , &j, &k);
  if (i < j )
         { if (j <k)
               {printf("%d\n",j);}
               else { if (i < k) /* i < j and k < j */
                        {printf("%d\n",i);} /* i < j and k < j and i < k*/
                        else {printf("%d\n",k);}
                    }
          }
    else { if (i <k)
               {printf("%d\n",i);}
               else { if (j < k)
                        {printf("%d\n",j);}
                        else {printf("%d\n",k);}
                    }
```

```
        }
}
```

When we write nested if ..else statements, it is a good practice to write the conditions that are explicitly satisfies at any stage of the code (as indicated in the comments alongside the second printf statement). This enables us to argue about correctness and debug a program if we are not getting desired results.

3.3 The notion of a variable

The word variable in the context of programming language should be confused with the usual notion of a mathematical variable (which are used to express some relationship in equations). Altough like mathematical variables, the programming language variable also has a domain of values, something $x = x + 1$ makes perfect sense ! A variable is a symbolic name for a memory location and contains a value from a predeclared domain. This value can be accessed (read operation) or modified (write operation or called an assignment operation).
The value of a variable remains unchanged unless it is assigned some value explicitly
For example, if $x = y + z$; $y = 10$; the value of x is not affected by the second statement, i.e. don't think of the assignment operator "=" as a mathematical equation between x, y and z.

The notation &x refers to the (numerical) memory location of x. Notice that the instruction scanf uses the address of the variable it is reading. One may guess that &x + 1 is the next address and it is intuitively correct modulo some technical details and one must be very careful with using numerical addresses. More discussion on this later.

Most programming anguages including C requires that we declare the variable before using it, which essentially is declaring the type (domain of the values it can take). The predefined types in C are int (integers in a limited range), floats (a subset of real numbers in a limited range) and char (set of characters in English language). These types can be prefixed by **long , short, unsigned**.

It is extremely important that types are not mixed although C is not a very strictly typed language. Variabes of one type can be converted to another by using some explicit instructions (typecasting). The important point is *limited range* of the each type (in particular integers and reals).This is a consequence of the fact that a memory location can hold a maximum of some k bits for k typically 32, 64, 128.

3.4 Expressions

An expression evaluates to some value of a certain type (primarily integer or float). In addition we have *Conditional/Boolean* expressions that evaluate to either True or False [2]. The

[2] In C there is no explicit Boolean type and True/False are represented by non-zero and zero values

integer/float expressions are called arithmetic expressions and one can compose arithmetic expressions with arithmetic operators like addition, multiplication and using other mathematical functions. The basic arithmetic expressions are constants and variables (C allows even assignment instructions).

C contains many predefined mathematical functions in a library called math.h that you can include in your program (and subsequently compile with -lm option to link the code).

The basic Boolean constants are True (non-zero) and False (zero). We can compose boolean expressions by using boolean operators like *AND* (`&&` in C) and *OR* (`||` in C). If X and Y are two boolean expressions then X`&&` Y is True iff both X, Y are True. On the other hand, X `||`Y is False iff both X, Y are False. Another useful operator is the unary operator `!` which when applied to an expression reverses the truth value.
Question: How many boolean functions are there with 2 inputs ?
Caution It is a good idea to use parentheses liberally when you are writing any expression to eliinate ambiguities,otherwise you must consult the precedence rules of C. For example `5 + 6 * 7` may evaluate to (5+6)*7 or 5+(6*7) depending on the precedence rules.

4 Lecture 5, Iteration

4.1 Finding the k-th Fibonacci number

The Fibonacci sequence consists of numbers $0, 1, 1, 2, 3, 5, 8, 11, \ldots$. We want to write a program that given some integer $k \geq 1$ prints out the k-th member, F_k, of this sequence. Now unlike some sequence like odd-number sequence there is no easy formula for F_k. The obvious alternative is to compute all the F_i's for $i \leq k$ starting from F_1. The input to the program is an integer and so is the output. It is extremely important that we understand these type of input and output - it is not so obvious in many cases as we shall see in future.

Starting with $F_1 = 0$ and $F_2 = 1$, we successively compute $F_3 = F_1 + F_2$, $F_4 = F_3 + F_2$ etc. **until** we reach F_k.

We have our second *control* instruction, viz., we are doing something repetitive until some *condition* is satisfied. Informally it is like after we compute F_i, we must check if $i = k$ before we compute F_{i+1}.

Describing this strategy like

> Is $k = 1$? then print F_1 otherwise
> Is $k = 2$? then print F_2 otherwise compute F_3
> Is $k = 3$ then print F_3 otherwise compute F_4
> ... ..

appears completely comprehensible - however it does not satisfy the finiteness property of a program as k is not known in advance. Moreover we do not even have a finite set of variables F_i. Here is where we exploit the property that only the last two terms are required to compute F_i which can be stored in some memory location. They must be updated suitably as we compute successive F_i's.

Remark Fibonacci sequence is also defined using the following simple inductive relation - $F_{i+2} = F_{i+1} + F_i$ What happens if you literally expand this definition ?
You may also want to prove that $F_{i+2} = 1 + \sum_{j-=1}^{i} F_j$.

There is also a closed form formula that given, k, computes F_k. You can write the formula as an expression to compute F_k. Find out more about the formula and you may want to compare the results of the two programs.

```
#include <stdio.h>
int i, j, k, prev, cur;

main ()
{
  printf("Which k ? \n");
  scanf("%d",&k);

    if (k == 1) printf("%d\n", 0);
    if (k == 2) printf("%d\n", 1);
```

```
   prev = 0; cur = 1;
    i = 2 ;
  while (i< k) /* k > 2 */
    {

     j = cur ;
    cur = cur + prev ; /* cur = F(i+1) */
     prev = j ;        /* prev = F(i) */
     i   = i+1 ;        /* cur = F(i), prev = F(i-1) */
    }              /* i equals k after we exit the loop */
   if (k > 2) printf("The %d Fibonacci number is %d\n", k, cur);

}
```

Inductive reasoning While filling out the details of a while loop, we often make mistakes regarding the exact condition that will give us the right results (i.e. in the above program replacing i $<$ k with i$\leq k$ will not give the right output). Rather than trying out things by trial and error which is not sound once you are dealing with nested loops, it is best to write out a an *induction hypothesis* for the i-th execution of the loop. We have written out these values in comments for the above program. With more experience, it may not be necessary to write these *loop invariants* for each instruction of a loop but for debugging purposes it is an extremely useful method.

You may want to try the following problem to appreciate the utility of such *invariants* - don't write a program but solve it theoretically.

> Given a jar of blue and red marbles, repeat the following process till there is only one marble left in the jar.
> Randomly pick two marbles - if their colours are same replace with a red marble (assume you have sufficient supply of red marbles), otherwise replace with a blue marble.
> Can you characterize the colour of the last marble in terms of the initial number of red and blue marbles ?

4.2 Other iteration constructs

C offers alternative constructs like Do .. while and For statements that sometimes offer more convenience. However all these instructions can be substituted by the other. In the beginning it is advisable to use only one of these, say the While construct. Once you master this, you can move to the other ones depending on the situation.

5 Jan 17,18 Writing functions and introduction to arrays

Functions map elements from a domain to a range and is defined for every element in the domain. C provides many packages of predefined functions called *libraries*, for example the Mathematical library called math.h. Nevertheless there are many siuations that we come across where we feel the necessity for additional functions. Being able to define our own functions within the context of a program is one of the most powerful features of a high level programming language. It allows us to write more compact and readable programs. (Imagine what would have happened if C didn't have a multiplication operator and you had to write it in terms of addition every time you needed it!) Some programming language (like LISP, ML, Smalltalk) are almost exclusively designed around viewing programs as functions and have extremely powerful features for defining functions. These languages are called *functional languages.*

Like a mathematical function, a programming language function when supplied with parameters (from the domain) returns a value from the range. The *body* of a function is a program that computes the value of the function when it is invoked with some specific parameter values (called actual parameters). The following is a C implementation of the function $f(n) = n \cdot (n-1) \cdot (n-2)..1$ if $n \geq 1$ and equal to 1 if $n = 0$ which is the factorial function.

```
#include <stdio.h>

 int factorial (int t) ; /* the name of the function is declaredin advance */
main ()
{
  int n, ans, k;
  printf("Supply n (+ve integer) and k (+ve integer) for n choose k \n");
  scanf("%d%d",&n,&k);
  ans = factorial(n)/(factorial((n-k))*factorial(k)) ;
  printf("%d\n", ans);
 }
int factorial (int n )  /*factorial fnction */
  {
   int ans ;
    if (n == 0) return 1;
     else
    {
     ans = n; /* initialization*/
     while (n > 1)
    {
        n = n-1;
       ans = ans*n ;
    }
    return ans;
  }
  }
```

To compute ${}^nC_k$, we need to compute factorial of n, k and $(n-k)$, so it makes perfect sense to write a function to compute factorial.

6 Arrays

When a program takes in a large number of similar (type) inputs (say exceeding 50), it become difficult for the programmer to keep track of these input variables and also to find different names for them. It may be easier to name them as $a_0, a_1 \cdot a_{49}$, where a_i is a variable. This is an example of a one dimensional *array* where an array is characterized by a name (it was **a** in the previous example) and the subscripts are called indexes that have integral value ≥ 1 upto a *prespecified range.* The declaration `int a[10]` in C defines a block of 10 consecutive memory locations called a[0] a[1] .. a[9] that can hold type int.

```
#include <stdio.h>

#define length 5
main ()  /*prints the distinct elements in an array */
{
 int i, j, a[length], more;
  printf("Supply %d integers in one line separated by spaces\n"length);
  for (i =0 ; i <= (length -1); i = i+1)
  {
  scanf("%d",&a[i]);
   }
     i = 0 ;
     while (i <= (length-2)) /*if an element a[j] for j > i */
                   /* equals a[i], don't print a[i] */
    {
        j = i+1 ; more = 0; /* more is a boolean variable to exit the */
                  /*loop prematurely */
     while ((j <= length) && !(more))
         {
          if ( a[i] == a[j]) more = 1;
           j = j+1;
         }
       if (!(more)) printf("%d ", a[i]);
       i = i+ 1;
    }
        printf("%d", a[i]) ; /* the last element will always be printed*/
}
```

Exercise Rewrite the above program by substituting the inner loop with a function.

7 Recursive functions

Many mathematical functions that do not have otherwise closed forms often have succint inductive definitions. For example,

$$factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot factorial(n-1) & \text{otherwise} \end{cases}$$

This definition actually forms the basis of computing the factorial function by iterative product that we have seen earlier. It required keeping track of the partial products and a loop control variable which is somewhat far-removed from the basic definition that it is derived from. The good news is that we can actually rewrite the function in C as

```
 int fact ( int n)

 {
   if ( n == 0) return 1;
    else return (n* fact(n-1));
 }
```

That is, we are allowed to make self referential call within the function which is one of the most powerful features of function calls. Such functions are often called *recursive* functions and often simplify programming by allowing us to write compact and readable codes. Moreover, we can argue about the correctness of the program using Mathematical induction very naturally.

Recursive programs may sometimes have some hidden pitfalls like the one presented below that computes Fibonacci numbers from its recursive definition.

```
#include <stdio.h>
 int fib (int n) ;
main ()
{
  int n;
  printf("Which n ? \n");
  scanf("%d",&n);

   printf("The %d Fibonacci number is %d\n", n, fib(n));
 }

 int fib (int n)

 {
    if (n == 1) {return 0;}
      else
        { if ( n == 2) {return 1;}
```

```
           else { return (fib (n-1) + fib(n-2)); }
        }

}
```

A somewhat different implementation of the same recursive definition is given below. Which one will you prefer and why ?

```
#include <stdio.h>
int tailfib (int cur, int prev, int i, int n );
main ()
{
  int n, ans ;
  printf("Which n ? \n");
  scanf("%d",&n);

   if ( n==1)  ans = 0;
      else
          { if ( n == 2) ans = 1;
             else ans = tailfib(1,1, 3, n);
          }
   printf("The %d Fibonacci number is %d\n", n, ans);
 }

 int tailfib (int cur, int prev, int i, int n )

 {
    if (i == n) return cur;
      else return tailfib( cur+prev, cur, i+1, n) ;

}
```

8 Jan 24, 25 Computer Organization and number representation

Discussion on major components of a computer system (like CPU, Memory, peripherals). Discussion on Memory hierarchy, Memory Management, Role of Operating system. Some discussion on compilers (machine dependent), assemblers, machine language/assembly language. Example of a source code being converted to assembly and running in the computer with interactions between registers, memory, ALU and control unit (animation available under Prof Abhijit Das's notes).

Internal representation of data types like integers, floats and character and comparing the choice of data types for a situation. Why is 2's complement better than a simple signed representation.

9 Jan 27, Higher dimensional arrays

For many applications, a two dimensional array is a natural representaion like a matrix. A two dimensional array is declared as `int a[10][20]` where a is a 10×20 array, 10 being the number of rows and 20 the number of columns. An element of the array is addressed as `a[i][j]` rather than `a[i,j]`. In other words, a two dimensional array is an array of arrays.

The following program reads the elements of a matrix in a *row-major* format (elements of row i is input before elements of row $i+1$) and prints out the matrix also in a row-major form.

```
#include <stdio.h>

 #define row 2
 #define col 2

  void readmatrix  (int a[][], int r, int c) ;
  void printmatrix (int a[][], int r , int c) ;

main ()  /*reads and prints elements of a row by col matrix */
{
 int a[row][col], b[row][col] , c[row][col],i, j ;
  printf("Supply %d times %d integers in a row major format\n",row, col);

     readmatrix(  a,  row, col );
     printmatrix(a , row, col) ;
 }
void readmatrix (int a[][col], int r , int c)

 {
   int i , j ;
  for (i = 0 ; i <= (r -1); i = i+1)
  {
      for ( j =0 ; j <= (c-1) ;j = j+1)

         scanf("%d",&a[i][j]);
   }
 }

void printmatrix (int a[][col], int r, int c)

 {
   int i , j ;

  for (i =0 ; i <= (r -1); i = i+1)
  {
       for ( j =0 ; j <= (c-1) ;j = j+1)

       {
         printf("%d ",a[i][j]);
       }
         printf("\n");
   }

  }
```

There are few things that are to be noted in this program.

- The type of the function readmatrix and printmatrix is "void". This implies that the functions do not return any value explicitly. The entire matrix is read (printed) within the function so there is no need for the function to return anything since the job is accomplished within the function. In some programming languages such functions are called *procedures*.
- Study the way the matrix is passed as a parameter to the functions from the main program (i.e. without the index square braces). We will discuss more when we study pointers in details.
- If a were a one dimensional array then writing `a[]` would have sufficed in formal parameter list of the functions.

10 Jan 31, Feb 1 Efficiency of programs

The running times of different programs (corresponding to the same problem) may vary significantly depending on the underlying algorithm. Of course, the same program will take different time to execute depending on the computer where it is running. So rather than comparing the real-time of execution, we will try to compare the number of instructions that are executed by the different programs. Here again we assume that each instruction takes unit time. [3]

The number of instructions executed will depend on the input or more precisely the *input-size.* The larger the input, the more the number of instructions executed. For example, for computing Fibonacci numbers, more instructions are executed to compute the 50th Fibonacci number than the 10th Fibonacci number. In fact, we would like to express the number of instructions to compute the k-th Fibonacci number as a function of k. So, the efficiency of a program can be expressed in terms of a function that represents the number of instructions executed in terms of the input-size.

This function is often referred to as time complexity

10.1 How to compare functions

If we have two programs have time complexities $3n + 6$ and $2n^2 + 8$ respectively then it is clear that we will prefer the first program. Here n is the input size. Suppose they have time complexities $9n + 20$ and $n^2 + 1$, then the choice may not be clear since the second program is superior initially but for larger n then the first program is better. In such situations, we will choose the program that *asymptotically* (eventually) becomes better. That is for some n_o, for all $n \geq n_o$ the program executes fewer instructions than others.

Therefore, it is meaningful only to look at the leading term (with the largest exponent) and we also ignore the multiplicative constants. In other words $3n^2 + 6n + 8$ will be approximated by n^2 - the notation used is $O(n^2)$ (read big-O of n^2).

10.2 Analysing programs

Let us try to come up with a function that captures the number of instructions in the program in Section 6 that prints out the distinct elements of an array. The primary contributors in any program are the loops, since they are executed many times - the remaining instructions are executed once and may be ignored since we are neglecting additive constants in the time complexity.

In this program, we have a nested loop, where the outer loop increments the variable `i` from 0 to `length -2` and in the inner loop the variable `j` is incremented from `i+1` to `length`. The input size of this program is clearly the size of the array which we have defined as `length`, so we will calculate the number of instructions as a function of `length`. To count the instructions, we write them in form of summations of instructions within a loop where the outer summation

[3]This is somewhat of an oversimplification as multiplication takes longer than addition - but still serves our purpose to get estimates of the running times

is for the outer loop and the inner summation is for the inner loop. By setting $n =$ `length`, we obtain

$$\sum_{i=0}^{i=(n-2)} 2 + \left(\sum_{j=i+1}^{n} 2 \right) + 2$$

which is $4(n-2) + 2\sum_{i=0}^{n-2}(n-i)$. This is approximately $2n^2 + 4n$.

Repeat the same exercise for the Fibonacci number program in section 4.1.

10.3 Efficiency of recursive functions

You may want to skip this part till you are comfortable with writing recursive functions. Since induction forms the basis of recursive functions, we express the number of instructions as a inductive relation often called *recurrence relations*. We then find solutions (which are functions) that satisfy this relation (analogous to finding roots of simple equations). Finding solutions to recurrence relations is often a very difficult exercise.

Consider the Fibonacci program in Section 7. The number of instructions for Fib(n) is the the summation of the number of instructions for Fib(n-1) and Fib(n-2) plus three more instructions (corresponding to two conditionals and one addition). Therefore we can express the recurrence relation for $T(n) =$ number of instructions for computing Fib(n) as

$$T(n) = T(n-1) + T(n-2) + 3$$

What function of n will satisfy this relation ?

However the other recursive version has a simpler structure since the function makes exactly one recursive call with `i` incremented by 1. Therefore we can express $T(i,n) = T(i+1,n) + 3$ and $T(n,n) = 1$ which yields $T(i,n) = 3(n-i)$ and in particular $T(3,n) = 3(n-3)$ since the initial recursive call is with $i = 3$.

11 Sorting : insertion sort (Feb 7,10)

The problem of arranging numbers in a non-decreasing or non-increasing order is considered to be one of the most fundamental problems in computer science and good programs for sorting have many applications. One of the natural methods for sorting is insertion sort which can described as follows. The n numbers are ordered as $a_1, a_2 \ldots a_n$ according to their initial input ordering. At any stage i ($i < n$), we have a sorted sequence of the the numbers $a_1, a_2 \ldots a_i$ - at stage 1, a_1 is a singleton sorted sequence. We now want to increase the length of the sorted sequence to include a_{i+1}. This amounts to finding the *correct* position for a_{i+1} in the sorted sequence $b_1, b_2 \ldots b_i$ where $b_j \leq b_{j+1}$. IN other words, we want to find b_k, b_{k+1} such that $b_k \leq a_{i+1} \leq b_{k+1}$. Once we locate this we must create *space* for a_{i+1} by moving all elements to the right of b_k by one position. The algorithm is complete when we have considered a_n. For phase i we may require $2i$ steps, in the most pessimistic scenario as we may have to compare a_{i+1} elements with all the elements to its left to find its correct position and another i steps to move the i elements one step to their right. This amounts to $\sum_{i=1}^{n} 2i = O(n^2)$ steps time complexity.

Is it possible to improve it ? At this stage we will only focus on the job of finding the correct location. Given a *sorted* [4] sequence $b_1, b_2 \ldots b_n$, and a search key a, how quickly can we locate $b_j \leq a \leq b_{j+1}$?

We can make the following observation - If we compare a with any arbitrary element b_k of the sequence, there are three possible scenarios-

- a equals b_k - then we have found b_j.
- $a > b_k$ - b_j is greater than b_k and hence we can eliminate $b_1, b_2 \ldots b_k$ from further considerations.
- $a < b_k$ - b_j is smaller than b_k and we can eliminate $b_{k+1}, b_{k+2} \ldots b_n$ from further considerations.

With a little thought, you can figure out that the best strategy to minimize the number of comparisons with a is to choose $k = n/2$ - then we can eliminate at least half the elements. This leads to $\log_2 n$ comparisons in the worst case to find b_j.

Using this result in insertion sort we can then bound the total number of comparisons to locate a_{i+1} over all stages is $\sum_{i=1}^{n} \lceil \log_2 i \rceil$ which is less than $n \log_2 n$. This is substantially less than n^2, so if we can reduce the data-movement part for creating space, then we have hope of obtaining a better result. We will return to this problem.

Given below is a recursive implementation of binary search.

```
#include <stdio.h>
#define length 8

  void bsearch (int a[] , int l, int u, int key , int *pos)
```

[4]This problem makes sense only when the elements are sorted.

```
 {
   int mid ;
    if (  l == u) { if ( a[l] == key ) *pos =  l ; else {
                                         *pos = -1 ; printf("not found\n");}
                 }
       else {
              mid = ((int) (l+u)/2) ;
              if ( a[mid] == key) { *pos = mid ;}
                 else { if  (a[mid] < key )
                          bsearch(a, mid+1, u, key, pos);
                                        /* key is larger than a[mid]*/
                        else bsearch(a, l, mid, key, pos);


                     }

      }


 }

  main()

 { int pos;
   int  a[length] = { 2, 4, 5, 8, 22, 31, 31, 60} ;
   bsearch (a, 0, 7, 2, &pos);
  if ( pos != -1) printf("found at %d\n", pos);

}
```

11.1 Discussion on pointers, dereferencing and parameter passing

If we want to manipulate the values of variables by making function calls, this is not supported directly since the parameters are passed as values to the function. All the manipulations that happen within the function body do not affect the values of the calling parameters. For example if we want to excahnge the values of two variables x,y using a function call `swap (int x , int y)` where the function is described as

```
 void swap ( int a ,  int b)
{ int t;
 t = a ; a = b ; b = t;
 }
```

ends up swapping values of a and b and not x and y. For this purpose, we modify the parameter passing in terms of the addresses of x and y (also called pointers to x and y).

```
 void swap ( int *a ,  int *b)
{ int t;
 t = *a ; *a = *b ; *b = t;

 }
```

The "*" operator is called the dereferencing or indirection operator and in some sense, the dual of the `&` operator. Given an address (pointer) a, `*a` refers to the contents of the memory location whose address is a. Clearly `*(&x)` is the same as x. Note that the *type* of a pointer is determined by what basic type it is pointing to, so `*a` can be int, char, float, double etc.

Another useful application is when we want to return more than one value from the function. We can pass the address of a variable(s) that stores the results. The variable `pos` in the function bsearch is used for this purpose.

12 Tower of Hanoi - fun with recursion (Feb 8)

This famous problem can be stated as follows. Given n disks of different diameters, we initially stack them in a way such that no larger disk rests on a smaller disk - call this pile A. We have to now move the pile to another pile B by

- we can't have more than three piles at any time call these A, B, C.
- exactly one disk can be moved from one pile to another.
- in each of the three piles we can never have a larger disk rest on a smaller disk.

Without a third pile, we cannot accomplish this task - just consider two disks. Given three piles, an easy solution is to move the smallest disk (let us denote these disks by 1, 2, ...n) from A to C. Move disk 2 from A to B and then move disk 1 from C to B. When we have 3 disks, we can move disks 1,2 to pile C by a sequence of moves where we interchange the roles of B and C. We then move disk 3 from A to B. Then again we move disks 1,2 by a sequence of moves from C to B (the roles of A and C are interchanged). Note that disk 3 never resides on top of disks 1 or 2.

In general, we devise an inductive strategy by first moving disks 1,2, n-1, from A to C (by a sequence of moves inductively), followed by moving disk n from A to B and then moving 1,2, n-1 from C to B (inductively). It is simple to prove by induction that this strategy works.

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

```
void towerhanoi(int n, int from, int to , int using) ;

 main()

 {
   int n ;
   printf("What is the number of discs \n");
   scanf ("%d", &n);
   towerhanoi(n , 1 ,2 ,3);
 }

 void towerhanoi (int n, int from, int to, int using )

 {
   if ( n == 1) {printf( "move disc from %d to %d \n", from, to);}
       else { towerhanoi( n-1 , from, using, to) ;
               printf( "move disk from %d to %d \n", from, to);
               towerhanoi( n-1 , using, to, from) ;


            }

 }
```

The number of moves required to shift n disks should satisfy the recurrence relation $T(n) = T(n-1) + 1 + T(n-1)$ and $T(1) = 1$. Here T(n) is the function that captures the number of moves required for n disks. You can verify that $T(n) = 2^n - 1$ satisfies this relation.
Problem Can you generalize the Tower of Hanoi to $m > 3$ piles and see if you can reduce the number of moves for n disks ?

13 Games between two players, Feb 28

Most of us have played tic-tac-toe long enough to know that when each layer plays to the best of their abilities, the game ends in a draw. There is a well-defined strategy (algorithm) such that if we follow it the outcome is either a win or a draw. Therefore we should be able to write a program to play tic-tac-toe. The basic idea is that at any stage of the game called a *state*, a player X (X is A or B), makes a move such that the other player cannot *force* a win.

So, for every possible move, player X must see if the other player can force a win - if so he must avoid that move. If there is no such move then X has already lost the game (assuming the other player plays it right). On the other hand if there is some move for X such that the irrespective of what the other player plays X can force a win then X is in a winning position.

Therefore we can categorize every state by { W, L, D } corresponding to Winning, Losing, or Draw from X's perspective [5]. Note that if it is Winning from X's perspective it is Losing from the other player's perspective. It is this nice symmetry that makes it easier to write a program that is naturally recursive by merely switching the roles of A and B and Win and Loss. The following algorithm is a general approach for the two player games. Let the function WinA(state) return 1 for player A if there is a winning move for A and 0 otherwise. Equivalently it will return 0 if there is a winning move for B and 1 otherwise.

```
WinA (state S)
If the state S is a base case then return 0 or 1 as the situation may be else
    if for some move i of A, WinA(S = S(i) ) == 1 (player B loses)
    then return 1
    else return 0.
```

The conventional tic-tac-toe is possibly not very interesting since we are already familiar with the possibilities. What makes a game interesting is the combinatorially explosive number of possibilities, say chess. Intuitively, the program is exploring all the possible ways in which the game can proceed and make a move that is most beneficial and once all the possibilities have been explored (which is not possible with the present technology for a game like chess), we have actually *solved the game* and it will cease to be interesting like the tic-tac-toe. However most programs only do *partial exploration* and make the next move corresponding to it (which may not be the best move if more explorations could be carried out).

14 Dynamic arrays and more on sorting - March 1

The function partition described in the midsem paper, groups the elements of an array into three contiguous regions based on a parameter x to the function. The first region consists of elements strictly smaller, the second region consists of elements which are equal (there may not be any) and the third region consists of elements strictly larger than x. The function manages to

[5] Many games like NIM has only two possibilities - Win or Lose.

do this quite efficiently, namely by making one pass through the array. Note that the regions of elements (except the middle one) are not necessarily sorted. Can we use the function partition to sort ?

One thing is clear - that the elements of region 2 are larger than the 1st region and the elements of the 3rd region are larger than the second region which gives us some partial ordering. Clearly if we can sort the first region and the 3rd region, then we can have the entire array sorted. The first and the third region can be sorted by recursively calling the function partition on those regions. With a little thought, you can argue that the parameter x should be chosen from the set of elements that we are trying to partition to guarantee that the algorithm terminates (we will have the problem sizes strictly shrinking). Moreover, to find out the extent of the regions for future recursive calls, we will modify the function partition to return the extent of the first region.[6].

The following is a complete program for sorting which is often called **Quicksort**

```
#include <stdio.h>

int partition ( int a[], int x, int l , int u) ;
/* The function partition takes four parameters - an array,
two indices, l and u and a number x. It partitions the subarray
a[l] .. a[u] into three regions, corresponding to elements < x,
==x and > x. It does so by performing some
exchanges and does not use an extra array. At any intermediate stage it
maintains the following invariant : R U W B, where R is the region corresponding
to elements < x, W is the region corresponding to elements == x and B
corresponds to elements > x. U corresponds to unclassified elements.
Initially all elements are unclassified and finally U is empty.
The three variables u,w,b in the function keeps track of the boundaries
of the four regions. In every iteration, a[w] is inspected and
the boundaries are modified accordingly. */
void qsort(int a[] , int l, int u)
 {
  int rank ;
  if ( u > l) {
  rank = partition(a, a[l], l, u) ;
  /*printf("\n The partitioned array is \n");*/
  if (rank > l)  qsort(a, l, rank) ;
  if (u > (rank +1)) qsort(a , rank+1, u);
   }
 }
```

[6]To return the boundary of the third region we have to use an extra parameter since a function returns only one value

```
 main ()

{
  int length, i, j;
  int *a ;

 printf("how many integers ?\n");
 scanf("%d", &length);
  a = (int *)malloc(length * sizeof(int)) ;
 printf("Supply %d integers in one line separated by spaces\n",length);
  for (i =0 ; i <= (length -1); i = i+1)
  {
  scanf("%d", &a[i]);
   }
  printf("\n The given integers are\n");
for (i =0 ; i <= (length -1); i = i+1)
  {
  printf("%d\n",a[i]);
   }

  qsort(a, 0, length-1) ;
   printf("\n The sorted array is \n");
 for (i =0 ; i <= (length -1); i = i+1)
  {
  printf("%d\n",a[i]);
   }

}

 int partition (int a[] , int x, int l, int u )

 {
  int r,w,b ; int t;

   r = l; w = u ; b = u;

 while ( w >= r)
  {
   if (a[w] < x ) { t = a[w]; a[w] = a[r]; a[r] = t;
                   r = r+1;}
       else {if (a[w] == x) {w = w -1;}
               else { t = a[w] ; a[w] = a[b] ; a[b] = t;
```

```
                        b = b-1; w = w-1;
                       }
               }
  }

    return r;
 }
```

14.1 Dynamic arrays

In the programs involving arrays, we have fixed the size of the array, namely we have not left it as an option to the user to specify the input size. Consequently if we need to run it on a larger input we must modify the size inside the program, recompile and then run it. There is a provision in C to define the size of an array *while the program is running.* The following piece of code achieves this

```
int *a, length ;

 printf("What is the length ?\n");
 scanf("%d", &length);
  a = (int *)calloc(length , sizeof(int)) ;
```

The instruction `calloc` requests an array of size `length` where each component of the array is an integer from the OS when the program is running. The memory management part of OS accedes to this request if it is available (will return NIL if not available) and returns a pointer to the first location of the array. This pointer doesn't have a type as it is at a very low level (where everything is a byte), so it is typecast into a pointer of the required type (in this case integer). For an array `a`, `&a[0]` is the same as `a` and `a[i]` refers to the i+1st location of the array. Therefore, the previous program can be made more flexible by dynamically declaring the size of the array and no further changes will be necessary.

Reading Exercise The instruction `malloc` is very similar that requests one object (rather than an array). Convince yourself that the instruction `malloc` can also serve the purpose of `calloc`. The instruction `realloc (p, size)` changes the size of the object pointed to by p to size.

The memory once allocated is under the purview of the program till it terminates. If the need for the memory is over before that, it is a good practice to *free* it explicitly to conserve space. The instruction `free(p)` frees the memory pointed to by p.

15 Structures - representing complex data

A file consists of *records* which may have several *fields*, each of which is some basic or derived data type. For example a file of complex numbers can be thought of as records with two fields both of which are floats. One of them represent the real part, the other represents the imaginary part.

The C syntax for representing a structure (or a record) is

```
struct complex { float realpart;
                 float img ;}
 complex x ; /* x is a variable of type complex*/
```

or equivalently

```
typedef complex_struct {
    float realpart;
    float img ;
   } complex; /* complex is the name of the structure defined above */
 complex x;
```

To refer/assign value to individual fields, we use the following instructions `x.realpart = 0.132 ; x.img =` If y is a pointer to a structure, say, `complex *y` , then we can write `y-> realpart = 0.5` instead of `(*y).realpart = 0.5`.

16 Ordered Lists - an abstract data type

An *ordered list* is collection of data items of the same type that is defined by *first element* and the remaining *ordered list*, i.e. it is an inductive definition. An ordered list can be an empty list also which is like the base case.

We would like to support various operations on this data type like inserting an element at a given position, deleting an element from a given position, counting the number of elements etc.

A character string is a very important ordered list data type and you can think of many operations on character strings. One of the most important feature of ordered lists is that it is dynamic, i.e. the length may change.

16.1 Implementation of Ordered List using arrays

By definition of arrays, there is a natural ordering among elements.

16.2 Implementation of Ordered List using pointers

A C implementation of this definition can be

```
typedef struct _node {
      char element; /*each component is a single character*/
      struct _node *next; /* the recursively defined part*/
   } node;

typedef node *string; /* string points to such a structure */
```

A recursive definition seems necessary to avoid the chicken-and-egg problem - how do we define the next pointer without the structure and how do we define the structure without the pointer.

17 Mergesort

All the previous soring algorithms that we have studied take roughly n^2 operations to sort n elements except Quicksort that has a good behavior for random permutations. One of the oldest and best known efficient algorithms for sorting is mergesort that is based on merging of two sorted arrays.

17.1 merging

Given two *sorted* arrays, we would like to combine them into a single sorted array. The straightforward method is based on the following simple observation, namely, the smallest element is the smaller of the two smallest elements in the two sorted sequences. If we move that into an output and delete it from the sequence where it occurred we are once again back to the problem of merging two sorted sequences. The procedure terminates when one of the sequences get exhausted and the remaining sequence is appended to the output.

The number of comparisons required to produce one output element is 1, and therefore the total cost of merging is the total number of elements in the two lists which is very efficient.

17.2 Sorting

The basic idea is as follows

1. Divide the input elements S into (roughly) equal parts, say S_1 and S_2.
2. Sort S_1 and S_2 recursively (base case can be size 2).
3. Merge S_1 and S_2 using the procedure described above.

Note that Step 1 involves only counting and dividing the number of elements into two parts which is straightforward. So the actual manipulations happen in Step 3, starting from base-size sequences. That is, we first have $n/2$ length 2 sorted sequences, followed by $n/4$, length 4 sorted sequences, till we have one sorted sequence.

Let the total number of operations required for a sequence of length n be $T(n)$. Then

$$T(n) = T(n/2) + T(n/2) + 2n$$

The solution for this using $T(2) = 1$ is $T(n) = 2n \log n$. You can also count the number of comparisons by counting the number of operations to pairwise merge $n/2$ length 2 sequences, $n/4$, length 4 sequences etc.

18 Generating Permutations, Mar 15

Given n objects, we will like to generate all possible permutations of the objects that are labelled using integers from 1 to n. For instance if all objects are distinct, then we have have the set $\{1, 2, 3, \ldots n\}$. For non-distinct objects, we will have some labels repeated.

There are various approaches that we can take to generate permutations - however we will like our solution to be both efficient in its use of space and time. Since permutations have a natural inductive definition, we can use it for our purpose. For every element[7] i, we fix i in the first position and generate recursively all the permutations of $\{1, 2 \ldots i-1, i+1 \ldots n\}$ objects.

Another similar approach is to generate all permutations of $\{1, 2, \ldots n-1\}$ recursively and for each such permutation, insert n in each of the n gaps. This approach is being left as an exercise.

The following program implements the first approach for distinct objects

```
#include <stdio.h>

void printarray ( int a[], int length)
{ int i;
for (i =0 ; i <= (length -1); i = i+1)
  {
  printf("%d ",a[i]);
   }
  printf("\n");

}
void permute (int a[] , int leftover , int length)
 {
  int rank, temp ;
  if ( leftover < (length-1))
      {
        for ( rank = leftover ; rank <= (length-1); rank++)
          {
             temp = a[leftover]; a[leftover] = a[rank]; a[rank] = temp;
             /* fixing position rank by swapping with a[leftover] */
             permute(a , leftover+1 , length);
             temp = a[leftover]; a[leftover] = a[rank]; a[rank] = temp;
           /* undoing the swap to get back the original permutation */


           }
      }
```

[7] assuming the distinct case

```
   else printarray(a, length);
 }
 main ()

{
  int length, i, j;
  int *a ;

 printf("how many integers ?\n");
 scanf("%d", &length);
  a = (int *)calloc(length,  sizeof(int)) ;
  for (i =0 ; i <= (length -1); i = i+1)
  {
  a[i] = i+1;
   }
 printarray(a , length);
  permute(a, 0 , length) ;

}
```

The following program generalises the previous approach to handle non-distinct objects. Note that the primary modification is that, we check if the object i has been considered in position `leftover`, before, by using the function new.

```
#include <stdio.h>

void printarray ( int a[], int length)
{ int i;
for (i =0 ; i <= (length -1); i = i+1)
  {
  printf("%d ",a[i]);
   }
  printf("\n");

}
 int new (int a[] , int leftover , int rank)  /* has a[rank] occurred
                                        between a[leftover] ..a[rank -1] ? */

  { int i , equal, pos ;
       equal = 1 ;
      for( i = leftover; i < rank ; i++)
         {
           if (a[rank] == a[i]) {equal = 0; pos = i;}
         }
      return equal ;
   }

void permute (int a[] , int leftover , int length)
 {
  int rank, temp ;
  if ( leftover < (length-1))
      {
        for ( rank = leftover ; rank <= (length-1); rank++)
          {
            if (new (a, leftover, rank))  /*only for distinct elements*/
             {
            /*printf("\n leftover , rank %d %d \n", leftover, rank);*/
           temp = a[leftover]; a[leftover] = a[rank]; a[rank] = temp;
             /* fixing position rank by swapping with a[leftover] */
             permute(a , leftover+1 , length);
             temp = a[leftover]; a[leftover] = a[rank]; a[rank] = temp;
           /* undoing the swap to get back the original permutation */
              /*printf("\n getting back\n"); printarray(a, length);*/

             }

           }
      }
   else printarray(a, length);
 }
 main ()

{
  int length, i, j;
  int *a ;

 printf("how many integers ?\n");
 scanf("%d", &length);
  a = (int *)calloc(length,  sizeof(int)) ;
  for (i =0 ; i <= (length -1); i = i+1)
  {
  a[i] = (i+1)% (length/2); /* generating some duplicate elements */
   }
 printarray(a , length);
  permute(a, 0 , length) ;
```

```
}
```

18.1 Analysis

In both the above programs, we are not using any extra space beyond array `a`. Moreover, a permutation is being *generated* exactly once - one can argue that the number of swaps is proportional to the number of permutations generated[8]. In the second program, one may be able to get a better implementation of he function `new`. able to

[8]consider the tree corresponding to the recursion where every leaf is a permutation. The number of swaps is proportional to the number of nodes in the tree which is proportional to the number of eaf nodes

19 The maze problem

Consider a rectangular maze of rooms where each adjacent room may or may not have a passage between them. We want to explore this maze starting from an initial room and covering as many rooms that can be reached from the initial room. We can move between adjacent rooms only if there is a passage between them. Any strategy that we adopt for solving this problem must ensure

- We visit all rooms that can be reached
- The search terminates, i.e. we should not be cycling endlessly.

19.1 Depth first strategy

From our present position i, we search all the neighbouring rooms one by one and after having done that we *backtrack* to $f(i)$, which is the room from which we arrived at i for the first time. To avoid cycles, we keep track of which neighbouring rooms have been explored, $f(i)$ (to backtrack). Moreover, if a certain room has been visited before we do not take that path.

Therefore at any point of time, we have a set of rooms (actually the doors) that we haven't completed exploring (in a reversed order of discovering them) and a list of rooms that have been visited along with their $f(i)$s.

We can formally prove that this strategy will discover all the reachable rooms by induction on the length of the paths [9]

19.2 Stacks

The set of rooms that we store in the reverse order of their discoveries also defines the order of backtracking, viz, if we discover room j after room i then we will backtrack from room i before we backtrack from room j. Once we backtrack we don't need to store that in the set.

Therefore we need to support the following operations on the set -

deleting the last element that was added
adding the next element that becomes the latest element to have been added.

Such a data organization is called *stack* - the position of the latest element is called *top* of stack - elements are deleted or added to he top of the stack. We also require a *bottom* marker to denote that the stack is empty.

Stacks may be implemented using (static or dynamic) arrays where the highest index in the array corresponds to top of stack and the zero-th index is the bottom of stack. Ordered lists can also be used to represent a stack where we have pointers to the first element and the last element can be the NULL pointer.

[9] Assume all rooms upto a certain distance i reachable, then all its neighbours are reachable

20 Breadth First Search

Here we search the neighbouring rooms, then the neighbours of the neighbouring rooms and then the neighbours of the neighbours and so on. We have a set of unexplored rooms according to some ordering - initially it is just the starting room. Whenever we find that a room X has not been visited, we schedule the (undiscovered) neighbours of the room for exploration *AFTER* the current set of rooms. Room X is removed from the set of rooms.

A formal proof can be worked out based on induction that all reachable rooms are visited using this strategy. Note that all rooms that are at a distance i from the starting room are visited before distance $i + 1$ nodes.

20.1 Queues

With a little thought the access pattern can be made out to be as follows - remove the highest priority element from the set and replace it by a set of elements (possibly empty) of the lowest priority. In other words, we need a data organization that supports operations of the following kind -

> add an element that has the lowest priority (all elements existing in the set have higher priority)
> delete the element with the highest priority

In addition, we want to check if the set is empty. This data structure is called *Queue* and can be easily implemented using static arrays or ordered lists. Unlike Stacks, we need to keep track of both the front and the back of the queue.

In the static array implementation, it is better if we use the indices modulo the size of the array (also known as *circular queues*) - which enables maximum space utilization of the array. As we add and delete elements, the front and the back index move in the same direction, so by letting them wrap-around (modulo calculation), we will not run out of space as long as the array has empty positions.