DATA STRUCTURES

A Data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.

Data structures can implement one or more particular abstract data types (ADT), which specify the operations that can be performed on a data structure and the computational complexity of those operations. In comparison, a data structure is a concrete implementation of the space provided by an ADT.

(or)

A data structure is a format of storing, organizing, transforming and retrieving data in a computer so that it can be used efficiently.

Performance and Complexity Analysis:

Performance : The amount of computer memory and the time needed to run a program.

There are two approaches to determine the performance of a program.

a) Performance analysis - use analytical methods
b) Performance measurement – Conduct experiments.

Space complexity : Space complexity of the program is the amount of memory it needs to run to completion. For following reasons the space complexity is needed.

1. If the Program is to be run on a multiuser computer system, then we may need to specify the amount of memory to be allocated to the program.
2. It may be needed to know in advance whether or not sufficient memory is available to run the program.
3. A problem might have several possible solutions with different space requirements.
   Eg: c++ compiler with 1MB, c++ compiler with 4MB
4. Use the space complexity to estimate the size of the largest problem that a program can solve.

Components of Space Complexity:

The space needed by a program has the following components.

1. Instruction space: Space needed to store the compiled version of the program instructions.
   The amount of instruction space that is needed depends on factors such as
   i. The compiler used to compile the program into machine code.
   ii. The compiler options in effect at the time of compilation.
   iii. The target computer.

   The compiler is the important factor in determining how much space the resulting code needs. Eg., Possible codes for the evaluation of a+b+b*c+(a+b-c)/(a+b)+4 are

(a)
```
LOAD  a
ADD b
STORE t1
LOAD b
MULT c
STORE t2
LOAD t1
ADD t2
STORE t3
LOAD t1
ADD t2
STORE t3
LOAD a
ADD b
SUB c
STORE t4
LOAD a
ADD b
STORE t5
LOAD t4
DIV t5
STORE t6
LOAD t3
ADD t6
ADD 4
```

b)
```
LOAD A
ADD b
STORE t1
SUB c
DIV t1
STORE t2
LOAD b
MUL c
STORE t3
LOAD b
MUL C
STORE t3
LOAD t1
ADD t3
ADD t2
ADD 4
```

c)
```
LOAD a
ADD b
STORE t1
SUB c
DIV t1
STORE t2
LOAD B
MUL c
ADD t2
ADD t1
ADD 4
```

These codes need different amount of space, and the compiler in use determines exactly which code will be generated. Even with the same compiler the size of the generated program code can vary. Space is required for the temporary variables.

Another option that can have a significant effect on program space is the overlay option in which space is assigned only to the program module that is currently executing. When a new module is invoked , then the code of the new module is read in from disk and it overwrites the old module. So the program space corresponding to the size of the largest code module is needed.

The configuration of the target computer also can effect the size of the compiled code.

2. Data space: Space needed to store all the constants and variable values. Data space has two components:
    a. Space needed by constants and simple variables.
    b. Space needed by dynamically allocated objects such as arrays and class instances.

Different data types have different space requirement. Eg., bool (1 byte) , char (1 byte) , int (4 bytes) etc.,

Space required for the structure variable is sum of the space of all its components.

Space required for array is multiplying array size and the space needs of a single array element.

Eg., double a[100]; space needed is 100* 8(double= 8 bytes)

Int maze[rows][cols]; rows*cols*4(int = 4 bytes)

3. Environmental stack space: The environment stack space is needed to resume execution of partially completed functions and methods. Each time a function is invoked the following data are saved on the environment stack.
   a. The return address
   b. The values of all local variables and the formal parameters in the function being invoked.

The total space needed by a program is divided into two parts:

1) A fixed part that is independent of instance characteristics which include instruction space, space for simple variables, and space for constants and so on.
2) A variable part that consists of the dynamically allocated space and the recursion space.

The space requirement of any program P may be written as

$c + S_p$(instance characteristics) where c is a constant that denotes the fixed part and S_p denotes the variable part. Accurate analysis should also include the space required for temporary variables generated during compilation.

Examples

1)
```
Template<class T>
int seqsearch(T a[],int n,const T& x)
{
 int i;
 for(i=0;i<n && a[i]!=x;i++)
if (i==n) return -1;
else return i;
}
```

The space complexity of this function in terms of instance characteristics ‘n’ is $S_{sequentialsearch}(n)=0$, Although the space for the formal parameters a, x, n, the constants 0 and -1, and space for code is needed ,but this space is independent of n.

2)
```
Template<class T>
T sum(T a[],int n)
{
   T theSum=0;
    for (int i=0;i<n;i++)
        theSum+=a[i];
    return theSum;
}
```

Here also S_{sum} (n)=0, with in the function space is needed for formal parameters , the local variables i and theSum, the constant 0 , and the instructions. The amount of space needed does not depend on value of n.

3)
```
int factorial(int n)
{
  if (n<=1) return 1;
  else return n* factorial(n-1);
}
```

The recursion depth is max{n,1}. The recursion stack saves a return address (4 bytes) and the value of n (4 bytes) each time the factorial is invoked. No additional space that is dependent on n is used, so $S_{factorial}$ (n) = 8*max{n,1}.

Time complexity: Time complexity of the program is the amount of computer time needed to run to completion. For following reasons the space complexity is needed.

1. Some Computer systems require the user to provide an upper limit on the amount of time the program will run. Once the upper limit is reached , the program is aborted.
2. The program developed might need to provide a satisfactory real –time response .From the time complexity of the program or program module , it is possible to decide whether or not the response time will be acceptable. If not, not acceptable either redesign the algorithm or give the user faster computer.
3. If we have alternative ways to solve a problem, then the decision on which to use will be based on the expected performance difference among these solutions.

Components of Time Complexity: The time complexity of a program depends on all the factors that the space complexity depends.

A program will run faster on a computer capable of executing 10^9 instructions/second than on one that can execute only 10^7 instructions/second.

Some compilers will take less time than others to generate the corresponding code.

Smaller problem instances will generally take less time than larger instances. The time taken by a program P is the sum of the compile time and run time. Compile time does not depend on instance characteristics. Compiled code can be run several times without recompilation. ∴ , Consider only run time of a program. This run time is denoted by t_P(instance characteristics). Because many of the factors t_P depends are not known when a program is created. Therefore only possibility is to estimate t_P.

If we knew the characteristics of the compiler used, then we could determine the number of additions , subtractions ,multiplications ,divisions, compares, loads, stores, and so on that the code for P would make. Then

$t_P(n)= c_aADD(n)+c_sSUB(n)+c_mMUL(n)+c_dDIV(n)+$…….. (a)

where c_a ,c_s ,c_m,c_d denote time taken for addition , subtraction, multiplication and division .

ADD,SUB,MUL,DIV are functions whose value is the number of additions, subtractions, multiplications, divisions that will be performed when the code for P is used on an instance with characteristic n.
The time needed for arithmetic operation depends on the type (int, float ,etc.,) of the numbers in the operation, an exact formula for run time must separate the operation counts by data type.
Fine-tuning the equation (a) this way still does not give us an accurate formula to predict run time because computers do not necessarily perform arithmetic operation in sequence.

Eg.,

a) computers can perform an integer operation and float operation at the same time.
b) Computers can have capability to pipeline arithmetic operations and also computers have memory hierarchy which means the time to perform m additions isn't necessarily m times the time to perform one.

Two more manageable approaches to estimating run time are

(1) Identify one or more key operations and determine the number of times these operations are done.
(2) Determine the total number of steps executed by the program.

OPERATION COUNTS : One of the method is to estimate time complexity of the program or method is to find the operations(add, sub, compare etc.,) and determine how many times each is done. The success of this method depends on the ability to identify the operations that contribute most to the time complexity.

Example 1 : To find the position of largest element in the given array.

```
Template<class T>
int indexOfMax(T a[], int a)
{
  if (n<=0)
       throw illegalParameterValue("n must be greater than 0");
   int indexOfMax = 0;
   for (int i=1; i<n;i++)
     if (a[indexOfMax] < a[i])
          indexOfMax = i;
return indexOfMax;
}
```

The time complexity of the program can be estimated by determining the number of comparisons made between the elements of array a.

Case 1: when n<=0 , no. of comparisons is 0. Exception is thrown.
Case 2: when n=1, the for loop is not entered. So no comparisons between elements of the array a are made.
Case 3: when n>1, the no. of comparison is n-1.

The number of element comparisons is max {n-1,0}.
Initializing of indexOfMax, for loop index, incrementing of for loop index, comparison of for loop index not included in the estimate of time. If included the time will be increased by a constant factor.

Example 2 : Polynomial evaluation

Consider the polynomial P(x)= $\sum_{i=0}^{n} c_i x^n$. if $c_n \neq 0$,P(x) is a polynomial of degree n.

```
template<class t>
t polynomial(t coeff[],int n,const t&x)
{
    t y = 1 , value = coeff[0];
    for (int i =1 ; i<=n; i++)
    {
      y = y *x;
      value += y* coeff[i];
}
return value;
}
```

It's time complexity is estimated by determining the number of additions and multiplications performed inside the for loop.
Degree n is used as instance characteristics.
The for loop is entered total n times. Each time the for loop is entered one addition and two multiplications are done. The number of additions is n, and the no.of multiplications is 2n.

Evaluation of the Polynomial is done by using Horner's rule as:
$P(x) = (\ldots(c_n * x + c_{n-1})*x + c_{n-2})*x + c_{n-3})*x\ldots)*x+c_0$
Then the polynomial evaluation function is :

```
template<class t>
t polynomial(t coeff[],int n,const t&x)
{
    t y = 1 , value = coeff[n];
    for (int i =1 ; i<=n; i++)

      value = value *x +  coeff[n-1];
return value;
}
```

The time complexity of this modified function in terms of no.of additions and no.of multiplications is : n additions and n multiplications.