CHAPTER 1

BASIC CONCEPT

All the programs in this file are selected from

Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
"Fundamentals of Data Structures in C /2nd Edition",
Silicon Press, 2008.

How to create programs

- Requirements
- Analysis: bottom-up vs. top-down
- Design:
- Refinement and Coding
- Verification
  - Program Proving
  - Testing
  - Debugging

Algorithm

- Definition
  An *algorithm* is a finite set of instructions that accomplishes a particular task.
- Criteria
  - input
  - output
  - definiteness: clear and unambiguous
  - finiteness: terminate after a finite number of steps
  - effectiveness: instruction is basic enough to be carried out

Data Abstraction(1/2)

- Data Type

  A *data type* is a collection of and a set of that act on those objects.

- Abstract Data Type

  An *abstract data type(ADT)* is a data type that is organized in such a way that the specification of the objects and the operations on the objects is separated from the representation of the objects and the implementation of the operations.

Data Abstraction(2/2)

- Basic Data types of C :
  - char, int, float, double
  -
- Grouping Data types of C :
  - Array
  -

*Structure 1.1: Abstract data type *Natural_Number*

structure Natural_Number is

objects: an ordered subrange of the integers starting at zero and ending at the maximum integer (INT_MAX) on the computer

functions:

for all x, y ∈ *Nat_Number*; *TRUE*, *FALSE* ∈ *Boolean*
and where +, -, <, and == are the usual integer operations.

Nat_No Zero ( ) ::= 0

Boolean Is_Zero(x) ::= **if** (x) **return** *FALSE*
else return *TRUE*

Nat_No Add(x, y) ::= **if** ((x+y) <= *INT_MAX*) **return** x+y
else return *INT_MAX*

Boolean Equal(x,y) ::= **if** (x== y) **return** *TRUE*
else return *FALSE*

Nat_No Successor(x) ::= **if** (x == *INT_MAX*) **return** x
else return x+1

Nat_No Subtract(x,y) ::= **if** (x<y) **return** 0
else return x-y

end *Natural_Number*

::= is defined as

Measurements

- Criteria
  - Is it correct?
  - Is it readable?
  - …

Space Complexity

- Space Requirements ( )
  Independent of the characteristics of the inputs and outputs
  - instruction space
  - space for simple variables, fixed-size structured variable, constants
- Space Requirements ( )
  depend on the instance characteristic I
  - number, size, values of inputs and outputs associated with I
  - recursive stack space, formal parameters, local variables, return address

Pointers

```
int i, *pi;
float j, *pj;
pi = &i;     // i=10 or *pi=10
pj = &j;     // j=1.5 or *pj=1.5

if (pi==NULL)
if (!pi)
```

Dynamic Memory Allocation

```
int i, *pi;
float f, *pf;
pi = (int *) malloc(sizeof(int));
pf = (float *) malloc (sizeof(float));
*pi =1024;
*pf =3.14;
printf(”an integer = %d, a float = %f\n”, *pi, *pf);
free(pi);
free(pf);
```

Array

```
int  a[5]={1, 2, 3, 4, 5};
…
fun(a, 5);
…

void(int b[], int size)
{
   int i;
   for(i=0; i<size; i++)
      b[i]*=2;
}
```

```
float abc(float a, float b, float c)
{
   return a + b + b * c + (a + b - c) / (a + b) + 4.00;
 }
```

```
float sum(float list[ ], int n)
{
  float tempsum = 0;
  int i;
  for (i = 0; i<n; i++)
     tempsum += list [i];
  return tempsum;
}
```

```
float rsum(float list[ ], int n)
{
  if (n) return rsum(list, n-1) + list[n-1];
  return 0;
 }
```

Assumptions:

Type	Name	Number of bytes
parameter: array pointer parameter: integer return address:(used internally)	list [ ] n	
TOTAL per recursive call		

Time Complexity

$$T(P)=C+T_P(I)$$

- independent of instance characteristics
-
- Definition

$$\boxed{T_P(n)=c_a ADD(n)+c_s SUB(n)+c_l LDA(n)+c_{st} STA(n)}$$

  A is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.
- Example
  – abc = a + b + b * c + (a + b - c) / (a + b) + 4.0
  – abc = a + b + c

Methods to compute the step count

- Introduce variable count into programs
- Tabular method
  - Determine the total number of steps contributed by each statement

  - add up the contribution of all statements

Iterative summing of a list of numbers

```
float sum(float list[ ], int n)
{
   float tempsum = 0;                    /* for assignment */
   int i;
   for (i = 0; i < n; i++) {
                                    /*for the for loop */
         tempsum += list[i];               /* for assignment */
   }
                    /* last execution of for */
   return tempsum;
                   /* for return */
}
```

```
float sum(float list[ ], int n)
{
   float tempsum = 0;
   int i;
   for (i = 0; i < n; i++)
       count += 2;
   count += 3;
   return 0;
}
```

Recursive summing of a list of numbers

```
float rsum(float list[ ], int n)
{
                                /*for if conditional */
        if (n) {
                                   /* for return and rsum invocation */
                return rsum(list, n-1) + list[n-1];
        }

        return list[0];
}
```

Matrix addition

```
void add( int a[ ] [MAX_SIZE], int b[ ] [MAX_SIZE],
                    int c [ ] [MAX_SIZE], int rows, int cols)
{
   int i, j;
   for (i = 0; i < rows; i++)
     for (j= 0; j < cols; j++)
       c[i][j] = a[i][j] +b[i][j];
}
```

```
void add(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE],
                  int c[ ][MAX_SIZE], int row, int cols )
{
  int i, j;
  for (i = 0; i < rows; i++){
                  /* for i for loop */
      for (j = 0; j < cols; j++) {
                  /* for j for loop */
        c[i][j] = a[i][j] + b[i][j];
                  /* for assignment statement */
      }
                  /* last time of j for loop */
  }
                  /* last time of i for loop */
}
```

```
void add(int a[ ][MAX_SIZE], int b [ ][MAX_SIZE],
                    int c[ ][MAX_SIZE], int rows, int cols)
{
   int i, j;
   for( i = 0; i < rows; i++) {
      for (j = 0; j < cols; j++)
         count += 2;
      count += 2;
   }
   count++;
}
```

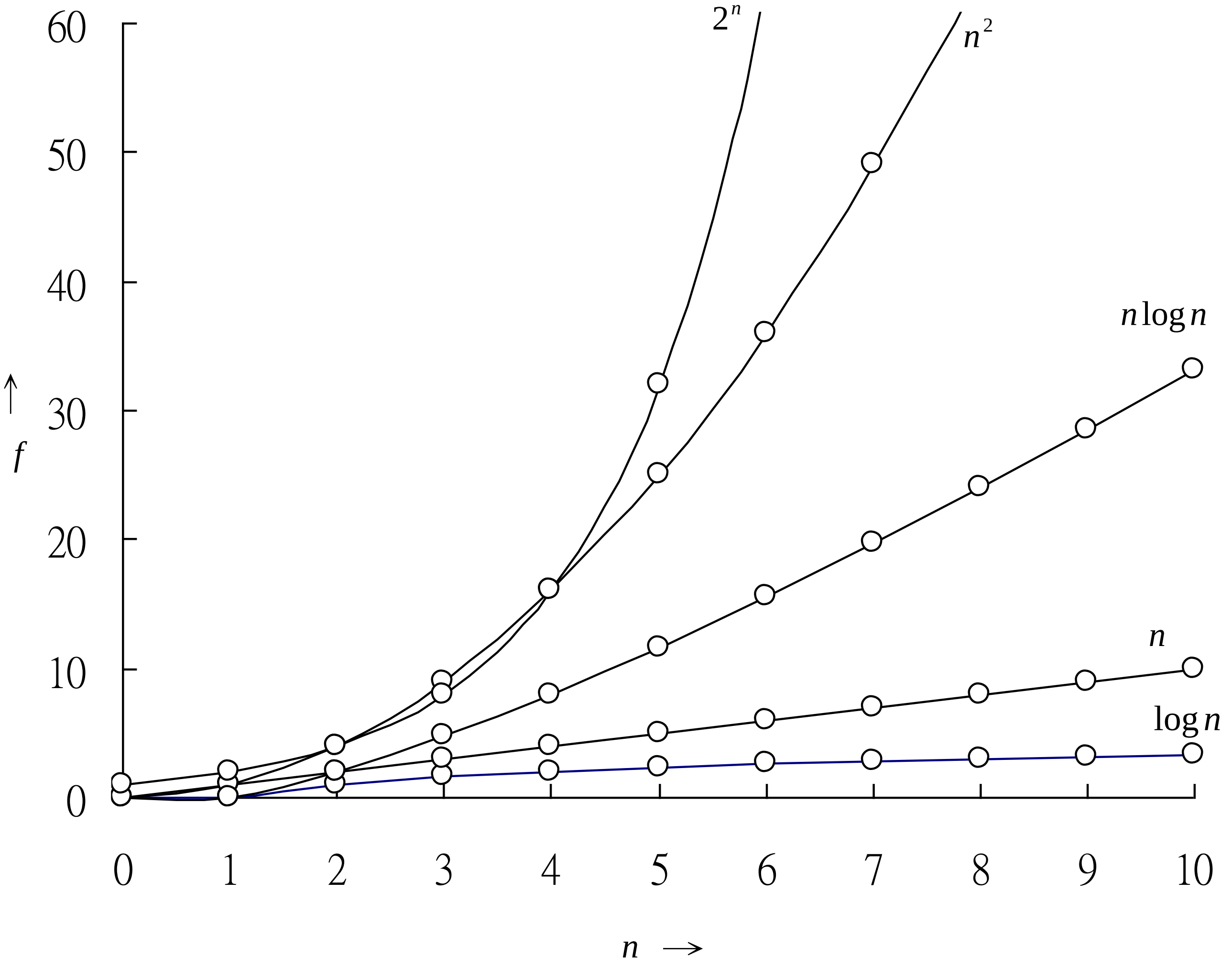
Asymptotic Notation (O)

- Definition
  $f(n) = O(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \le cg(n)$ for all n, $n \ge n_0$.
- Examples
  - $3n+2=O(n)$ /* $3n+2\le 4n$ for $n\ge 2$ */
  - $3n+3=O(n)$ /* $3n+3\le 4n$ for $n\ge 3$ */
  - $100n+6=O(n)$ /* $100n+6\le 101n$ for $n\ge 10$ */
  - $10n^2+4n+2=O(n^2)$ /* $10n^2+4n+2\le 11n^2$ for $n\ge 5$ */
  - $6*2^n+n^2=O(2^n)$ /* $6*2^n+n^2 \le 7*2^n$ for $n\ge 4$ */

- : Constant
- : Logarithmic
- : Linear
- : Log Linear
- : Quadratic
- : Cubic
- : Exponential
- : Factorial

$\log n$	n	$n\log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1024	32,768	4,294,967,296

		Instance characteristic n					
Time	Name	1	2	4	8	16	32
1	Constant	1	1	1	1	1	1
$\log n$	Logarithmic	0	1	2	3	4	5
n	Linear	1	2	4	8	16	32
$n \log n$	Log linear	0	2	8	24	64	160
n^2	Quadratic	1	4	16	64	256	1024
n^3	Cubic	1	8	64	512	4096	32768
2^n	Exponential	2	4	16	256	65536	4294967296
$n!$	Factorial	1	2	24	40326	20922789888000	26313×10^{33}



60
50
40
30
20
10
0
f
2^n
n^2
$n \log n$
n
$\log n$
0
1
2
3
4
5
6
7
8
9
10
n

	Time for $f(n)$ instructions on a 10^9 instr/sec computer						
n	$f(n)=n$	$f(n)=\log_2 n$	$f(n)=n^2$	$f(n)=n^3$	$f(n)=n^4$	$f(n)=n^{10}$	$f(n)=2^n$
10	.01μs	.03μs	.1μs	1μs	10μs	10sec	1μs
20	.02μs	.09μs	.4μs	8μs	160μs	2.84hr	1ms
30	.03μs	.15μs	.9μs	27μs	810μs	6.83d	1sec
40	.04μs	.21μs	1.6μs	64μs	2.56ms	121.36d	18.3min
50	.05μs	.28μs	2.5μs	125μs	6.25ms	3.1yr	13d
100	.10μs	.66μs	10μs	1ms	100ms	3171yr	$4*10^{13}$yr
1,000	1.00μs	9.96μs	1ms	1sec	16.67min	$3.17*10^{13}$yr	$32*10^{283}$yr
10,000	10.00μs	130.03μs	100ms	16.67min	115.7d	$3.17*10^{23}$yr	
100,000	100.00μs	1.66ms	10sec	11.57d	3171yr	$3.17*10^{33}$yr	
1,000,000	1.00ms	19.92ms	16.67min	31.71yr	$3.17*10^{7}$yr	$3.17*10^{43}$yr	

μs = microsecond = 10^{-6} seconds
ms = millisecond = 10^{-3} seconds
sec = seconds
min = minutes
hr = hours
d = days
yr = years