Time Complexity

Time Complexity

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

- Compile time (C)
  independent of instance characteristics
- run (execution) time T_P
- Definition

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

- A *program step* 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
    step per execution $\times$ frequency
  - add up the contribution of all statements

Iterative summing of a list of numbers

***Program 1.12:** Program 1.10 with count statements (p.23)

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

2n + 3 steps

***Program 1.13:** Simplified version of Program 1.12 (p.23)

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

2n + 3 steps

Recursive summing of a list of numbers

***Program 1.14:** Program 1.11 with count statements added (p.24)

```
float rsum(float list[ ], int n)
{
count++;       /*for if conditional */
  if (n) {
count++;  /* for return and rsum invocation */
    return rsum(list, n-1) + list[n-1];
  }
count++;
return list[0];
}
```

$\boxed{2n+2}$

Recursive summing of a list of numbers

***Program 1.14:** Program 1.11 with count statements added (p.24)

```
float rsum(float list[ ], int n)
{
count++;       /*for if conditional */
if (n) {
count++;  /* for return and rsum invocation */
return rsum(list, n-1) + list[n-1];
}
count++;
return list[0];
}
```

$\boxed{2n+2}$

Matrix addition

***Program 1.15:** Matrix addition (p.25)*

```
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];
}
```

***Program 1.16:** Matrix addition with count statements (p.25)

```
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++){
        count++; /* for i for loop */
        for (j = 0; j < cols; j++) {
            count++; /* for j for loop */
            c[i][j] = a[i][j] + b[i][j];
            count++; /* for assignment statement */
        }
        count++;    /* last time of j for loop */
    }
    count++;        /* last time of i for loop */
}
```

$\boxed{2\ rows * cols + 2\ rows + 1}$

***Program 1.17:** Simplification of Program 1.16 (p.26)

```
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++;
}
```

$2rows \times cols + 2rows + 1$

Suggestion: Interchange the loops when rows >> cols

Tabular Method

***Figure 1.2:** Step count table for Program 1.10 (p.26)*

Iterative function to sum a list of numbers

steps/execution

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

Recursive Function to sum of a list of numbers

***Figure 1.3:** Step count table for recursive summing function (p.27)

Statement	s/e	Frequency	Total steps
float rsum(float list[ ], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list, n-1)+list[n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
Total			2n+2

Matrix Addition

***Figure 1.4:** Step count table for matrix addition (p.27)

Statement	s/e	Frequency	Total steps
Void add (int a[ ][MAX_SIZE]• • •)	0	0	0
{	0	0	0
int i, j;	0	0	0
for (i = 0; i < row; i++)	1	rows+1	rows+1
for (j=0; j< cols; j++)	1	rows• (cols+1)	rows• cols+rows
c[i][j] = a[i][j] + b[i][j];	1	rows• cols	rows• cols
}	0	0	0
Total			2rows• cols+2rows+1

Exercise 1

***Program 1.18:** Printing out a matrix (p.28)*

```
void print_matrix(int matrix[ ][MAX_SIZE], int rows, int cols)
{
    int i, j;
    for (i = 0; i < row; i++) {
        for (j = 0; j < cols; j++)
            printf("%d",  matrix[i][j]);
        printf( "\n");
    }
}
```

Exercise 2

***Program 1.19:Matrix multiplication function**(p.28)

```
void mult(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE], int c[ ][MAX_SIZE])
{
    int i, j, k;
    for (i = 0; i < MAX_SIZE; i++)
    for (j = 0; j< MAX_SIZE;  j++) {
        c[i][j] = 0;
    for (k = 0; k < MAX_SIZE; k++)
        c[i][j]  +=  a[i][k] * b[k][j];
    }
}
```

Exercise 3

***Program 1.20:Matrix product function**(p.29)

```
void prod(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE], int c[ ][MAX_SIZE],
                                        int rowsa, int colsb, int colsa)
{
    int i, j, k;
    for (i = 0; i < rowsa; i++)
        for (j = 0; j< colsb;  j++) {
            c[i][j] = 0;
            for (k = 0; k< colsa; k++)
                c[i][j]  +=  a[i][k] * b[k][j];
        }
}
```

Exercise 4

***Program 1.21:Matrix transposition function** (p.29)

```
void transpose(int a[ ][MAX_SIZE])
{
    int i, j, temp;
    for (i = 0; i < MAX_SIZE-1; i++)
        for (j = i+1; j < MAX_SIZE;  j++)
            SWAP (a[i][j], a[j][i], temp);
}
```

Asymptotic Notation (O)

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

Example

- Complexity of $c_1n^2+c_2n$ and c_3n
  - for sufficiently large of value, c_3n is faster than $c_1n^2+c_2n$
  - for small values of n, either could be faster
    - $c_1=1, c_2=2, c_3=100$ --> $c_1n^2+c_2n \leq c_3n$ for $n \leq 98$
    - $c_1=1, c_2=2, c_3=1000$ --> $c_1n^2+c_2n \leq c_3n$ for $n \leq 998$
  - break even point
    - no matter what the values of c1, c2, and c3, the n beyond which c_3n is always faster than $c_1n^2+c_2n$

- $O(1)$: constant
- $O(n)$: linear
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(2^n)$: exponential
- $O(\log n)$
- $O(n\log n)$

***Figure 1.7:** Function values (p.38)*

		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}

*Figure 1.8: Plot of function values(p.39)

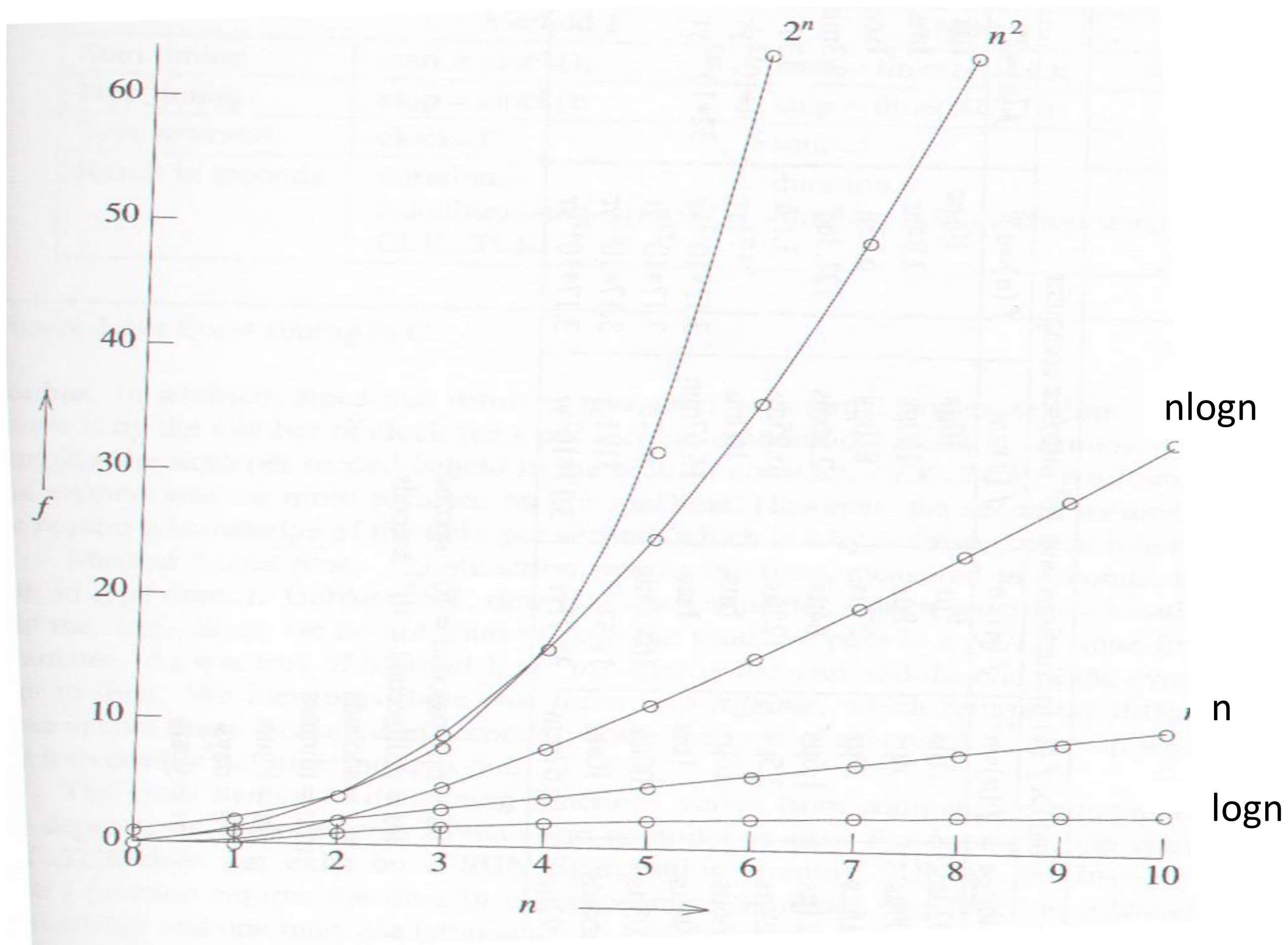

*Figure 1.9: Times on a 1 billion instruction per second computer(p.40)

	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