Chap 1. Basic Concepts (2)

Contents

Performance Evaluation

- Performance analysis
  - obtaining estimates of time and space that are *machine independent*

- Performance measurement
  - obtaining *machine-dependent* running time

1.5 Performance Analysis

Definition

- ***Space complexity*** (공간복잡도) :
  the amount of memory that a program needs to run to completion

- ***Time complexity*** (시간복잡도) :
  the amount of computation time that a program needs to run to completion

1.5.1 Space Complexity

- $\boldsymbol{S(P) = c + S_P(I)}$
  - $S(P)$: total space requirement of a program P
  - c : constant for fixed space requirement
  - $S_P(I)$:
    - the variable space requirement of a program P working on an instance I
    - a function of some *characteristics* of the instance I, where the characteristics include the *number, size, and values of the I/O associated with I*

- $S_P(I)$
  - Ex)
    - If the input is an array containing *n* numbers, then *n* is an instance characteristic.
    - If *n* is the only characteristic we wish to use, when computing $S_P(I)$, we will use $S_P(n)$ to represent $S_P(I)$.

  - When analyzing the space complexity of a program, we are usually concerned with $S_P(I)$.

- **Example : abc**
  - has only fixed space requirements
  - $S_{abc}(I) = 0$

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

Program 1.10: Simple arithmetic function

- **Example : sum**
  - Input includes an array with size n
  - $S_{sum}(I)$
    - Depends on how the array is passed into the function
    - C passes the address of the first element of the array

$$S_{sum}(I) = S_{sum}(n) = 0$$

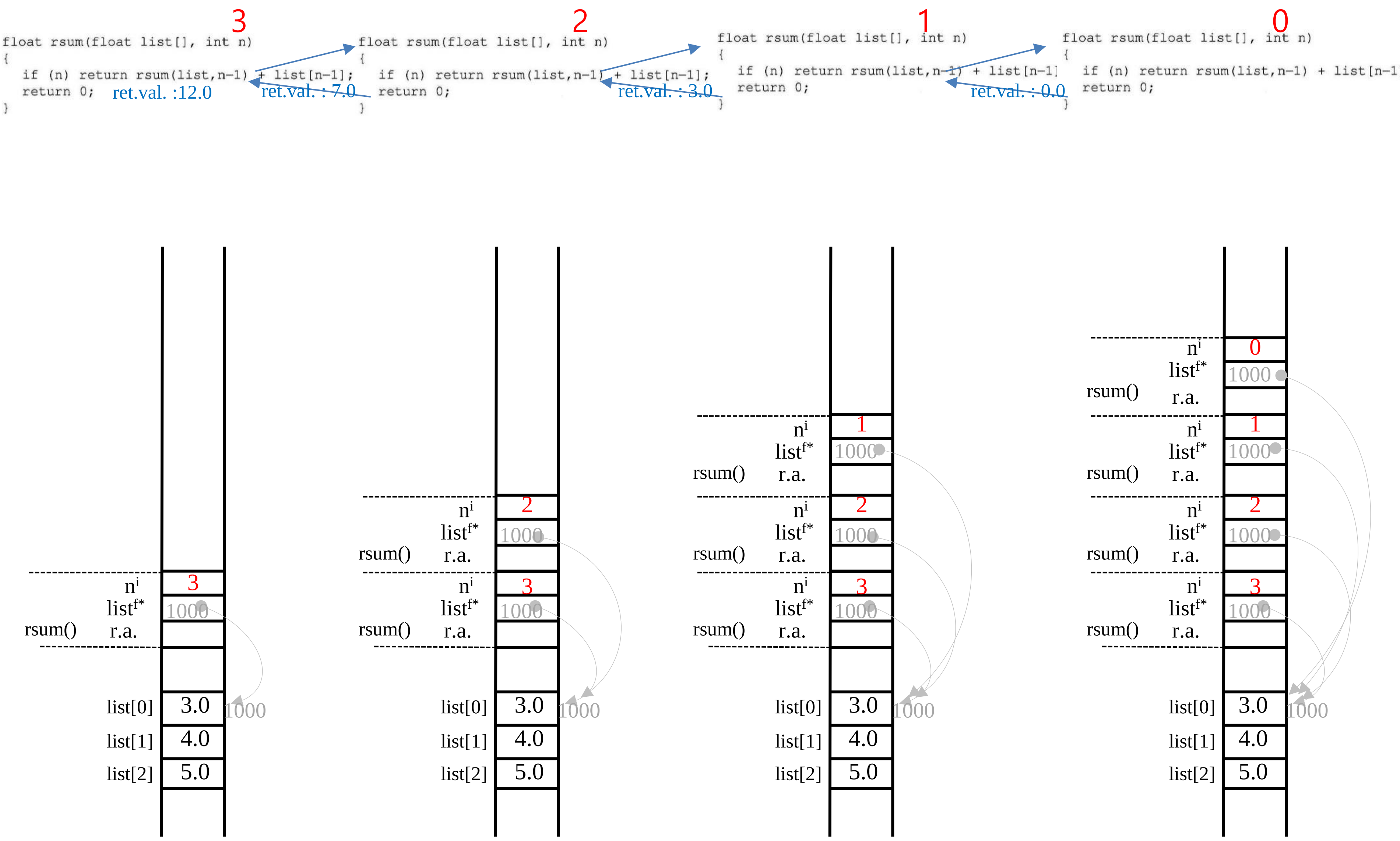
```
float sum(float list[], int n)
{
   float tempsum = 0;
   int i;
   for (i = 0; i < n; i++)
      tempsum += list[i];
   return tempsum;
}
```

Program 1.11: Iterative function for summing a list of numbers

- **Example : rsum**
  - for each recursive call, compiler must save
    - the parameters, local variables, the return address for each recursive call

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

Program 1.12: Recursive function for summing a list of numbers



3
float rsum(float list[], int n)
{
if (n) return rsum(list,n-1) + list[n-1];
return 0;
}
ret.val. :12.0
ret.val. : 7.0
2
float rsum(float list[], int n)
{
if (n) return rsum(list,n-1) + list[n-1];
return 0;
}
ret.val. : 3.0
1
float rsum(float list[], int n)
{
if (n) return rsum(list,n-1) + list[n-1]
return 0;
}
ret.val. : 0.0
0
float rsum(float list[], int n)
{
if (n) return rsum(list,n-1) + list[n-1]
return 0;
}
rsum()
n^i
list^f*
r.a.
3
1000
list[0]
list[1]
list[2]
3.0
4.0
5.0
1000
2
1
0

stack

Type	Name	Number of bytes
parameter: array pointer	*list*[]	4
parameter: integer	*n*	4
return address: (used internally)		4
TOTAL per recursive call		12

Figure 1.1: Space needed for one recursive call of Program 1.12

- If the array has n = *MAX_SIZE* numbers, then $S_{rsum}(MAX\_SIZE) = 12 * MAX\_SIZE$
- The recursive version has a far greater overhead than its iterative counterpart!!!

1.5.2 Time Complexity

- **$T(P)$ = compile time + execution time(T_P)**
  - $T(P)$: time taken by a program P
  - compile time : fixed, or without recompilation

- **We are really concerned only with the T_P !!!**
  - But, determining T_P is not an easy task
  - Alternatively, we could count *program step*.
    - a machine independent estimate

Definition

- ***program step*** :
  Syntactically or semantically meaningful program segment *whose execution time is independent of the instance characteristics.*

Ex)

a = 2;	*1 step*
*a = 2*b+3*c/d-e+f/g/a/b/c;*	*1 step*

- **How to count program steps? 2 ways!**

1. using a global variable, *count* with initial value 0
2. using a tabular method

– Only consider the program steps required by each *executable statement*

- **Example [Iterative summing]**

```
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 */
   count++; /* for return */  return tempsum;
}
```

Program 1.13: Program 1.11 with count statements

- The simplification makes easier to express the count arithmetically.
- The final *count* value will be **2*n*+3**

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

Program 1.14: Simplified version of Program 1.13

- **Example [Recursive summing]**
  - for $n = 0$, *count* is 2.
  - for $n > 0$, n recursive calls, *count* $= 2n$
  - the final *count* value will be $\mathbf{2n+2}$

```
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 0;
}
```

Program 1.15: Program 1.12 with count statements added

- **Example [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];
}
```

Program 1.16: 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++) {
    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 */
}
```

Program 1.17: Matrix addition with count statements

– The final *count* value will be $2rows \cdot cols + 2rows + 1$

- We should interchange the matrices if the number of rows is significantly larger than the number of columns.

```
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++;
}
```

Program 1.18: Simplification of Program 1.17

- **Example**

※ s/e : step counts per each statement

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			$2n+3$

Figure 1.2: Step count table for Program 1.11

- **Example**

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 0;`	1	1	1
}	0	0	0
Total			$2n+2$

Figure 1.3: Step count table for recursive summing function

- **Example**

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<rows; i++)	1	$rows+1$	$rows+1$
for (j = 0; j < cols; j++)	1	$rows \cdot (cols+1)$	$rows \cdot cols + rows$
c[i][j] = a[i][j] + b[i][j];	1	$rows \cdot cols$	$rows \cdot cols$
}	0	0	0
Total			$2rows \cdot cols + 2rows+1$

Figure 1.4: Step count table for matrix addition

- Three kinds of step count
  - Best-case step count
    - The minimum number of steps that can be executed for given parameters
  - Worst-case step count
    - The maximum number of steps that can be executed for given parameters
  - Average step count
    - The average number of steps on instance with the given parameters

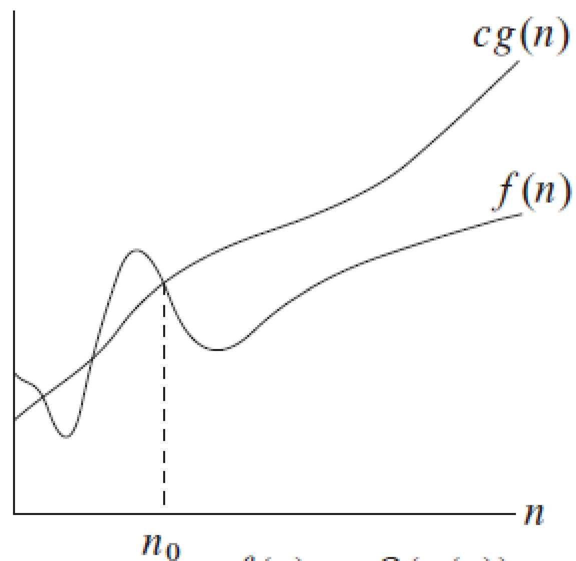
1.5.3 Asymptotic Notation (O, Ω, Θ)

- Motivation to determine *step counts*
  - Compare the time complexities of two programs for the same function
  - Predict the growth in run time as the instance characteristics change
- However,
  - Determining the exact step count is very difficult.
  - *The exact step count is not very useful for comparative purposes*.
    - The notion of a step is itself inexact.

- **Asymptotic notations (O, Ω, Θ)**
  - Enable us to make *meaningful* (but *inexact*) statement about *the time and space complexities* of a program

Definition [*Big "oh"*]

f(n)=O(g(n)) (read as "f of n is big oh of g of n")
iff $\exists c, n_0 > 0$, s.t. $f(n) \le cg(n) \;\; \forall n, n \ge n_0$



$f(n) = O(g(n))$

- Examples

$n \geq 2, \quad 3n + 2 \leq 4n \quad \Rightarrow 3n + 2 = O(n)$

$n \geq 3, \quad 3n + 3 \leq 4n \quad \Rightarrow 3n + 3 = O(n)$

$n \geq 10, \quad 100n + 6 \leq 101n \quad \Rightarrow 100n + 6 = O(n)$

$n \geq 5, \quad 10n^2 + 4n + 2 \leq 11n^2 \Rightarrow 10n^2 + 4n + 2 = O(n^2)$

$n \geq 4, \quad 6*2^n + n^2 \leq 7*2^n \quad \Rightarrow 6*2^n + n^2 = O(2^n)$

$n \geq 2, \quad 3n + 3 \leq 3n^2 \quad \Rightarrow 3n + 3 = O(n^2)$

$n \geq 2, \quad 10n^2 + 4n + 2 \leq 10n^4 \Rightarrow 10n^2 + 4n + 2 = O(n^4)$

$3n+2 \neq O(1)$

  - for any constant c and all n, $n \geq n_0$, $3n + 2 \leq c$ is false.

$10n^2 + 4n + 2 \neq O(n)$

– O(1) means a computing time is a constant.

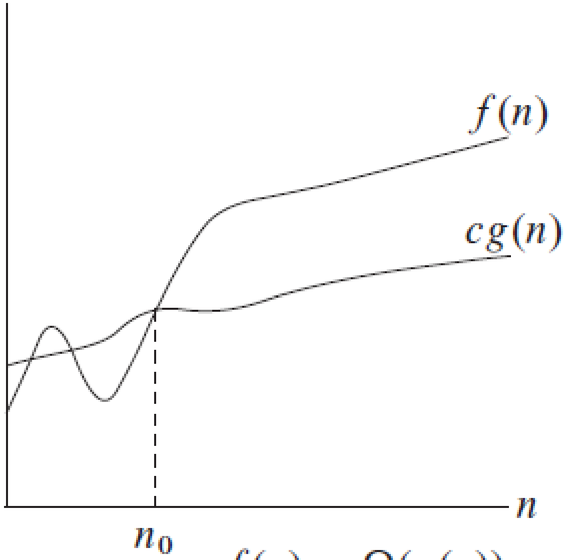
- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

- *$f(n) = O(g(n))$ ('=' means 'is' not 'equal')*
  – $\forall n, n \geq n_0$, $g(n)$ is an upper bound on $f(n)$
  – $g(n)$ should be as small as one can come up with
    - in order for the statement *$f(n) = O(g(n))$* to be informative.

Definition [*Omega*]

$f(n)=\Omega(g(n))$ *iff*

$\exists c, n_0 > 0, s.t.\ f(n) \geq cg(n)\ \ \forall n, n \geq n_0$



$f(n) = \Omega(g(n))$

- Examples

$n \geq 1, \quad 3n + 2 \geq 3n \quad \Rightarrow 3n + 2 = \Omega(n)$

$n \geq 1, \quad 3n + 3 \geq 3n \quad \Rightarrow 3n + 3 = \Omega(n)$

$n \geq 1, \quad 100n + 6 \geq 100n \quad \Rightarrow 100n + 6 = \Omega(n)$

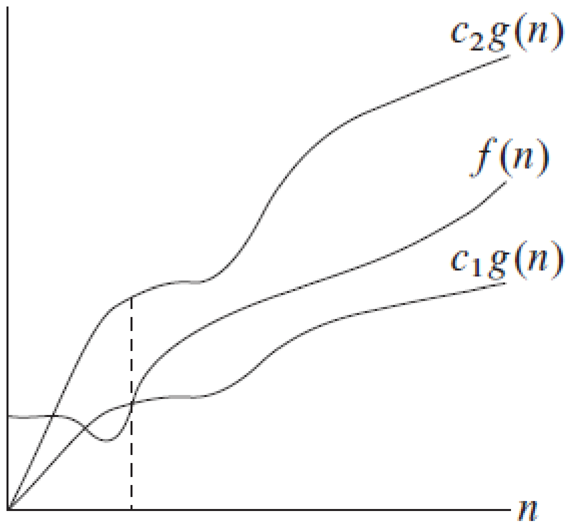
$n \geq 1, \quad 10n^2 + 4n + 2 \geq n^2 \Rightarrow 10n^2 + 4n + 2 = \Omega(n^2)$

$n \geq 1, \quad 6*2^n + n^2 \geq 2^n \quad \Rightarrow 6*2^n + n^2 = \Omega(2^n)$

- $f(n) = \Omega(g(n))$
  - for all n, $n \geq n_0$, $g(n)$ is a lower bound on $f(n)$
  - $g(n)$ should be as large as one can come up with

Definition [*Theta*]

$f(n)=\Theta(g(n))$ iff
$\exists c_1, c_2, n_0 > 0$, s.t. $c_1 g(n) \leq f(n) \leq c_2 g(n) \;\; \forall n, n \geq n_0$



$f(n) = \Theta(g(n))$

- Examples

  $n \geq 2, \quad 3n \leq 3n + 2 \leq 4n \quad \Rightarrow 3n + 2 = \Theta(n),$

  where $c_1 = 3$, $c_2 = 4$, $n_0 = 2$

  $3n + 3 = \Theta(n)$

  $10n^2 + 4n + 2 = \Theta(n^2)$

  $6*2^n + n^2 = \Theta(2^n)$

  $10*\log n + 4 = \Theta(\log n)$

  $2*n*m + 2*n + 1 = \Theta(n*m)$

- $f(n) = \Theta(g(n))$
  - $g(n)$ is both an upper and lower bound on $f(n)$

- Coefficients of all of the $g(n)$'s is 1 !!!
  - We do not write $O(3n)$, $\Omega(4n^2)$, $\Theta(32n)$ but $O(n)$, $\Omega(n^2)$, $\Theta(n)$

Theorem 1.2: If $f(n) = a_m n^m + \ldots + a_1 n + a_0$, then $f(n) = O(n^m)$.

Proof: $f(n) \le \sum_{i=0}^{m} |a_i| n^i$

$$\le n^m \sum_{0}^{m} |a_i| n^{i-m}$$

$$\le n^m \sum_{0}^{m} |a_i|, \text{ for } n \ge 1$$

So, $f(n) = O(n^m)$. $\square$

Theorem 1.3: If $f(n) = a_m n^m + \ldots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$.

Theorem 1.4: If $f(n) = a_m n^m + \ldots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Theta(n^m)$.

- Asymptotic complexity (O,Ω,Θ) is determined easily *without determining the exact step count*.
  - Add up asymptotic complexities for each statement(or group of statements)

- **Example**

Statement	Asymptotic complexity
void add(int a[][MAX_SIZE] $\cdots$)	0
{	0
int i, j;	0
for (i=0; i<rows; i++)	$\Theta(rows)$
for (j = 0; j < cols; j++)	$\Theta(rows.cols)$
c[i][j] = a[i][j] + b[i][j];	$\Theta(rows.cols)$
}	0
Total	$\Theta(rows.cols)$

Figure 1.5: Time complexity of matrix addition

1.5.4 Practical Complexities

$\log n$	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

Figure 1.7: Function values

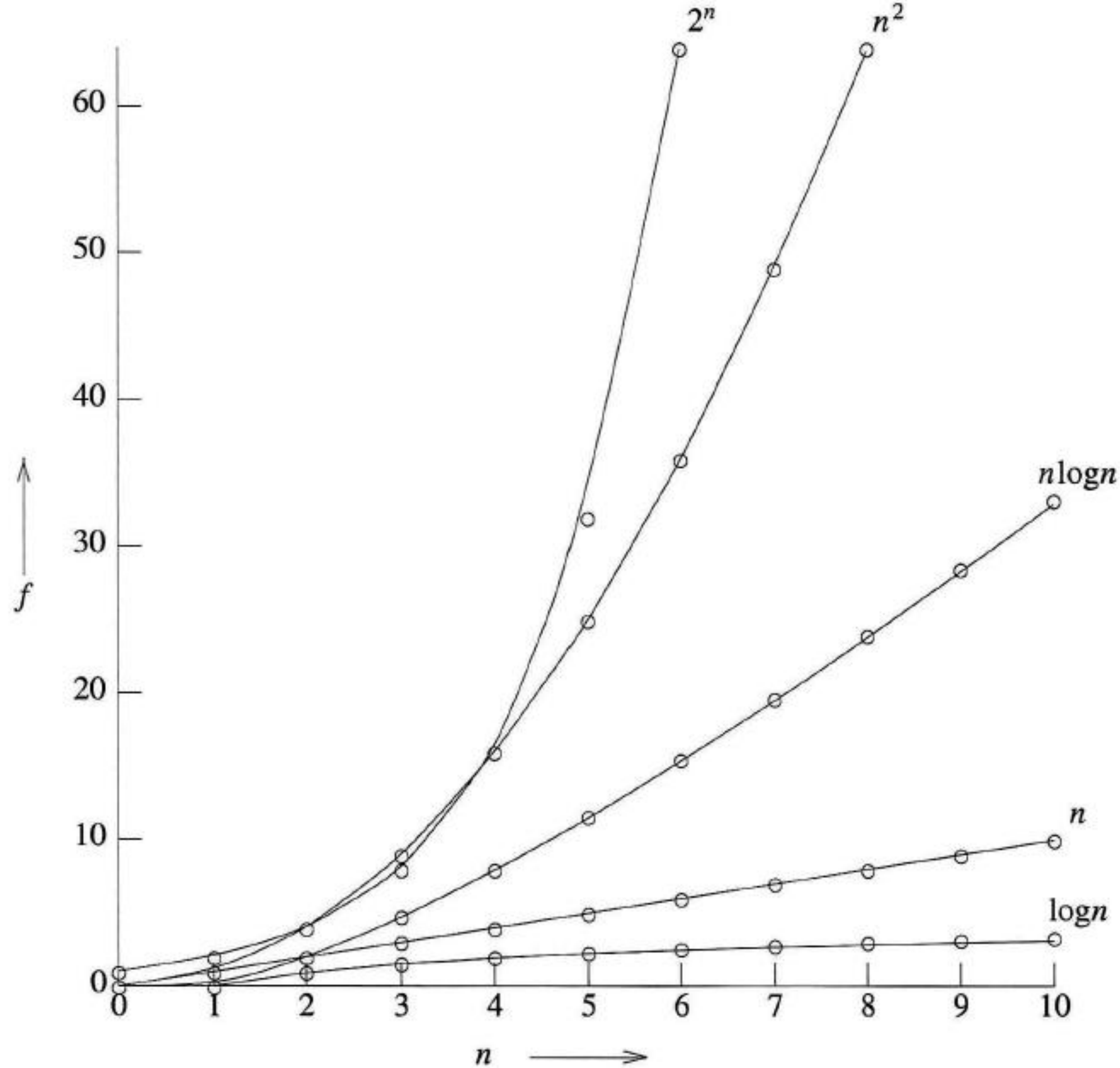


Figure 1.8 Plot of function values

	$f(n)$						
n	n	$n\log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	.01 µs	.03 µs	.1 µs	1 µs	10 µs	10 s	1 µs
20	.02 µs	.09 µs	.4 µs	8 µs	160 µs	2.84 h	1 ms
30	.03 µ	.15 µ	.9 µ	27 µ	810 µ	6.83 d	1 s
40	.04 µs	.21 µs	1.6 µs	64 µs	2.56 ms	121 d	18 m
50	.05 µs	.28 µs	2.5 µs	125 µs	6.25 ms	3.1 y	13 d
100	.10 µs	.66 µs	10 µs	1 ms	100 ms	3171 y	$4*10^{13}$ y
10^3	1 µs	9.96 µs	1 ms	1 s	16.67 m	$3.17*10^{13}$ y	$32*10^{283}$ y
10^4	10 µs	130 µs	100 ms	16.67 m	115.7 d	$3.17*10^{23}$ y	
10^5	100 µs	1.66 ms	10 s	11.57 d	3171 y	$3.17*10^{33}$ y	
10^6	1 ms	19.92 ms	16.67 m	31.71 y	$3.17*10^7$ y	$3.17*10^{43}$ y	

µs = microsecond = 10^{-6} seconds; ms = milliseconds = 10^{-3} seconds
s = seconds; m = minutes; h = hours; d = days; y = years

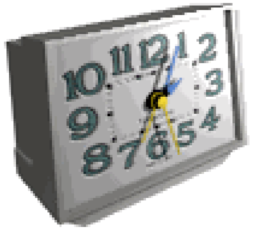
Figure 1.9: Times on a 1-billion-steps-per-second computer

1.6 Performance Measurement

- Measure actual time on an actual computer.

- Data to use for measurement
  - worst-case data
  - best-case data
  - average-case data

- Timing mechanism - clock



- Timing events in C
  - Use clock() or time() function in the C standard library.
  - #include <time.h>

	Method 1	Method 2
Start timing	start = clock();	start = time(NULL);
Stop timing	stop = clock();	stop = time(NULL);
Type returned	clock_t	time_t
Result in seconds	duration = ((double) (stop−start)) / CLOCKS_PER_SEC;	duration = (double) difftime(stop,start);

typedef long clock_t
#define CLOCKS_PER_SEC ((clock_t)1000)
clock(): the return value is in ms (milli-second, 10^{-3} second)

```
#include <stdio.h>
#include <time.h>
#include "selectionSort.h"
#define MAX_SIZE 1001
void main(void)
{
   int i, n, step = 10;
   int a[MAX_SIZE];
   double duration;
   clock_t start;

   /* times for n = 0, 10, ..., 100, 200, ..., 1000 */
   printf("    n     time\n");
   for (n = 0; n <= 1000; n += step)
   {/* get time for size n */

      /* initialize with worst-case data */
      for (i = 0; i < n; i++)
         a[i] = n - i;

      start = clock( );
      sort(a, n);
      duration = ((double) (clock() - start))
                             / CLOCKS_PER_SEC;
      printf("%6d   %f\n", n, duration);
      if (n == 100) step = 100;
   }
}
```

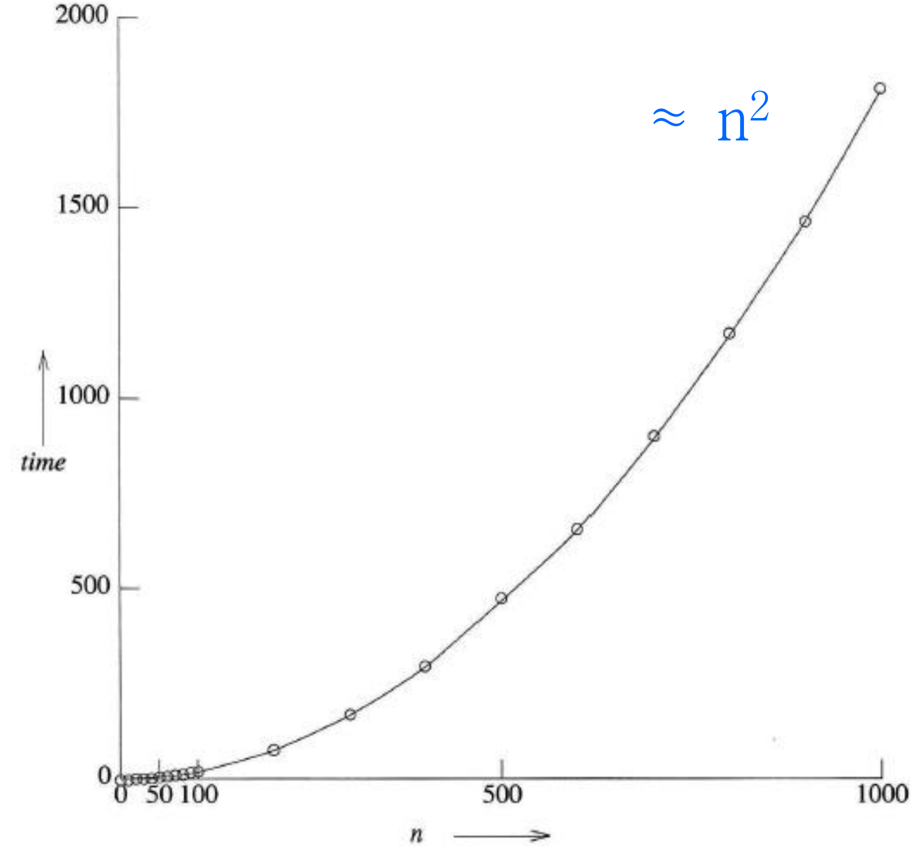
Program 1.24: First timing program for selection sort

```c
#include <stdio.h>
#include <time.h>
#include "selectionSort.h"
#define MAX_SIZE 1001
void main(void)
{
   int i, n, step = 10;
   int a[MAX_SIZE];
   double duration;

   /* times for n = 0, 10, ..., 100, 200, ..., 1000 */
   printf("     n     repetitions     time\n");
   for (n = 0; n <= 1000; n += step)
   {
      /* get time for size n */
      long repetitions = 0;
      clock_t start = clock( );
      do
      {
         repetitions++;

         /* initialize with worst-case data */
         for (i = 0; i < n; i++)
            a[i] = n - i;

         sort(a, n);
      } while (clock( ) - start < 1000);
           /* repeat until enough time has elapsed */

      duration = ((double) (clock() - start))
                                  / CLOCKS_PER_SEC;
      duration /= repetitions;
      printf("%6d   %9d   %f\n", n, repetitions, duration);
      if (n == 100) step = 100;
   }
}
```

Program 1.25: More accurate timing program for selection sort

n	repetitions	time
0	8690714	0.000000
10	2370915	0.000000
20	604948	0.000002
30	329505	0.000003
40	205605	0.000005
50	145353	0.000007
60	110206	0.000009
70	85037	0.000012
80	65751	0.000015
90	54012	0.000019
100	44058	0.000023
200	12582	0.000079
300	5780	0.000173
400	3344	0.000299
500	2096	0.000477
600	1516	0.000660
700	1106	0.000904
800	852	0.001174
900	681	0.001468
1000	550	0.001818

Figure 1.11: Worst-case performance of selection sort (seconds)



Time axis in microseconds

Figure 1.12: Graph of worst-case performance of selection sort