BASIC CONCEPT

1.2 Pointers and Dynamic Memory Allocation

1.3 Algorithm Specification

1.4 Data Abstraction

성능 분석

1.5 Performance Analysis

1.6 Performance Measurement

Evaluating Programs

– …

Criterias:

– Does the program meet the original specifications of the task?
– Does it work correctly?
– Does the program contain documentation that shows how to use it and how it works?
– Does the program effectively use functions to create logical units?
– Is the program's code readable?
– Does the program efficiently use primary and secondary **storage**?
– Is the program's **running time** acceptable for the task?

Performance Evaluation

Performance Analysis

- Focuses on obtaining estimates of time and space that are machine independent
- Known as complexity theory

Performance Measurement

- Machine dependent running times

Complexity: Space and Time

Space complexity

- **The amount of memory** that a program needs to run to completion.

Time complexity

- **The amount of computation time** that a program needs to run to completion.

1.5.1 SPACE COMPLEXITY

Space Complexity

The space needed by a program :

(1) Fixed Space Requirements (c) 고정적으로 필요한 메모리

- Independent on the number and size of the program's inputs and outputs

  Input, output에 대해 독립적임

  코드 영역 변수들
- Space for instruction(code), simple variables, fixed-size structured variables, and constants

가변적으로 필요한 메모리

(2) Variable Space Requirements ($S_P(I)$)

Input의 data set

- Depend on the characteristics of particular instance *I* of the program;

  ex) 배열을 Sorting
- The number, size, and values of the inputs and outputs
- The additional space required when a function uses recursion

  재귀

Space Complexity

Total space requirement **S(P)** of any program:

$$S(P) = c + \mathbf{S_P(I)}$$

Usually concerned with only …

– When we want to compare the space complexity of several programs

Ex 1.6 → $S_{abc}(I) = \ldots$

*** Program 1.10:** Simple arithmetic function (p.23)

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

고정적으로 필요한 메모리 : 지역변수 a, b, c
+ 코드영역

Ex 1.7 → $S_{sum}(I) = ?$

- In Pascal: …
- In C: ..

***Program 1.11:** Iterative function for summing a list of numbers (p.24)*

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

C 기준으로 가변 메모리 X

Ex 1.8 → $S_{rsum}(I) = ?$

재귀(o)

***Program 1.12:** Recursive function for summing a list of numbers*

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

Ex 1.8 → $S_{rsum}(I) = ?$

***Program 1.12:** Recursive function for summing a list of numbers*

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

For each recursive call, compiler must save the parameters, local variables, the return address for each recursive call

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 1000 numbers? input에 따라 필요한 메모리 크기가 달라짐

12000 byte

BASIC CONCEPT

1.2 Pointers and Dynamic Memory Allocation

1.3 Algorithm Specification

1.4 Data Abstraction

1.5 Performance Analysis

- Space Complexity
- **Time Complexity**
- Asymptotic Notation
- Practical Complexities

1.6 Performance Measurement

Time $T(P)$ taken by a program P:

$T(P) = C + \mathbf{T_P(I)}$

- C: compile time (constant)
- $T_P(I)$: run (or execution) time

Compile time

- Fixed
- Independent of instance characteristics (Input과 독립적)

Determining T_p is not an easy task

- It requires a detailed knowledge of the compiler's attributes

Ex)

- Suppose we have a simple program that adds and subtracts numbers:
- $T_P(n) = c_a ADD(n) + c_s SUB(n) + c_l LDA(n) + c_{st} STA(n)$
  - n: instance characteristic
  - c_a, c_s , c_l, c_{st} : constants (time needed to perform each operation)

Alternative:

- Count # of operations the program performs
- Machine-independent estimate, but we must know how to divide the program into distinct steps

Def. : **program step**

A syntactically or semantically meaningful <u>program segment</u> whose execution time is independent of the instance characteristics

Ex)

Each executable statement is counted as one step :

```
a = 2;                                              // 1 step
a = a + b + b * c + (a + b - c) / (a + b) + 4.0;    // 1 step
```

여러 연산이 있음에도 1 step

(=문장단위)

How to count program steps?

1. Using a global **variable**, *count*
2. Using a **tabular** method

```
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

Number of steps? : …

$2n+3$

```
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;
}
```

i가 한번더 (하나씩 마지막)

Program 1.13: Program 1.11 with count statements

Ex 1.10 [Recursive summing]

```
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

rsum(0)   rsum(1)(2)(3)

2 + 2x3
(n)

=2n+2

Number of steps? …

[Ex.1.12] Tabular method

(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 → 개하고 사용이 종료되기 때문에 세임
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

[Ex.1.13]

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			$2n+2$

※ $2n + 3$ (iterative) > $2n + 2$ (recursive)

[Ex.1.14]

Statement	s/e	Frequency	Total Steps
void add(int a[][MAX_SIZE] $\cdots$)	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			$\boxed{2rows \cdot cols + 2rows+1}$

더듬고
마지막 for
(탈출할때)

마찬가지로
for문 탈출까지

Figure 1.4: Step count table for matrix addition

BASIC CONCEPT

1.2 Pointers and Dynamic Memory Allocation

1.3 Algorithm Specification

1.4 Data Abstraction

1.5 Performance Analysis

- Space Complexity
- Time Complexity
- **Asymptotic Notation**
- Practical Complexities

1.6 Performance Measurement

1.5.3 ASYMPTOTIC NOTATION (O, Ω, Θ)

Motivation to determine step counts :

- To compare the time complexities of two programs for the same function
- To predict the growth in run time <u>as the instance characteristics change</u>

Determining the exact step count (either worst case or average)

- Exceedingly difficult task for most of the programs
- The notion of a step is itself inexact
- Not very useful for comparative purposes

Asymptotic complexity

- Provides meaningful(but inexact) statements about the time and space complexities of a program
- Determined quite easily without determining the exact step count

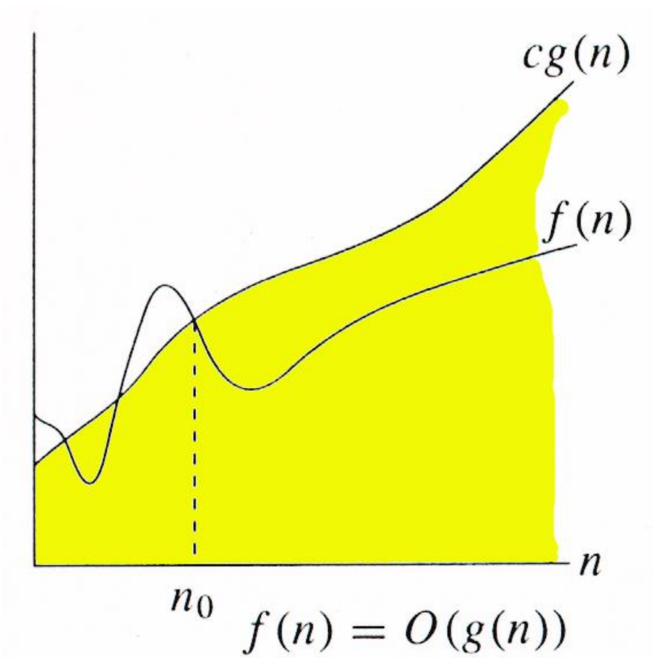
Notations: O, Ω, Θ

- O (Big “oh”): Upper bound 최대 이정도 걸린다
- Ω (Omega): Lower bound 최소 이정도는 걸린다
- Θ (Theta): Upper and lower bound

Def.) [Big “oh”] $f(n) = \mathbf{O}(\boxed{g(n)})$

iff there exist positive constants c and n_0 such that

$f(n) \leq c\boxed{g(n)}$ for all n, $n \geq n_0$.



$\forall n, n \geq n_0,$

$g(n)$ is an upper bound on the value of $f(n)$

상수 다 떼고 식으로 표현

5 10 3 8 1 6

5 10 15 20 25 30

9 4 2

35 40 45 50 55

Def.) [Big "oh"] $f(n) = O(g(n))$
iff there exist positive constants c and n_0 such that
$f(n) \le cg(n)$ for all n, $n \ge n_0$.

Ex. 1.15)

- $3n+2=O(n)$ /* $3n+2 \le 4n$ for all $n \ge 2$ */
- $3n+3=O(n)$ /* $3n+3 \le 4n$ for all $n \ge 3$ */
- $100n+6=O(n)$ /* $100n+6 \le 101n$ for all $n \ge 10$ */
- $10n^2+4n+2=O(n^2)$ /* $10n^2+4n+2 \le 11n^2$ for all $n \ge 5$ */
- $6*2^n+n^2=O(2^n)$ /* $6*2^n+n^2 \le 7*2^n$ for all $n \ge 4$ */

Theorem 1.2:

if $f(n) = a_m n^m + \ldots + a_1 n + a_0$,

then $f(n) = O(n^m)$.

```
void swap(int *x, int *y)
{/* both parameters are po
   int temp = *x;    /* dec
                 to it the
   *x = *y; /* stores what
               where x poi
   *y = temp; /* places th
                 pointed t
}
```

O(1) is called constant

$O(n^2)$: quadratic

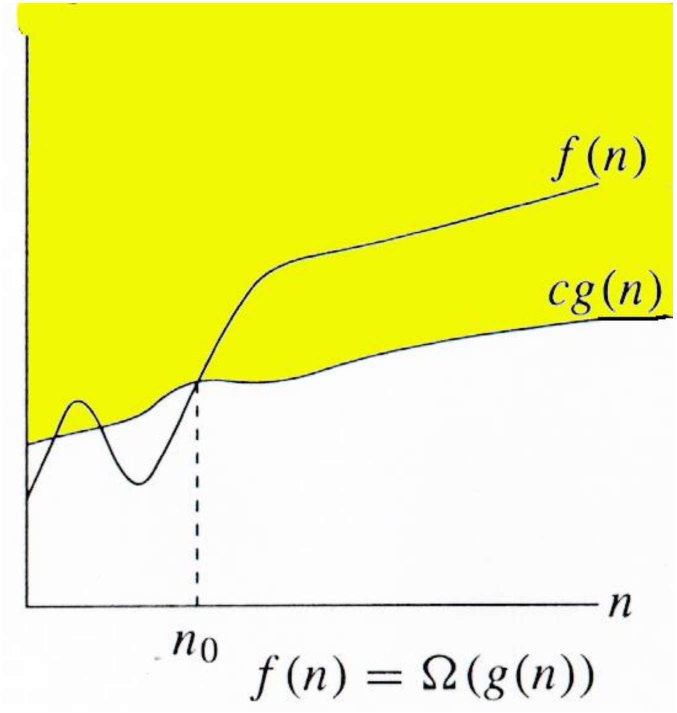
$O(n^3)$: cubic

$O(2^n)$: exponential

Computing Times :

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

Def.) [Omega] $f(n) = \Omega(g(n))$
iff $\exists c, n_0 > 0$, s.t. $f(n) \geq cg(n)$ for $\forall n, n \geq n_0$



For all $n, n \geq n_0$,
$g(n)$ is a **lower bound** on the value of $f(n)$

Def.) [Omega] $f(n) = \Omega(g(n))$

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

Ex 1.16)

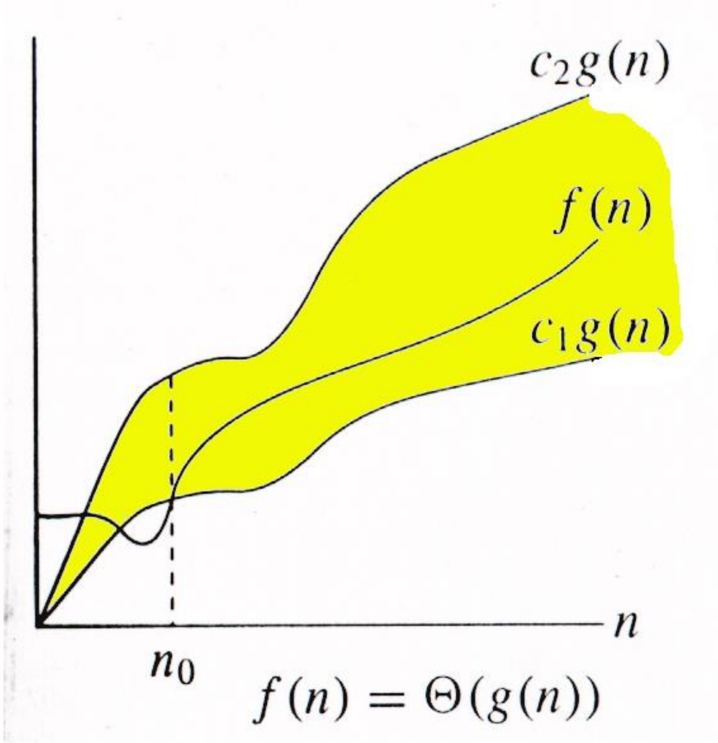
$n \geq 1, \quad 3n + 2 \geq 3n \quad \Rightarrow 3n + 2 = \Omega(n)$

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

$n \geq 1, \quad 10n^2 + 4n + 2 \geq n^2 \quad \Rightarrow 100n^2 + 4n + 2 = \Omega(n^2)$

Def.) [Theta] $f(n) = \mathbf{\Theta(g(n))}$

iff $\exists c_1, c_2, n_0 > 0$, s.t. $c_1 g(n) \le f(n) \le c_2 g(n)$ for $\forall n, n \ge n_0$



g(n): both an upper and lower bound on the value of f(n)

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

Ex. 1.17)

- $3n+2 = \Theta(n)$
  $\rightarrow n \geq 2,\ 3n \leq 3n+2 \leq 4n$
- $10n^2+4n+2 = \Theta(n^2)$
- $6*2^n+n^2 = \Theta(2^n)$

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 $f(n) = \Theta(n^m)$.

O (Big "oh"): Upper bound

Ω (Omega): Lower bound

Θ (Theta): Upper and lower bound

Ex 1.18 [Complexity of matrix addition]:

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

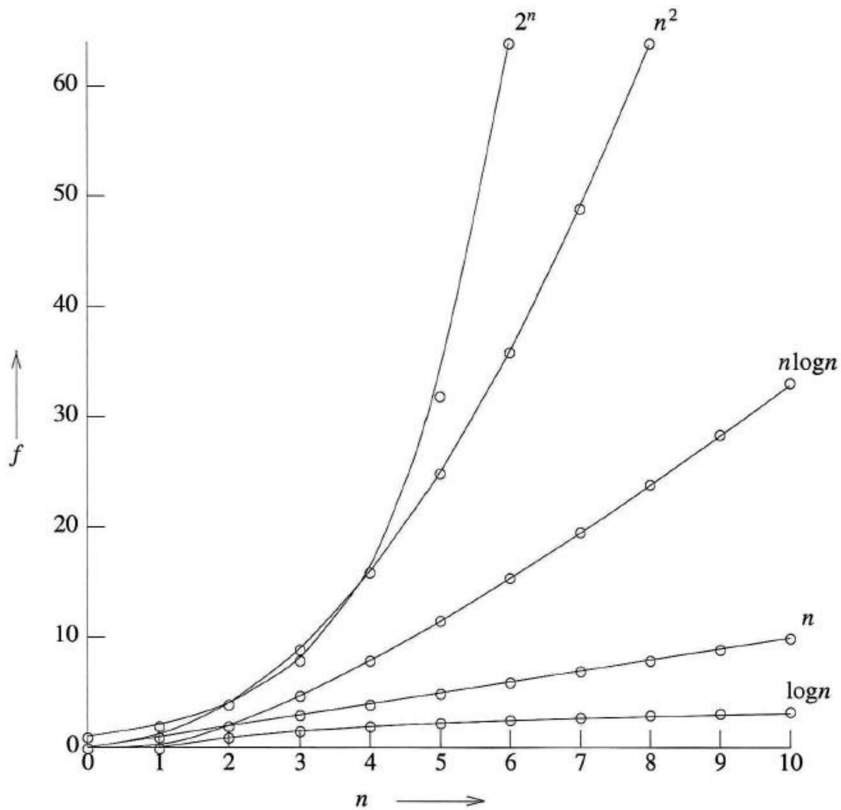


Figure 1.8 Plot of function values

	$f(n)$						
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

BASIC CONCEPT

1.2 Pointers and Dynamic Memory Allocation

1.3 Algorithm Specification

1.4 Data Abstraction

1.5 Performance Analysis

1.6 Performance Measurement

1.6.1 Clocking

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);

Ex 1.22 [Worst-case performance of selection sort]:

```
#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

```
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, duratio
      if (n == 100) step = 100;
   }
}
```

Program 1.25: More accurate timing program for selection sort selection sort 알고리즘 시간복잡도 ⇒ $O(n^2)$

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)

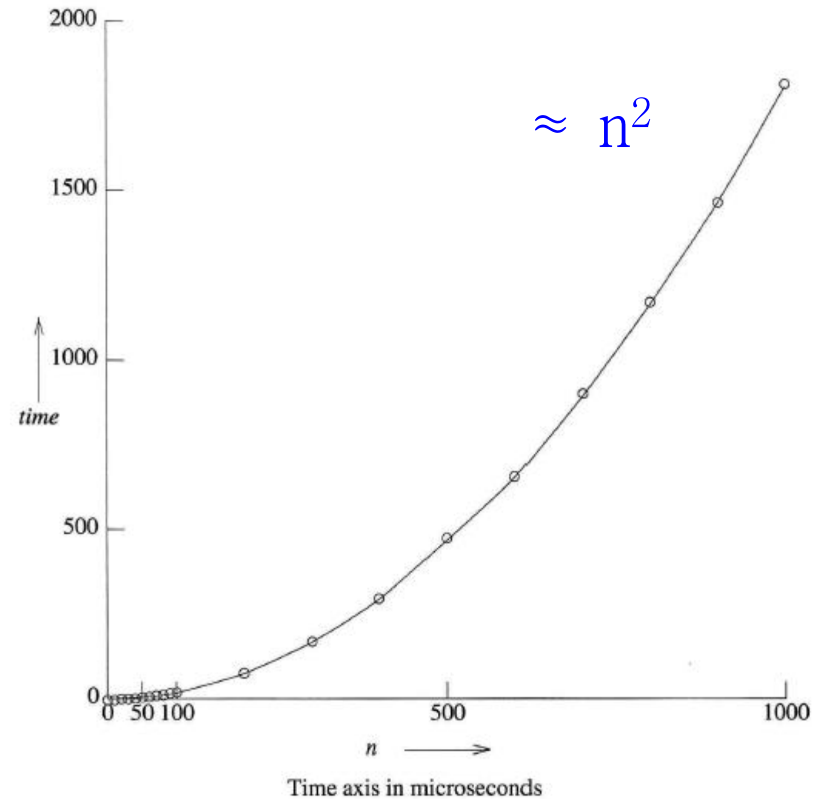


Figure 1.12: Graph of worst-case performance of selection sort

```
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
```

```
void sort(int list[], int n)
{
    int i, j, min, temp;
    for(i = 0; i < n-1;  i++) {
            min = i;
            for (j = i + 1; j < n; j++)
                    if(list[j] < list[min])
                            min = j;
            SWAP(list[i], list[min], temp);
    }
}
```

Program 1.4: Selection sort

```
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, duratio
      if (n == 100) step = 100;
   }
}
```

Program 1.25: More accurate timing program for selection sort

BASIC CONCEPT

1.2 Pointers and Dynamic Memory Allocation

1.3 Algorithm Specification

1.4 Data Abstraction

1.5 Performance Analysis

1.6 Performance Measurement