Basic Concepts

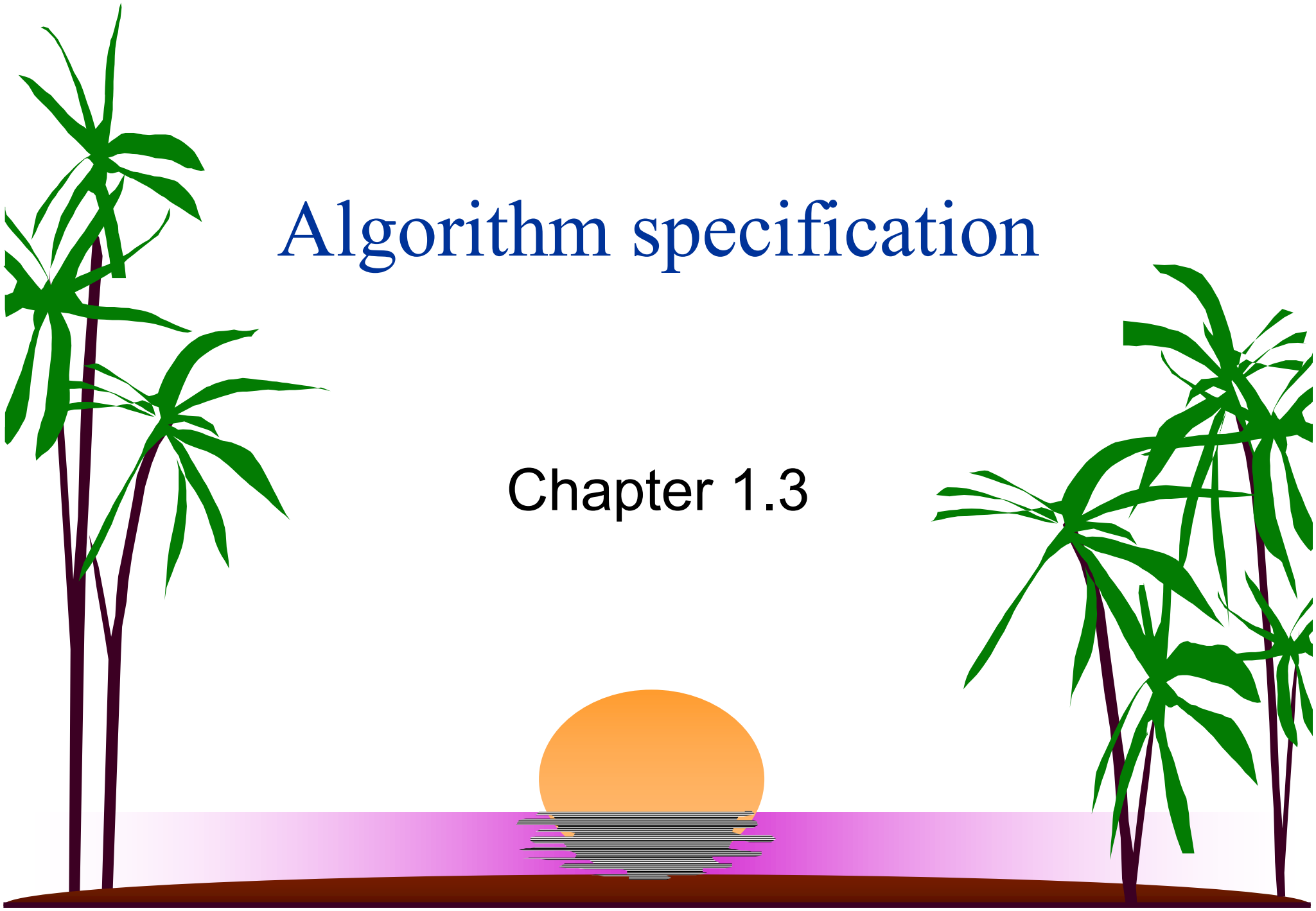
Chapter 1

강의 스케쥴

1. 1장: Binary search, performance analysis
2. 2장: Arrays and structures
3. 2장: Polynomials, Sparse matrices
4. 3장: Stacks and queues
5. 4장: Linked list
6. 4장: Polynomials, List operations
7. 5장: Binary Trees
8. 5장: Heaps, Binary Search Trees
9. 6장: Graph
10. 6장 : Minimum Cost Spanning Trees
11. 7장: Insertion Sort, Quick Sort
12. 7장: Merge Sort, Heap Sort
13. 8장: Hashing

Algorithm specification

Chapter 1.3



Algorithm specification

Definition An algorithm is a finite set of instructions that, if followed, accomplishes a particular task

All algorithm must satisfy the following criteria:

1) Input
2) Output
3) Definiteness
4) Effectiveness

Example: Search problem

Task: Find a specific value n among N distinct numbers

Input. : n, N numbers

Output: 1 if n exists among N numbers

0 otherwise

How?

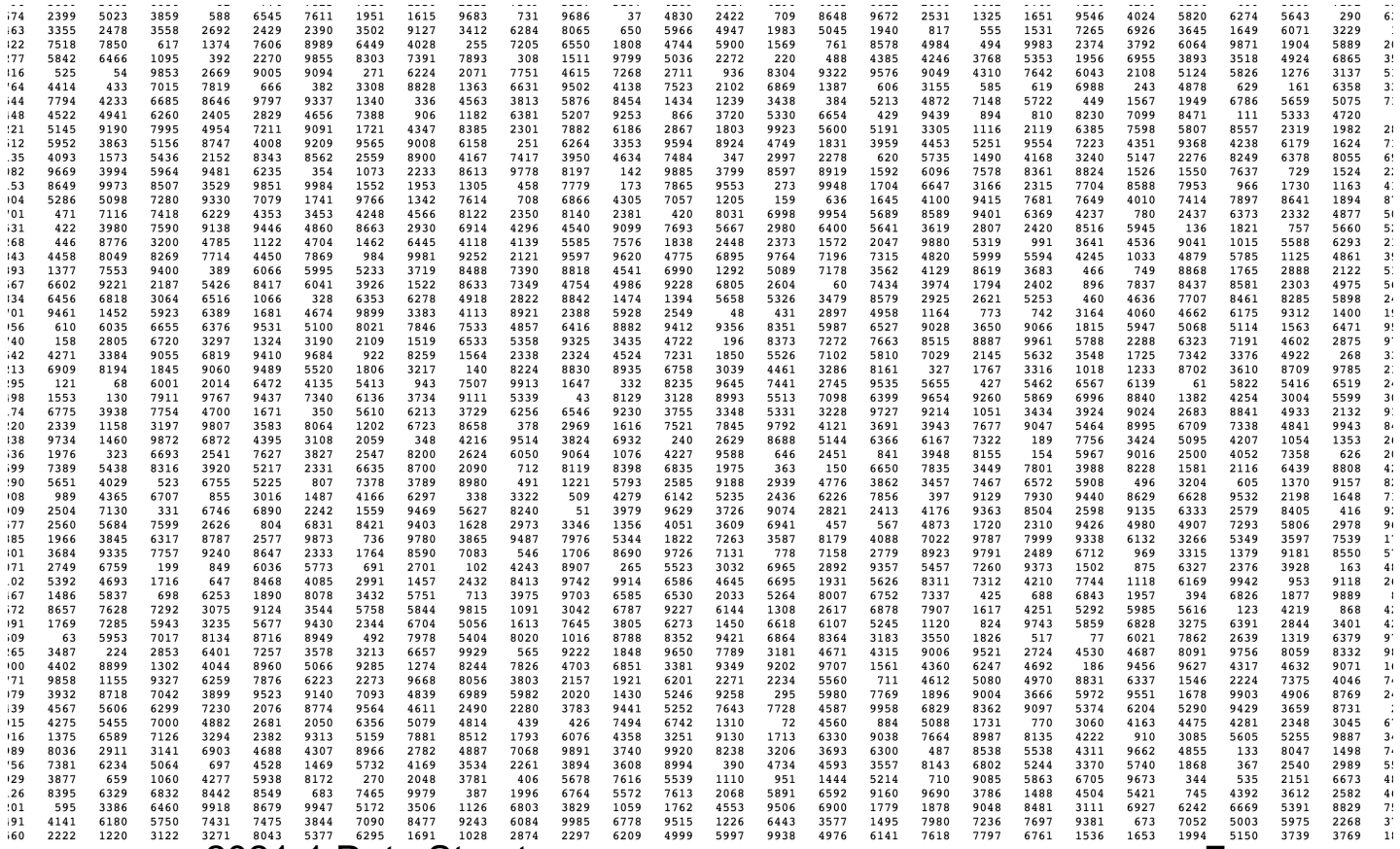
What kind of data types?

12 5 10 21 33 7 8

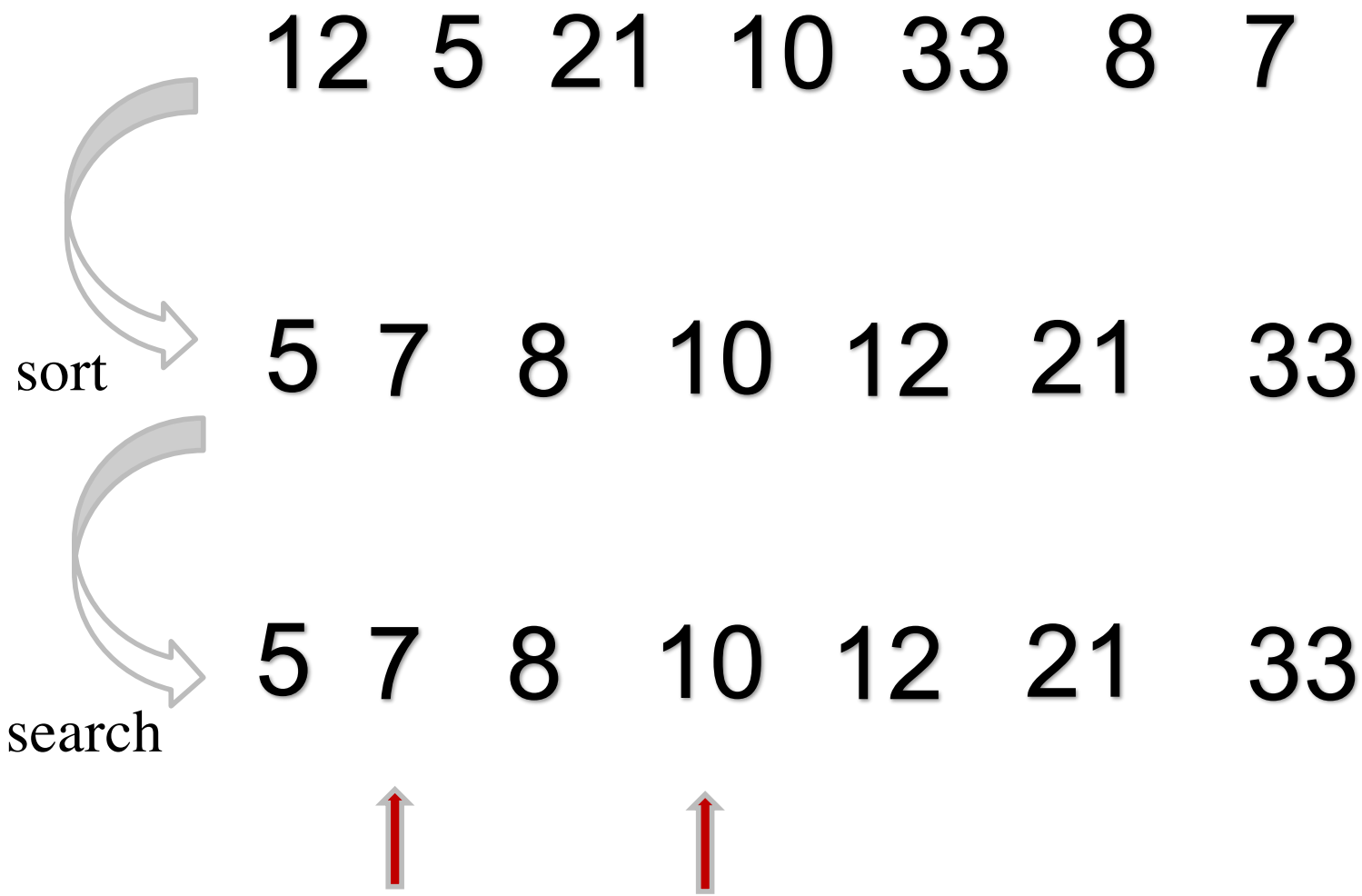
How to search?

12 5 21 10 33 8 7

Millions of numbers



Better methods?



(1) Sort: Selection sort

- **Task:** From those integers that are currently unsorted, find the smallest and place it next in the sorted list.
- **Input:** n integers are stored in an array list[i], 0 <= i < n.
- **Output:** a sorted list[i], 0 <= i < n.
- **Specification:**

```
for (i=0; i<n; i++) {
    Examine list[i] to list[n-1] and
        suppose that the smallest integer is at list[min];
    Interchange list[i] and list[min];
}
```

(2) Search: Binary search

- **Task:**
  - Assume that we have n>=1 distinct integers that are already sorted and stored in the array *list*.
  - For a given integer m, we want to know whether there is *i* such that *list[i]* = *m.*
- **Input:** a sorted array list[i], 0 <= i < n, an integer m
- **Output:** i if list[i] = m, -1 if there is no such i
- **Specification:**

  let left = 0, right = n-1, middle = (left + right)/2

  1) If m < list[middle], then right = middle -1
  2) If m = list[middle], then return middle
  3) If m > list[middle], then left = middle +1
  4) Recalculate middle and repeat 1) ~3) while there is more integers to check

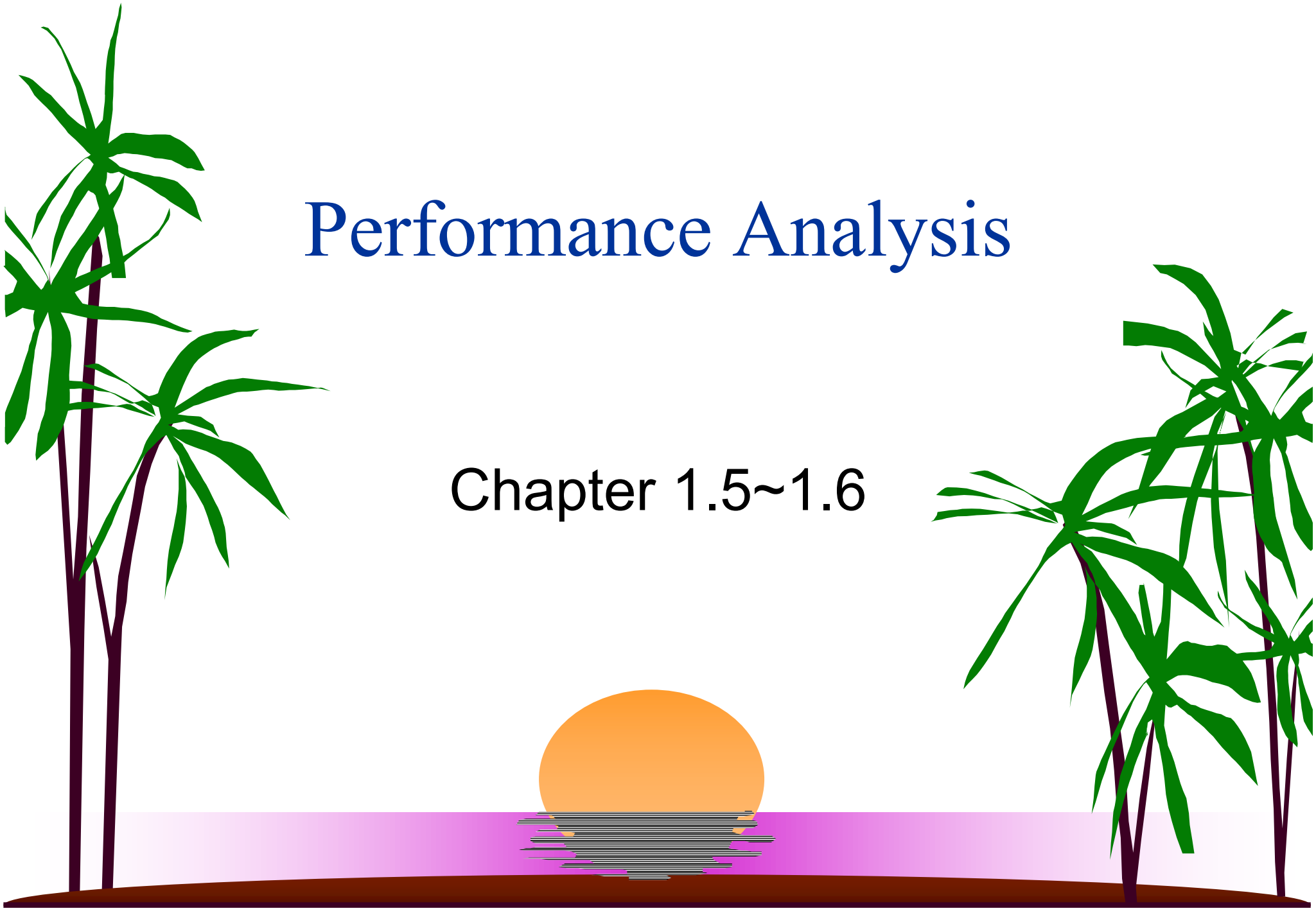
Implementation: program structure

```
#define LIST_SIZE 500
int list[LIST_SIZE];
int num_cnt = 0;

void main(int argc, char* argv[]){
  // read data list[] and num_cnt
     …
  selection_sort(num_cnt);
  // write sorted list
     …
  fprintf(stdin, “enter the number to search: “);
  fscanf(stdin, “%d”, &searchnum);
  result = binsearch(searchnum, 0, num_cnt);
  if(result >=0)
       fprintf(stdout, ”number %d is %dth number in the list\n”, searchnum, result);
  else fprintf(stdout, “number %d is not in the list\n”, searchnum);
  return;
}
```

Performance Analysis

Chapter 1.5~1.6



Performance Analysis

(성능/효율 분석)

- One of the goals of this lecture is to develop your skills for making evaluative judgments about programs.
  - Does the program efficiently use primary and secondary storage?
  - Is the program's running time acceptable for the task?

Which one has better performance?

```
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

```
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

Some Uses Of Performance Analysis

- Determine practicality of algorithm
- Predict run time on large instance
- Compare two algorithms that have different asymptotic complexity (점근적 복잡도)
  - e.g., O(n) and $O(n^2)$

Program Complexity

- Program complexity
  - space complexity : the amount of memory it needs
  - time complexity : the amount of computer time it needs

- Performance evaluation phases
  - performance analysis : a priori estimates (연역적 평가)
  - performance measurement : a posteriori testing (귀납적 테스팅)

Space Complexity

- Fixed part : independent of the characteristics of the inputs and outputs
  - Instruction space, space for simple variables, fixed-size structured variables, constants
- Variable part : depends on the instance characteristics
- S(P) = c+Sp(n)
  - S(P):space requirement of program P
  - c : constant (for fixed space requirements)
  - n : Instance characteristics (ex: I/O size, number)

Space Complexity : Example(1)

```
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

```
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

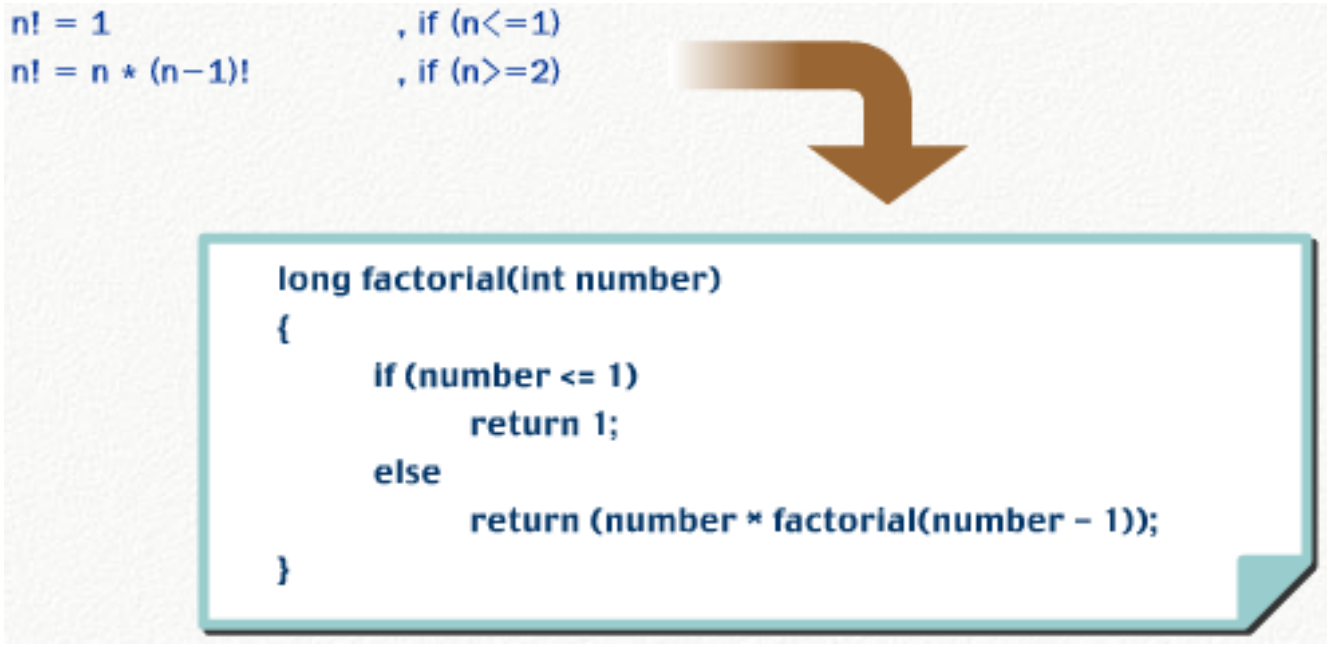
Space Complexity : Example(2)

```
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

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

Exercise 1


n! = 1 , if (n<=1)
n! = n * (n−1)! , if (n>=2)
long factorial(int number)
{
if (number <= 1)
return 1;
else
return (number * factorial(number – 1));
}

Time complexity

- $T(P) = c + T_p(n)$
  - T(P) : time taken by program P
  - c : compile time
  - T_p : run time
  - n : instance characteristics

Program Steps

- We could count the number of operations the program performs to analyze the time requirements
  - Machine-independent estimate
  - We must know how to divide the program into distinct steps
- Definition: *A program step*
  - A meaningful segment of a program whose execution time is independent of the instance characteristics

Determining the number of steps : examples

Only need to worry about executable statements

```
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

Number of steps?

Determining the number of steps : examples

```
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

Number of steps?

Determining the number of steps : examples

```
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

Number of steps?

Determining the number of steps : step count table

statement	steps	freq	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

Determining the number of steps : step count table

statement	steps	freq	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

Exercise 2

- Write the step count table

statement	steps	freq	total steps

void printMatrix(int matrix[] [MAX-SIZE],
int rows, int cols)
{
int i, j;
for (i = 0; i < rows; i++) {
for (j = 0; j < cols; j++)
printf("%d",matrix[i] [j]);
printf (" \n");
}
}			

total			

Exercise 3

- Write the step count table

statement	steps	freq	total steps

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];
}
}			

total			

Asymptotic notation (O, Ω, Θ)

- Determining the exact step count is very difficult
- The notion of a step is inexact – not so useful for comparative purposes.
  - 100n+10 vs. 3n+3
  - 80n, 85n, 75n

We need to be able to make meaningful statements about the time and space complexities of a program.

Definition [Big "oh"]

- f(n)=O(g(n)) iff there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all n, $n \geq n_0$
- g(n) is an upper bound
- e.g:

  $3n+3 = O(n)$ as $3n+3 \leq 4n$ for all $n \geq 3$

  $3n+3 = O(n^2)$ as $3n+3 \leq 3n^2$ for $n \geq 2$

  $10n^2 + 4n + 2 =$

  $6*2^n + n^2 =$
- $O(n)$, $O(n^2)$, $O(n^3)$, $O(n\log n)$

Definition [Omega]

- $f(n)=\Omega(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all n, $n \geq n_0$

- g(n) is a lower bound

  e.g: $3n+2=\Omega(n)$

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

  $6*2^n + n^2 =$

Definition [Theta]

- $f(n)=\Theta(g(n))$ iff there exist positive constants c_1, c_2 and n_0 such that

  $c_1 g(n) \le f(n) \le c_2 g(n)$ for all n, $n \ge n_0$

- $g(n)$ is both an upper and lower bound

  e.g: $3n+2=\Theta(n)$

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

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

  2*n*m + 2*n + 1 =

Asymptotic Complexity: Examples

Statement	Asymptotic complexity
void add(int a[][MAX_SIZE] ···)	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	$\Theta(rows.cols)$

Practical Complexities

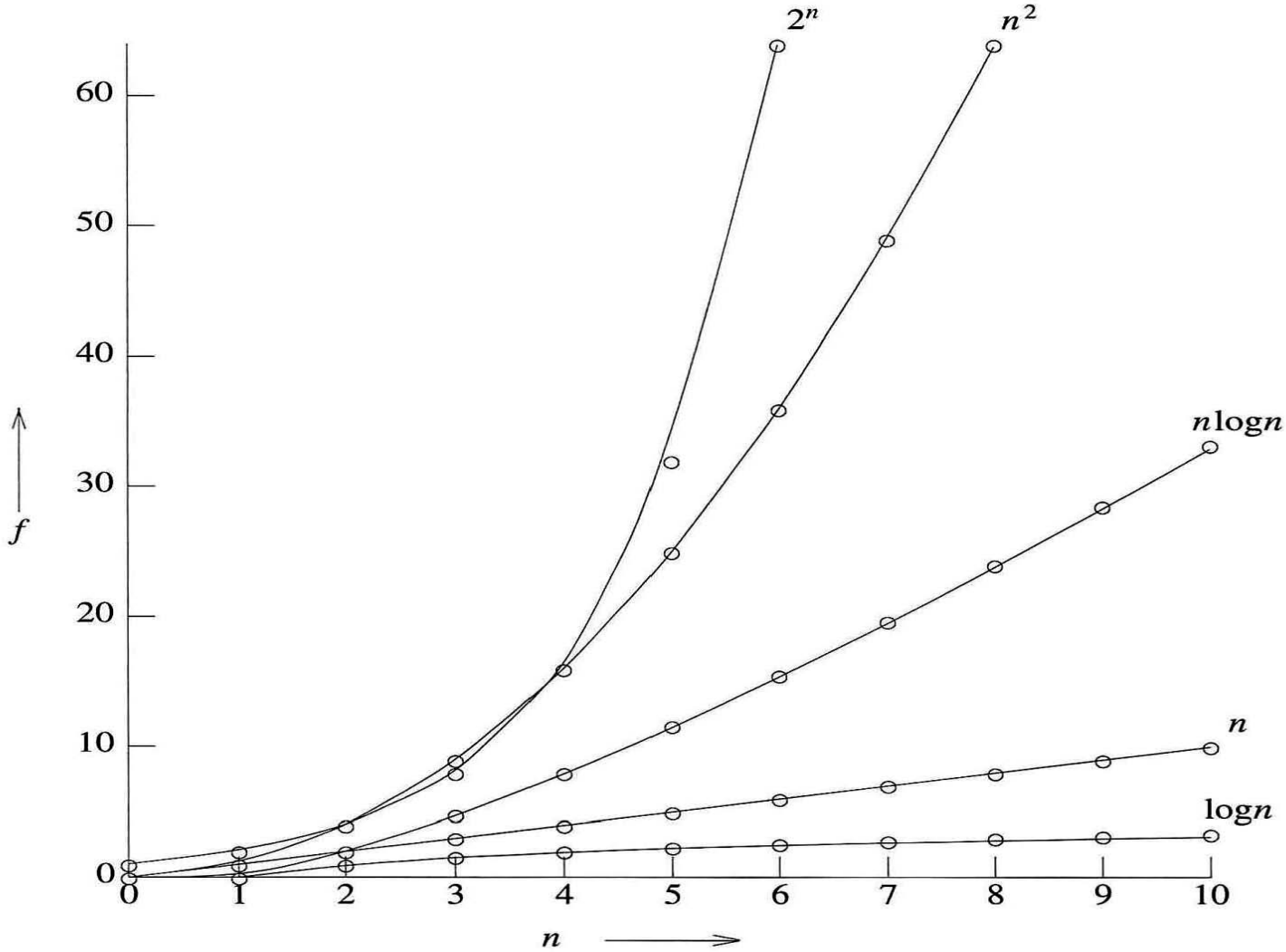


Figure 1.8 Plot of function values

Performance Measurement

- Measure actual time on an actual computer.
- Clocking

Performance Measurement Needs

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

- Timing mechanism --- clock

Event timing In C

- Use clock() function 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);

Event timing in C : a sample program

```
#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

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

Bad Way To Time

```
do {
        counter++;
        start = clock();
        doSomething();
        stop = clock();
       elapsedTime += stop - start;
    } while (elapsedTime < 1000)
```

Event timing in C : more accurate timing

```
#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

Summary

- We learned
  - Algorithm specification & basic programming skills
    - Binary search example
  - Performance analysis
    - Asymptotic notation
  - Performance measurement
    - Clocking
- Those will be used throughout this lecture

Next Time

1. 1장: Binary search, performance analysis
2. 2장: Arrays and structures
3. 2장: Polynomials, Sparse matrices
4. 3장: Stacks and queues
5. 4장: Linked list
6. 4장: Polynomials, List operations
7. 5장: Binary Trees
8. 5장: Heaps, Binary Search Trees
9. 6장: Graph
10. 6장 : Minimum Cost Spanning Trees
11. 7장: Insertion Sort, Quick Sort
12. 7장: Merge Sort, Heap Sort
13. 8장: Hashing