

**POSTGRADUATE DEPARTMENT OF COMPUTER
APPLICATIONS,
GOVERNMENT ARTS COLLEGE(AUTONOMOUS),
COIMBATORE 641018.**

DATA STRUCTURES AND ALGORITHMS

The contents in this E material are from

**Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
“Fundamentals of Data Structures in C”,
Computer Science Press, 1992.**

UNIT 1

FACULTY

Dr.R.A.ROSELINE M.Sc.M.Phil.,Ph.D,
Associate Professor and Head,
Postgraduate Department of Computer Applications,
Government Arts College(Autonomous),
Coimbatore 641018.

Introduction

- The methods of algorithm design form one of the core practical technologies of computer science.
- The main aim of this lecture is to familiarize the student with the framework we shall use through the course about the design and analysis of algorithms.
- We start with a discussion of the algorithms needed to solve computational problems. The problem of *sorting* is used as a running example.
- We introduce a *pseudocode* to show how we shall specify the algorithms.

Algorithms

- The word *algorithm* comes from the name of a Persian mathematician Abu Ja'far Mohammed ibn-i Musa al Khowarizmi.
- In computer science, this word refers to a special method useable by a computer for solution of a problem. The statement of the problem specifies in general terms the desired input/output relationship.
- For example, sorting a given sequence of numbers into nondecreasing order provides fertile ground for introducing many standard design techniques and analysis tools.

Analysis of algorithms

Why study algorithms and performance?

- Algorithms help us to understand **scalability**.
- Performance often draws the line between what is feasible and what is impossible.
- Algorithmic mathematics provides a **language** for talking about program behavior.
- The lessons of program performance generalize to other computing resources.
- Speed is fun!

Running Time

- The running time depends on the input: an already sorted sequence is easier to sort.
- Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

Kinds of analyses

Worst-case: (usually)

- $T(n)$ = maximum time of algorithm on any input of size n .

Average-case: (sometimes)

- $T(n)$ = expected time of algorithm over all inputs of size n .
- Need assumption of statistical distribution of inputs.

Best-case:

- Cheat with a slow algorithm that works fast on some input.

Growth of Functions

Although we can sometimes determine the exact running time of an algorithm, the extra precision is not usually worth the effort of computing it.

For large inputs, the multiplicative constants and lower order terms of an exact running time are dominated by the effects of the input size itself.

Measurements

- Criteria
 - Is it correct?
 - Is it readable?
 - ...
- Performance Analysis (machine independent)
 - space complexity: storage requirement
 - time complexity: computing time
- Performance Measurement (machine dependent)

Space Complexity

$$S(P) = C + S_P(I)$$

- Fixed Space Requirements (C)
Independent of the characteristics of the inputs and outputs
 - instruction space
 - space for simple variables, fixed-size structured variable, constants
- Variable Space Requirements ($S_P(I)$)
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

*Program 1.9: Simple arithmetic function (p.19)

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

$$S_{abc}(I) = 0$$

*Program 1.10: Iterative function for summing a list of numbers
(p.20)

```
float sum(float list[ ], int n)
```

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

```
float tempsum = 0;
```

```
int i;
```

```
for (i = 0; i < n; i++)
```

```
tempsum += list [i];
```

```
return tempsum;
```

```
}
```

Recall: pass the address of first element of the array & pass by value

*Program 1.11: Recursive function for summing a list of numbers

(p.20)

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

$$S_{\text{sum}}(l) = S_{\text{sum}}(n) = 6n$$

Assumptions:

*Figure 1.1: Space needed for one recursive call of Program 1.11

(p.21)

Type	Name	Number of bytes
parameter: float	list []	2
parameter: integer	n	2
return address:(used internally)		2(unless a far address)
TOTAL per recursive call		6

Time Complexity

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

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

- Definition

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

A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

- Example

– $abc = a + b + b * c + (a + b) - c$ (regard as 4.0 unit machine independent)

– $abc = a + b + c$

Methods to compute the step count

- Introduce variable count into programs
- Tabular method
 - Determine the total number of steps contributed by each statement
 $\text{step per execution} \times \text{frequency}$
 - 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];
}
```

$2n+2$

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 */
}
```

*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 × 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

The notation we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers

$$\mathbf{N} = \{0, 1, 2, \dots\}$$

O-notation

- For a given function $g(n)$, we denote by $O(g(n))$ the set of functions

$$O(g(n)) = \left\{ \begin{array}{l} f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

- We use O-notation to give an asymptotic upper bound of a function, to within a constant factor.
- $f(n) = O(g(n))$ means that there exists some constant c s.t. $f(n)$ is always $\leq cg(n)$ for large enough n .

Ω -Omega notation

- For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions

$$\Omega(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

- We use Ω -notation to give an asymptotic lower bound on a function, to within a constant factor.
- $f(n) = \Omega(g(n))$ means that there exists some constant c s.t. $f(n)$ is always $\geq cg(n)$ for large enough n .

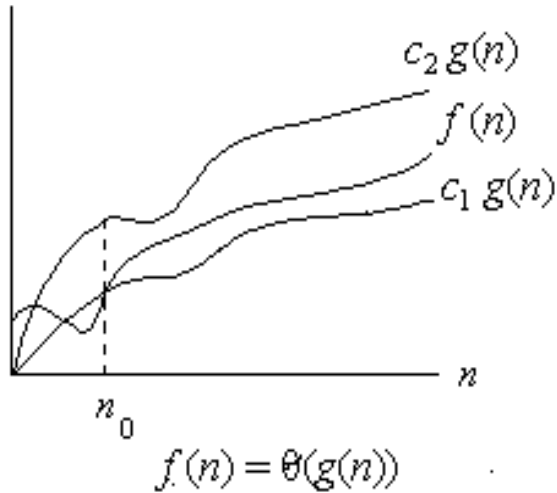
Θ -Theta notation

- For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions

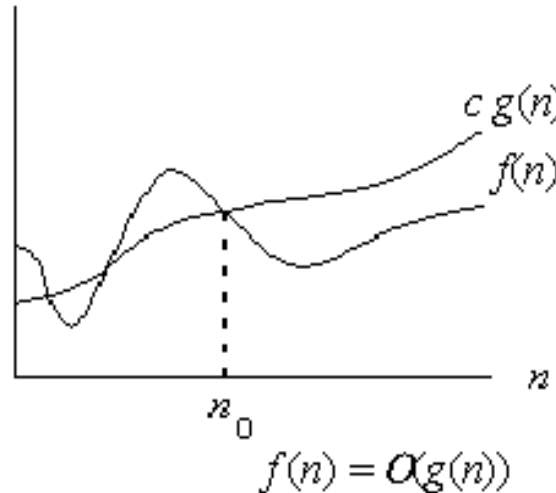
$$\Theta(g(n)) = \left\{ \begin{array}{l} f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ s.t.} \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

- A function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be “sandwiched” between $c_1 g(n)$ and $c_2 g(n)$ or sufficiently large n .
- $f(n) = \Theta(g(n))$ means that there exists some constant c_1 and c_2 s.t. $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for large enough n .

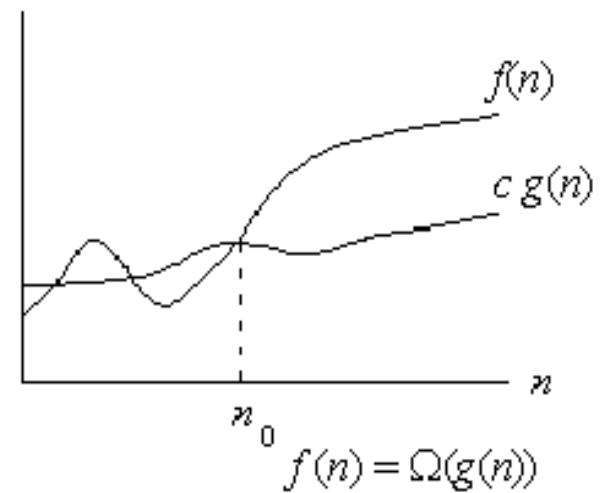
Asymptotic notation



(a)



(b)



(c)

Graphic examples of Θ , O , and Ω .

Example 1.

Show that $f(n) = \frac{1}{2}n^2 - 3n = \Theta(n^2)$

We must find c_1 and c_2 such that

$$c_1n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2n^2$$

Dividing bothsides by n^2 yields

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

For $n_0 \geq 7$, $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

Theorem

- For any two functions $f(n)$ and $g(n)$, we have

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

if and only if

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)).$$

Example 2.

$$f(n) = 3n^2 - 2n + 5 = \Theta(n^2)$$

Because :

$$3n^2 - 2n + 5 = \Omega(n^2)$$

$$3n^2 - 2n + 5 = O(n^2)$$

Example 3.

$$3n^2 - 100n + 6 = O(n^2) \quad \text{since for } c = 3, \quad 3n^2 > 3n^2 - 100n + 6$$

Example 3.

$3n^2 - 100n + 6 = O(n^2)$ since for $c = 3$, $3n^2 > 3n^2 - 100n + 6$

$3n^2 - 100n + 6 = O(n^3)$ since for $c = 1$, $n^3 > 3n^2 - 100n + 6$ when $n > 3$

Example 3.

$3n^2 - 100n + 6 = O(n^2)$ since for $c = 3$, $3n^2 > 3n^2 - 100n + 6$

$3n^2 - 100n + 6 = O(n^3)$ since for $c = 1$, $n^3 > 3n^2 - 100n + 6$ when $n > 3$

$3n^2 - 100n + 6 \neq O(n)$ since for any c , $cn < 3n^2$ when $n > c$

Example 3.

$3n^2 - 100n + 6 = O(n^2)$ since for $c = 3$, $3n^2 > 3n^2 - 100n + 6$

$3n^2 - 100n + 6 = O(n^3)$ since for $c = 1$, $n^3 > 3n^2 - 100n + 6$ when $n > 3$

$3n^2 - 100n + 6 \neq O(n)$ since for any c , $cn < 3n^2$ when $n > c$

$3n^2 - 100n + 6 = \Omega(n^2)$ since for $c = 2$, $2n^2 < 3n^2 - 100n + 6$ when $n > 100$

Example 3.

$3n^2 - 100n + 6 = O(n^2)$ since for $c = 3$, $3n^2 > 3n^2 - 100n + 6$

$3n^2 - 100n + 6 = O(n^3)$ since for $c = 1$, $n^3 > 3n^2 - 100n + 6$ when $n > 3$

$3n^2 - 100n + 6 \neq O(n)$ since for any c , $cn < 3n^2$ when $n > c$

$3n^2 - 100n + 6 = \Omega(n^2)$ since for $c = 2$, $2n^2 < 3n^2 - 100n + 6$ when $n > 100$

$3n^2 - 100n + 6 \neq \Omega(n^3)$ since for $c = 3$, $3n^2 - 100n + 6 < n^3$ when $n > 3$

Example 3.

$3n^2 - 100n + 6 = O(n^2)$ since for $c = 3$, $3n^2 > 3n^2 - 100n + 6$

$3n^2 - 100n + 6 = O(n^3)$ since for $c = 1$, $n^3 > 3n^2 - 100n + 6$ when $n > 3$

$3n^2 - 100n + 6 \neq O(n)$ since for any c , $cn < 3n^2$ when $n > c$

$3n^2 - 100n + 6 = \Omega(n^2)$ since for $c = 2$, $2n^2 < 3n^2 - 100n + 6$ when $n > 100$

$3n^2 - 100n + 6 \neq \Omega(n^3)$ since for $c = 3$, $3n^2 - 100n + 6 < n^3$ when $n > 3$

$3n^2 - 100n + 6 = \Omega(n)$ since for any c , $cn < 3n^2 - 100n + 6$ when $n > 100$

Example 3.

$3n^2 - 100n + 6 = O(n^2)$ since for $c = 3$, $3n^2 > 3n^2 - 100n + 6$

$3n^2 - 100n + 6 = O(n^3)$ since for $c = 1$, $n^3 > 3n^2 - 100n + 6$ when $n > 3$

$3n^2 - 100n + 6 \neq O(n)$ since for any c , $cn < 3n^2$ when $n > c$

$3n^2 - 100n + 6 = \Omega(n^2)$ since for $c = 2$, $2n^2 < 3n^2 - 100n + 6$ when $n > 100$

$3n^2 - 100n + 6 \neq \Omega(n^3)$ since for $c = 3$, $3n^2 - 100n + 6 < n^3$ when $n > 3$

$3n^2 - 100n + 6 = \Omega(n)$ since for any c , $cn < 3n^2 - 100n + 6$ when $n > 100$

$3n^2 - 100n + 6 = \Theta(n^2)$ since both O and Ω apply.

Example 3.

$3n^2 - 100n + 6 = O(n^2)$ since for $c = 3$, $3n^2 > 3n^2 - 100n + 6$

$3n^2 - 100n + 6 = O(n^3)$ since for $c = 1$, $n^3 > 3n^2 - 100n + 6$ when $n > 3$

$3n^2 - 100n + 6 \neq O(n)$ since for any c , $cn < 3n^2$ when $n > c$

$3n^2 - 100n + 6 = \Omega(n^2)$ since for $c = 2$, $2n^2 < 3n^2 - 100n + 6$ when $n > 100$

$3n^2 - 100n + 6 \neq \Omega(n^3)$ since for $c = 3$, $3n^2 - 100n + 6 < n^3$ when $n > 3$

$3n^2 - 100n + 6 = \Omega(n)$ since for any c , $cn < 3n^2 - 100n + 6$ when $n > 100$

$3n^2 - 100n + 6 = \Theta(n^2)$ since both O and Ω apply.

$3n^2 - 100n + 6 \neq \Theta(n^3)$ since only O applies.

Example 3.

$3n^2 - 100n + 6 = O(n^2)$ since for $c = 3$, $3n^2 > 3n^2 - 100n + 6$

$3n^2 - 100n + 6 = O(n^3)$ since for $c = 1$, $n^3 > 3n^2 - 100n + 6$ when $n > 3$

$3n^2 - 100n + 6 \neq O(n)$ since for any c , $cn < 3n^2$ when $n > c$

$3n^2 - 100n + 6 = \Omega(n^2)$ since for $c = 2$, $2n^2 < 3n^2 - 100n + 6$ when $n > 100$

$3n^2 - 100n + 6 \neq \Omega(n^3)$ since for $c = 3$, $3n^2 - 100n + 6 < n^3$ when $n > 3$

$3n^2 - 100n + 6 = \Omega(n)$ since for any c , $cn < 3n^2 - 100n + 6$ when $n > 100$

$3n^2 - 100n + 6 = \Theta(n^2)$ since both O and Ω apply.

$3n^2 - 100n + 6 \neq \Theta(n^3)$ since only O applies.

$3n^2 - 100n + 6 \neq \Theta(n)$ since only Ω applies.

Standard notations and common functions

- Floors and ceilings

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

Standard notations and common functions

- Logarithms:

$$\lg n = \log_2 n$$

$$\ln n = \log_e n$$

$$\log^k n = (\log n)^k$$

$$\lg \lg n = \lg(\lg n)$$

Standard notations and common functions

- Logarithms:

For all real $a > 0$, $b > 0$, $c > 0$, and n

$$a = b^{\log_b a}$$

$$\log_c (ab) = \log_c a + \log_c b$$

$$\log_b a^n = n \log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

Standard notations and common functions

- Logarithms:

$$\log_b (1/a) = -\log_b a$$

$$a^{\log_b c} = c^{\log_b a}$$

$$\log_b a = \frac{1}{\log_a b}$$

Standard notations and common functions

- Factorials

For $n \geq 0$ the Stirling approximation:

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

$$n! = o(n^n)$$

$$n! = \omega(2^n)$$

$$\lg(n!) = \Theta(n \lg n)$$

Algorithm Analysis...

- Factors affecting the running time
 - computer
 - compiler
 - algorithm used
 - input to the algorithm
 - The content of the input affects the running time
 - typically, the *input size* (number of items in the input) is the main consideration
 - E.g. sorting problem \Rightarrow the number of items to be sorted
 - E.g. multiply two matrices together \Rightarrow the total number of elements in the two matrices
- Machine model assumed
 - Instructions are executed one after another, with no concurrent operations \Rightarrow Not parallel computers

Example

- Calculate

$$\sum_{i=1}^N i^3$$

```
int sum(int n)
{
    int partialSum;

    1  partialSum=0;           1
    2  for (int i=1;i<=n;i++) 2N+2
    3      partialSum += i*i*i; 4N
    4  return partialSum;     1
}
```

- Lines 1 and 4 count for one unit each
- Line 3: executed N times, each time four units
- Line 2: (1 for initialization, N+1 for all the tests, N for all the increments) total 2N + 2
- total cost: 6N + 4 \Rightarrow O(N)

Worst- / average- / best-case

- **Worst-case running time** of an **algorithm**
 - The longest running time for **any input of size n**
 - An upper bound on the running time for any input
 - ⇒ guarantee that the algorithm will never take longer
 - Example: Sort a set of numbers in increasing order; and the data is in decreasing order
 - The worst case can occur fairly often
 - E.g. in searching a database for a particular piece of information
- **Best-case running time**
 - sort a set of numbers in increasing order; and the data is already in increasing order
- **Average-case running time**
 - May be difficult to define what “average” means

Running-time of algorithms

- Bounds are for the **algorithms**, rather than **programs**
 - programs are just implementations of an algorithm, and almost always the details of the program do not affect the bounds
- Bounds are for **algorithms**, rather than **problems**
 - A problem can be solved with several algorithms, some are more efficient than others

Arrays

One-Dimensional Arrays

- A list of values with the same data type that are stored using a single group name (array name).

- General array declaration statement:

data-type array-name[number-of-items];

- The *number-of-items* must be specified before declaring the array.

const int SIZE = 100;

float arr[SIZE];

One-Dimensional Arrays (cont.)

- Individual elements of the array can be accessed by specifying the name of the array and the element's *index*:

arr[3]

- *Warning*: indices assume values from 0 to *number-of-items - 1!!*

One-Dimensional Arrays (cont.)



Start here



element 3

1D Array Initialization

- Arrays can be initialized during their declaration

```
int arr[5] = {98, 87, 92, 79, 85};
```

```
int arr[5] = {98, 87} - what happens in this case??
```

- What is the difference between the following two declarations ?

```
char codes[] = {'s', 'a', 'm', 'p', 'l', 'e'};
```

```
char codes[] = "sample";
```

codes[0] codes[1] codes[2] codes[3] codes[4] codes[5] codes[6]



Two-dimensional Arrays

- A two-dimensional array consists of both rows and columns of elements.
- General array declaration statement:
data-type array-name[number-of-rows][number-of-columns];

Two-dimensional Arrays (cont.)

- The *number-of-rows* and *number-of-columns* must be specified before declaring the array.

```
const int ROWS = 100;
```

```
const int COLS = 50;
```

```
float arr2D[ROWS][COLS];
```

- Individual elements of the array can be accessed by specifying the name of the array and the element's row, column *indices*.

```
arr2D[3][5]
```

2D Array Initialization

- Arrays can be initialized during their declaration

```
int arr2D[3][3] = { {98, 87, 92}, {79, 85, 19},  
                  {32, 18, 2} };
```

- The compiler fills the array row by row (elements are stored in the memory in the same order).

1D Arrays as Arguments

- Individual array elements are passed to a function in the same manner as other variables.

```
max = find_max(arr[1], arr[3]);
```

- To pass the whole array to a function, you need to specify the name of the array only!!

```
#include <iostream.h>

float find_average(int [], int);

void main()
{
    const numElems = 5;
    int arr[numElems] = {2, 18, 1, 27, 16};

    cout << "The average is " << find_average(arr, numElems) << endl;
}

float find_average(int vals[], int n)
{
    int i;
    float avg;

    avg=0.0;
    for(i=0; i<n; i++)
        avg += vals[i];

    avg = avg/n;

    return avg;
}
```

1D Arrays as Arguments

(cont.)

- *Important:* this is essentially "call by reference":
 - a) The name of the array *arr* stores the address of the first element of the array *arr[0]* (i.e., $\&arr[0]$).
 - b) Every other element of the array can be accessed by using its index as an offset from the first element.



The starting address of *arr* array is $\&arr[0]$.

This is passed to the function `find_average()`

2D Arrays as Arguments

- Individual array elements are passed to a function in the same manner as other variables.

```
max = find_max(arr2D[1][1], arr2D[1][2]);
```

- To pass the whole array to a function, you need to specify the name of the array only!!
- The number of columns **must be specified** in the function prototype and function header.

```
#include <iostream.h>
```

```
float find_average(int [][][2], int, int);
```

```
void main()
```

```
{
```

```
    const numRows = 2;
```

```
    const numCols = 2;
```

```
    int arr2D[numRows][numCols] = {2, 18, 1, 27};
```

```
    float average;
```

```
    average = find_average(arr2D, numRows, numCols);
```

```
    cout << "The average is " << average << endl;
```

```
}
```

```
float find_average(int vals[][2], int n, int m)
{
    int i,j;
    float avg;

    avg=0.0;
    for(i=0; i<n; i++)
        for(j=0; j<m; j++)
            avg += vals[i][j];

    avg = avg/(n*m);

    return avg;
}
```

2D Arrays as Arguments (cont.)

- *Important*: this is essentially "call by reference":
 - a) The name of the array *arr2D* stores the address of *arr2D[0]* (i.e., *&arr2D[0]*)
 - b) *arr2D[0]* stores the address of the first element of the array *arr2D[0][0]* (*&arr2D[0][0]*)
 - c) Every other element of the array can be accessed by using its indices as an offset from the first element.

Searching and Sorting

- **Linear Search**
- **Binary Search**

Linear Search

- Searching is the process of determining whether or not a given value exists in a data structure or a storage media.
- We discuss two searching methods on one-dimensional arrays: linear search and binary search.
- The linear (or sequential) search algorithm on an array is:
 - Sequentially scan the array, comparing each array item with the searched value.
 - If a match is found; return the index of the matched element; otherwise return -1 .
- Note: linear search can be applied to both sorted and unsorted arrays.

Linear Search

- The algorithm translates to the following Java method:

```
public static int linearSearch(Object[] array,  
    Object key)  
{  
    for(int k = 0; k < array.length; k++)  
        if(array[k].equals(key))  
            return k;  
    return -1;  
}
```

Binary Search

- Binary search uses a recursive method to search an array to find a specified value
- The array must be a sorted array:
 $a[0] \leq a[1] \leq a[2] \leq \dots \leq a[\text{finalIndex}]$
- If the value is found, its index is returned
- If the value is not found, -1 is returned
- Note: Each execution of the recursive method reduces the search space by about a half

Binary Search

- An algorithm to solve this task looks at the middle of the array or array segment first
- If the value looked for is smaller than the value in the middle of the array
 - Then the second half of the array or array segment can be ignored
 - This strategy is then applied to the first half of the array or array segment

Binary Search

- If the value looked for is larger than the value in the middle of the array or array segment
 - Then the first half of the array or array segment can be ignored
 - This strategy is then applied to the second half of the array or array segment
- If the value looked for is at the middle of the array or array segment, then it has been found
- If the entire array (or array segment) has been searched in this way without finding the value, then it is not in the array

Pseudocode for Binary Search

Display 11.5 Pseudocode for Binary Search ❖

ALGORITHM TO SEARCH $a[\text{first}]$ THROUGH $a[\text{last}]$

```
/**  
  Precondition:  
   $a[\text{first}] \leq a[\text{first} + 1] \leq a[\text{first} + 2] \leq \dots \leq a[\text{last}]$   
*/
```

TO LOCATE THE VALUE KEY:

```
if (first > last) //A stopping case  
  return -1;  
else  
{  
  mid = approximate midpoint between first and last;  
  if (key == a[mid]) //A stopping case  
    return mid;  
  else if key < a[mid] //A case with recursion  
    return the result of searching  $a[\text{first}]$  through  $a[\text{mid} - 1]$ ;  
  else if key > a[mid] //A case with recursion  
    return the result of searching  $a[\text{mid} + 1]$  through  $a[\text{last}]$ ;  
}
```

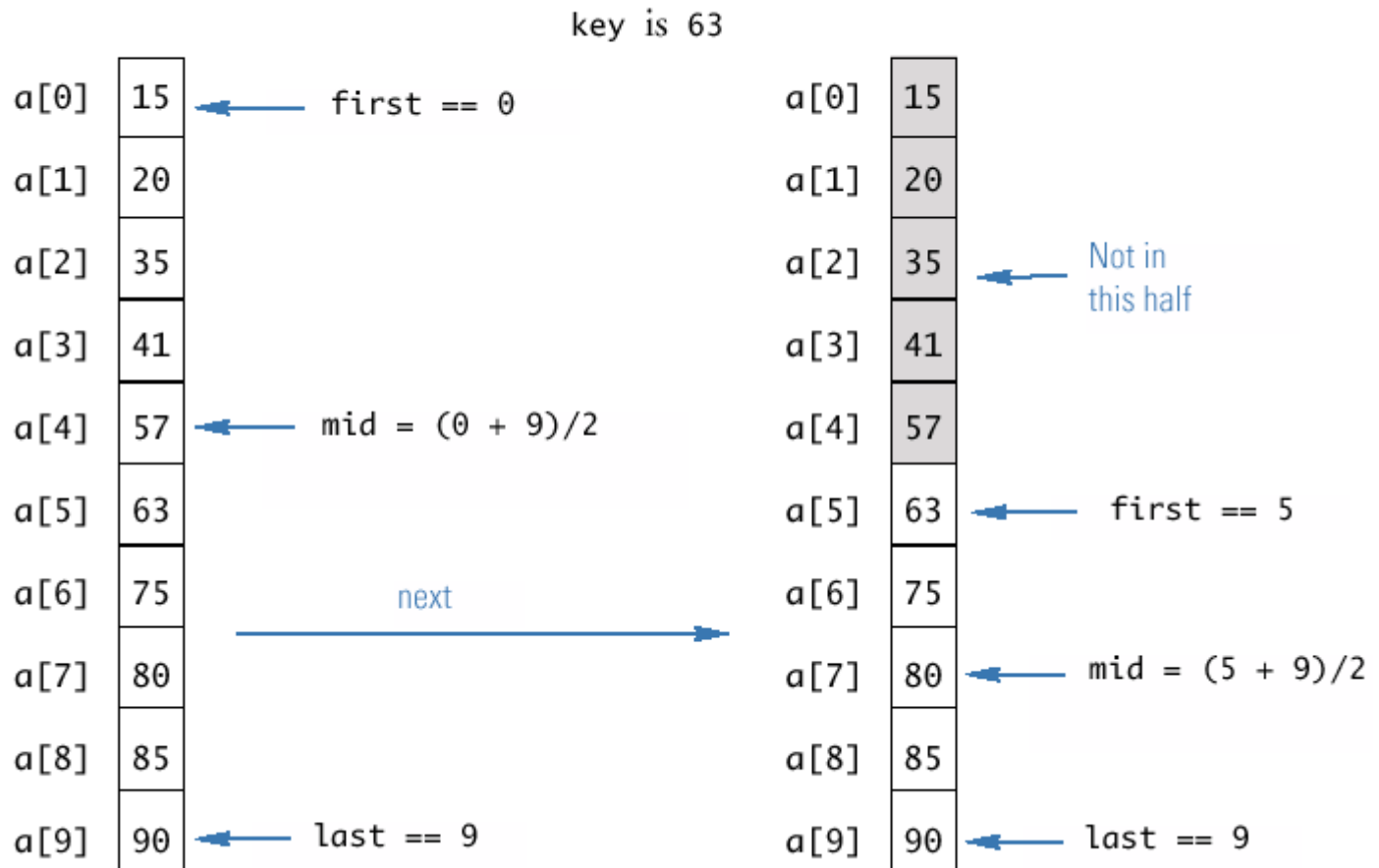
Recursive Method for Binary

Display 11.6 Recursive Method for Binary Search ❖

```
1 public class BinarySearch
2 {
3     /**
4     Searches the array a for key. If key is not in the array segment, then -1 is
5     returned. Otherwise returns an index in the segment such that key == a[index].
6     Precondition: a[first] <= a[first + 1]<= ... <= a[last]
7     */
8     public static int search(int[] a, int first, int last, int key)
9     {
10         int result = 0; //to keep the compiler happy.
11
12         if (first > last)
13             result = -1;
14         else
15         {
16             int mid = (first + last)/2;
17
18             if (key == a[mid])
19                 result = mid;
20             else if (key < a[mid])
21                 result = search(a, first, mid - 1, key);
22             else if (key > a[mid])
23                 result = search(a, mid + 1, last, key);
24         }
25     }
26 }
```

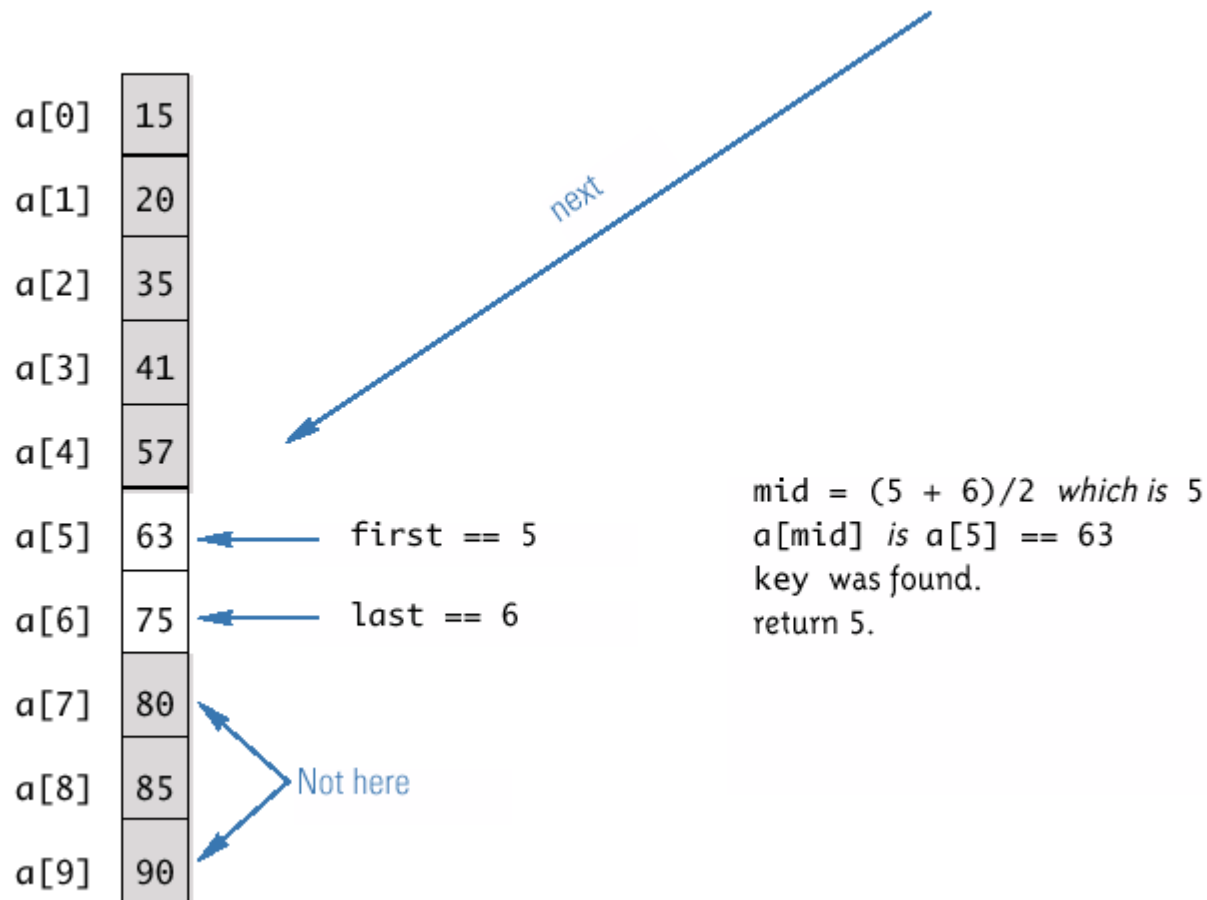
Execution of the Method `search` (Part 1 of 2)

Display 11.7 Execution of the Method `search` ❖



Execution of the Method `search` (Part 1 of 2)

Display 11.7 Execution of the Method `search` ✦ (continued)



Checking the search Method

1. There is no infinite recursion
 - On each recursive call, the value of **first** is increased, or the value of **last** is decreased
 - If the chain of recursive calls does not end in some other way, then eventually the method will be called with **first** larger than **last**

Checking the `search` Method

2. Each stopping case performs the correct action for that case
 - If `first > last`, there are no array elements between `a[first]` and `a[last]`, so `key` is not in this segment of the array, and `result` is correctly set to `-1`
 - If `key == a[mid]`, `result` is correctly set to `mid`

Checking the search Method

3. For each of the cases that involve recursion, *if* all recursive calls perform their actions correctly, *then* the entire case performs correctly
 - If `key < a[mid]`, then `key` must be one of the elements `a[first]` through `a[mid-1]`, or it is not in the array
 - The method should then search only those elements, which it does
 - The recursive call is correct, therefore the entire action is correct

Checking the `search` Method

- If `key > a[mid]`, then `key` must be one of the elements `a[mid+1]` through `a[last]`, or it is not in the array
- The method should then search only those elements, which it does
- The recursive call is correct, therefore the entire action is correct

The method `search` passes all three tests:

Therefore, it is a good recursive method definition

Efficiency of Binary Search

- The binary search algorithm is extremely fast compared to an algorithm that tries all array elements in order
 - About half the array is eliminated from consideration right at the start
 - Then a quarter of the array, then an eighth of the array, and so forth

Efficiency of Binary Search

- Given an array with 1,000 elements, the binary search will only need to compare about 10 array elements to the key value, as compared to an average of 500 for a serial search algorithm
- The binary search algorithm has a worst-case running time that is logarithmic: $O(\log n)$
 - A serial search algorithm is linear: $O(n)$
- If desired, the recursive version of the method `search` can be converted to an iterative version that will run more efficiently

Iterative Version of Binary Search (Part 1 of 2)


Display 11.9 Iterative Version of Binary Search ❖

```
1  /**
2   Searches the array a for key. If key is not in the array segment, then -1 is
3   returned. Otherwise returns an index in the segment such that key == a[index].
4   Precondition: a[lowEnd] <= a[lowEnd + 1]<= ... <= a[highEnd]
5   */
6  public static int search(int[] a, int lowEnd, int highEnd, int key)
7  {
8      int first = lowEnd;
9      int last = highEnd;
10     int mid;

11     boolean found = false; //so far
12     int result = 0; //to keep compiler happy

13     while ( (first <= last) && !(found) )
14     {
15         mid = (first + last)/2;
```

Iterative Version of Binary Search (Part 2 of 2)

Display 11.9 Iterative Version of Binary Search  (continued)

```
16         if (key == a[mid])
17         {
18             found = true;
19             result = mid;
20         }
21         else if (key < a[mid])
22         {
23             last = mid - 1;
24         }
25         else if (key > a[mid])
26         {
27             first = mid + 1;
28         }
29     }

30     if (first > last)
31         result = -1;

32     return result;
33 }
```

Fibonacci Search

- Given a sorted array `arr[]` of size `n` and an element `x` to be searched in it. Return index of `x` if it is present in array else return `-1`.

Examples:

- Input: `arr[] = {2, 3, 4, 10, 40}`, `x = 10` Output: 3 Element `x` is present at index 3. Input: `arr[] = {2, 3, 4, 10, 40}`, `x = 11` Output: -1 Element `x` is not present. Fibonacci Search is a comparison-based technique that uses Fibonacci numbers to search an element in a sorted array.

Similarities with Binary Search:

- Works for sorted arrays
- A Divide and Conquer Algorithm.
- Has $\log n$ time complexity.

Differences with Binary Search:

- Fibonacci Search divides given array in unequal parts
- Binary Search uses division operator to divide range. Fibonacci Search doesn't use $/$, but uses $+$ and $-$. The division operator may be costly on some CPUs.
- Fibonacci Search examines relatively closer elements in subsequent steps. So when input array is big that cannot fit in CPU cache or even in RAM, Fibonacci Search can be useful.