IECS Data Structures

- Credit: 3, + 1.5 (Fall, 2019)
- Classes: 15:10-17:00, 8:10-9:00 (Mon., Tue.); 9:10-12:00 (Tue.)
- Instructor: Yi-Fen Liu
- Textbook:

  2008, E. Horowitz, S. Sahni and S. Anderson-Freed, Fundamentals of Data Structures in C (2nd edition), by Silicon Press.
- Grading:

  Program Exercises(30%), In-class Program Tests (15%)

  Mid-term (20%), Final Examination (25%)

  Quizzes (10%)

  Pass the CPE (the maximum # of passed Qs.) (the upper bound of the bonus: +10)

  - 1Q (+1) 2Qs (+3) 3Qs (+6) 4Qs(+10) ... 7Qs(+10)
- TA:

  張維峻(d0657491@mail.fcu.edu.tw)

  林品秀 (d0676600@o365.fcu.edu.tw)

- Course content:
  - Ch1

Chapter 1
Basic Concept

Yi-Fen Liu

Department of IECS, FCU

References:
- E. Horowitz, S. Sahni and S. Anderson-Freed, *Fundamentals of Data Structures (2nd Edition)*
- Slides are credited from Prof. Chung, NTHU

Outline

- Overview: System Life Cycle
- Algorithm Specification
- Data Abstraction
- Performance Analysis

SYSTEM LIFE CYCLE

System Life Cycle (1)

- As systems, these programs undergo a development process called the **system life cycle**
- The cycle consists of
  - Requirements
  - Analysis
  - Design
  - Coding
  - Verification

System Life Cycle (2)

- Requirements
  - All large programming projects begin with **a set of specifications** that define **the purpose of the project**
- Analysis
  - Breaking the problem down into manageable pieces
  - Two approaches: **bottom-up** and **top-down**
- Design
  - The designer approaches the system from the perspectives of both the **data objects** and the **operations** performed on them

System Life Cycle (3)

- Refinement and coding
  - **Choose representations for the data objects** and **write algorithm** for each operation on them
- Verification
  - Developing correctness proofs for the program
  - Testing the program with a variety of input data
  - Removing errors (viz. debugging)

ALGORITHM SPECIFICATION

Algorithm Specification (1)

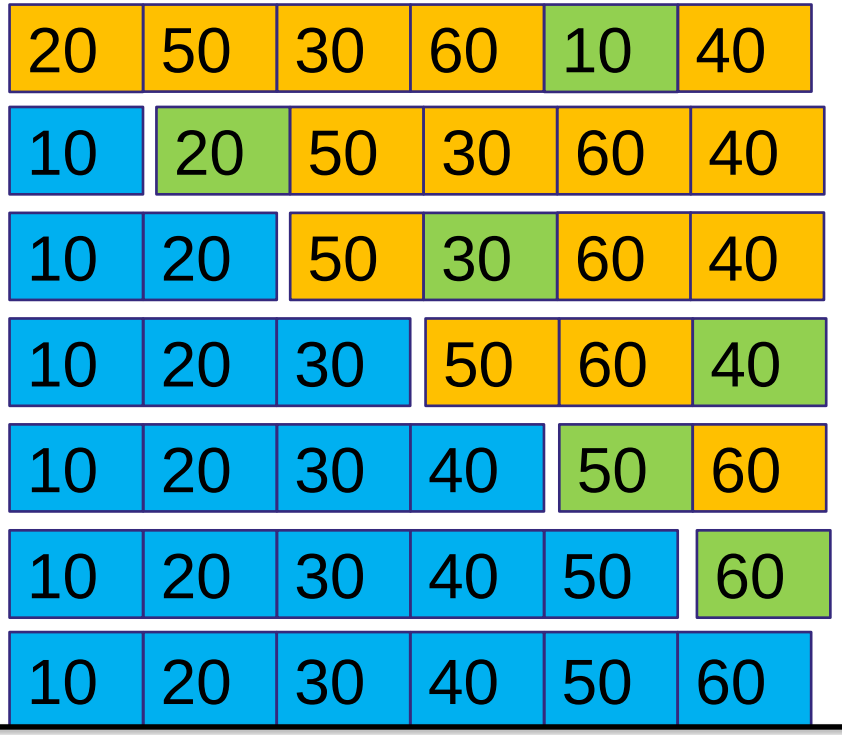
- An *algorithm* is a finite set of instructions that accomplishes a particular task.
  - Criteria
    - **Input:** zero or more quantities that are externally supplied
    - **Output:** at least one quantity is produced
    - **Definiteness:** clear and unambiguous
    - **Finiteness:** terminate after a finite number of steps
    - **Effectiveness:** instruction is basic enough to be carried out
- A *program* does not have to satisfy the **finiteness** criteria

Algorithm Specification (2)

- Representation
  - A natural language, like English or Chinese.
  - A graphic, like flowcharts.
  - A computer language, like C.
- Programs = Algorithms + Data structures [Niklus Wirth]

Selection Sort (1)

- From those integers that are currently unsorted, find the smallest and place it next in the sorted list
  - Ex: 20, 50, 30, 60, 10, 40



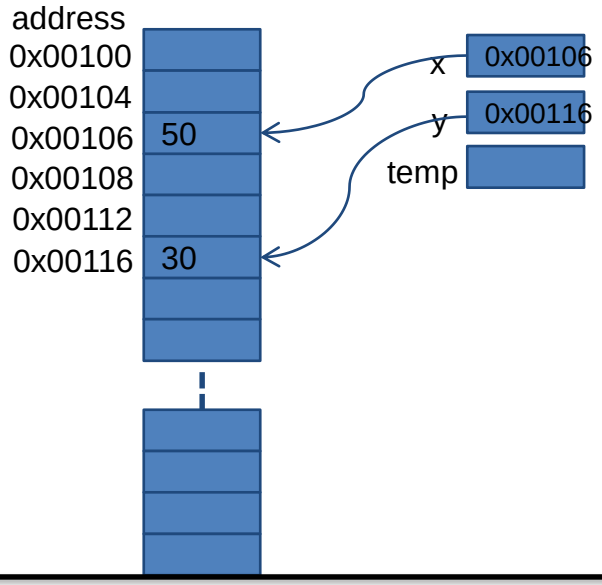
Selection Sort (2)

```
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];
}
```

```
void swap(int *x, int *y)
{
     int  temp = *x;

     *x = *y;
     *y = temp;
}
```



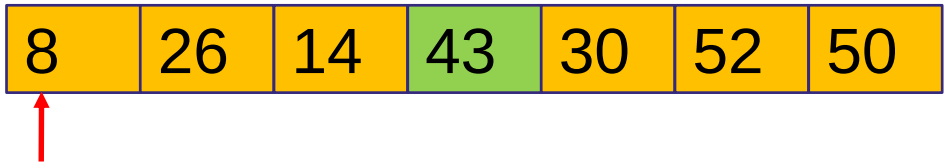
#define SWAP (x, y, t) ((t) = (x), (x) = (y), (y) = (t))

Selection Sort (3)

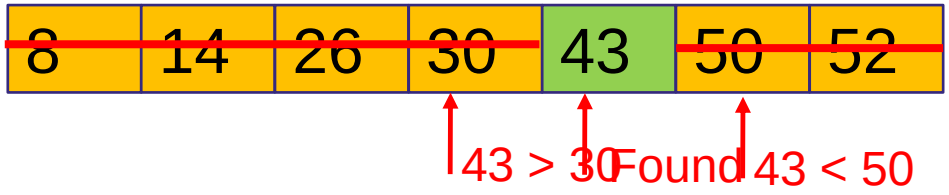
```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x,y,t) ((t) = (x), (x)= (y), (y) = (t))
void sort(int [],int); /*selection sort */
void main(void)
{
   int i,n;
   int list[MAX_SIZE];
   printf("Enter the number of numbers to generate: ");
   scanf("%d",&n);
   if( n < 1 || n > MAX_SIZE) {
     fprintf(stderr, "Improper value of n\n");
     exit(1);
   }
   for (i = 0; i < n; i++) {/*randomly generate numbers*/
      list[i] = rand() % 1000;
      printf("%d  ",list[i]);
   }
   sort(list,n);
   printf("\n Sorted array:\n ");
   for (i = 0; i < n; i++) /* print out sorted numbers */
     printf("%d  ",list[i]);
   printf("\n");
}
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);
   }
}
```

Binary Search (1)

- Unordered list
  - Ex:
  - Find 43
  - Search time: *n*



- Ordered list
  - Ex:
  - Find 43



Binary Search (2)

[0]	[1]	[2]	[3]	[4]	[5]	[6]
8	14	26	30	43	50	52

left	*right*	*middle*	*list*[*middle*]	:	*searchnum*
0	6	3	30	<	43
4	6	5	50	>	43
4	4	4	43	==	43
0	6	3	30	>	18
0	2	1	14	<	18
2	2	2	26	>	18
2	1	-			

Binary Search (3)

- Algorithm

```
while (there are more integers to check) {
    middle = (left + right) / 2;
    if (searchnum < list[middle])
        right = middle - 1;
    else if (searchnum == list[middle])
        return middle;
    else left = middle + 1;
}
```

Binary Search (4/4)

- Program

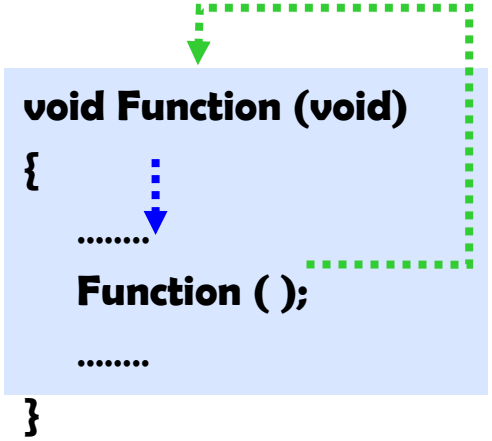
```
int binsearch (int list[], int searchnum, int left, int right){
/* search list[0] <= list[1] <= … <= list[n-1] for searchnum.
    Return its position if found. Otherwise return -1 */
    int middle;
    while (left <= right) {
      middle = (left + right)/2;
      switch ( COMPARE (list[middle], searchnum)){
      case -1: left = middle + 1;
            break;
      case 0 : return middle;
      case 1 : right = middle – 1;
      }
    }
    return -1;
}
```

Recursive Algorithms

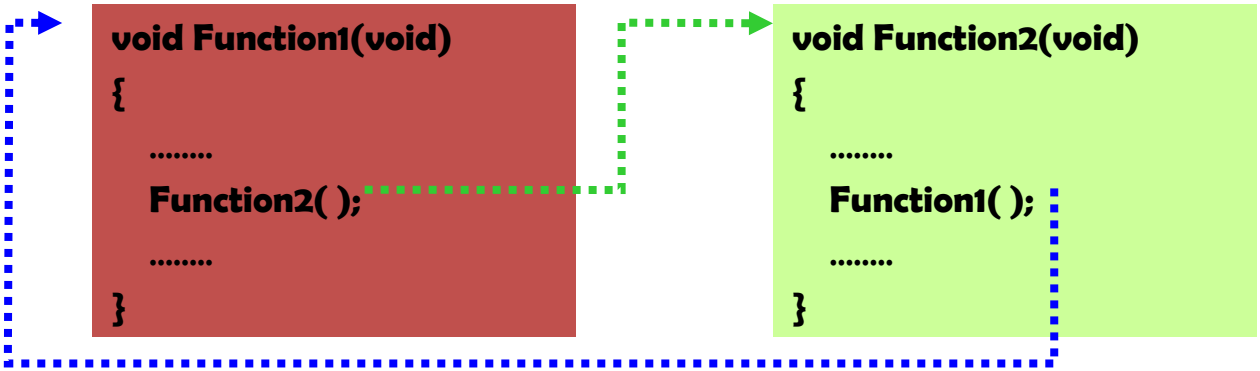
- Programmer view a function as something that is invoked (called) by another function
  - It executes its code and then returns control to the calling function
- This perspective ignores the fact that functions can call themselves (*direct recursion*)
- They may call other functions that invoke the calling function again (*indirect recursion*)
  - extremely powerful
  - frequently allow us to express an otherwise complex process in very clear term
- We should express a recursive algorithm when the problem itself is defined recursively

Types of Recursion

- Direct recursion



- Indirect recursion



Example 1: Factorial (1)

- The factorial of a number is the product of the integral values from 1 to the number

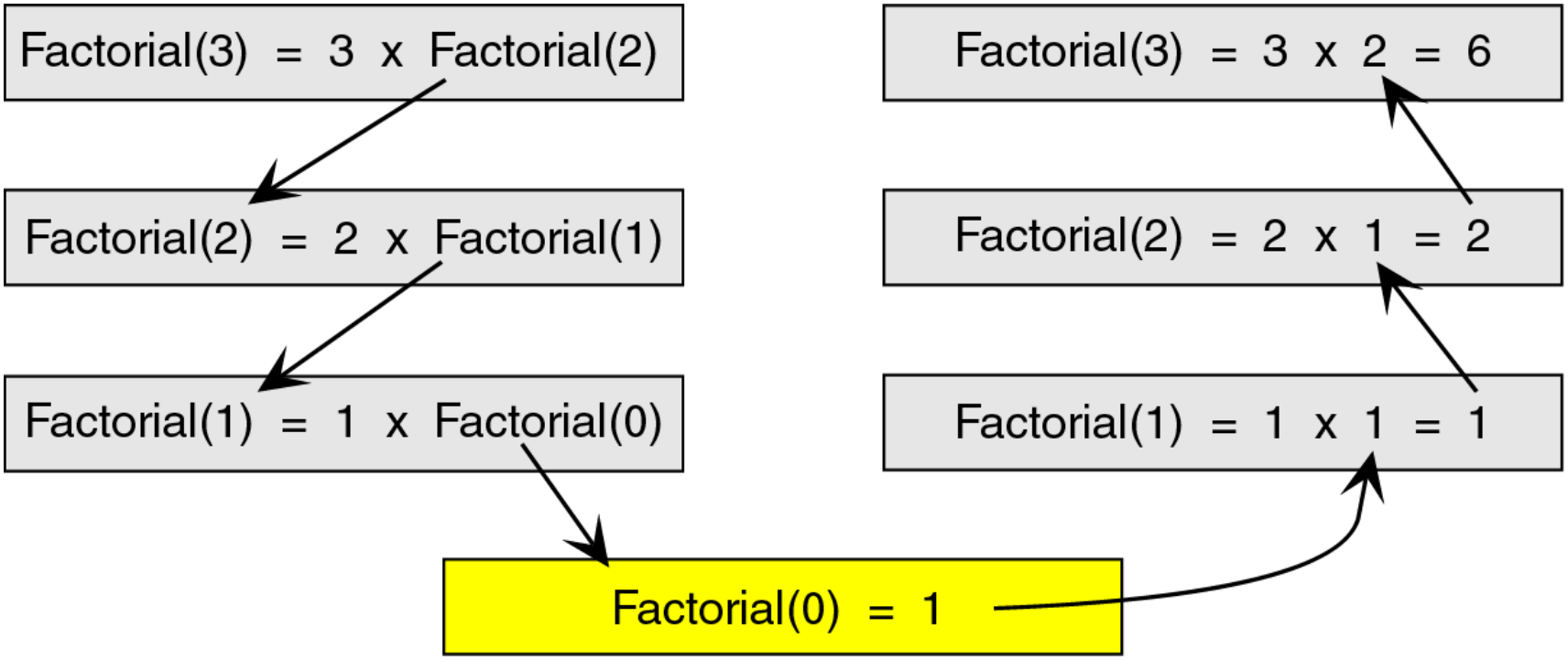
$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 & \text{if } n > 0 \end{cases}$$

$= \text{Factorial}(n-1)$

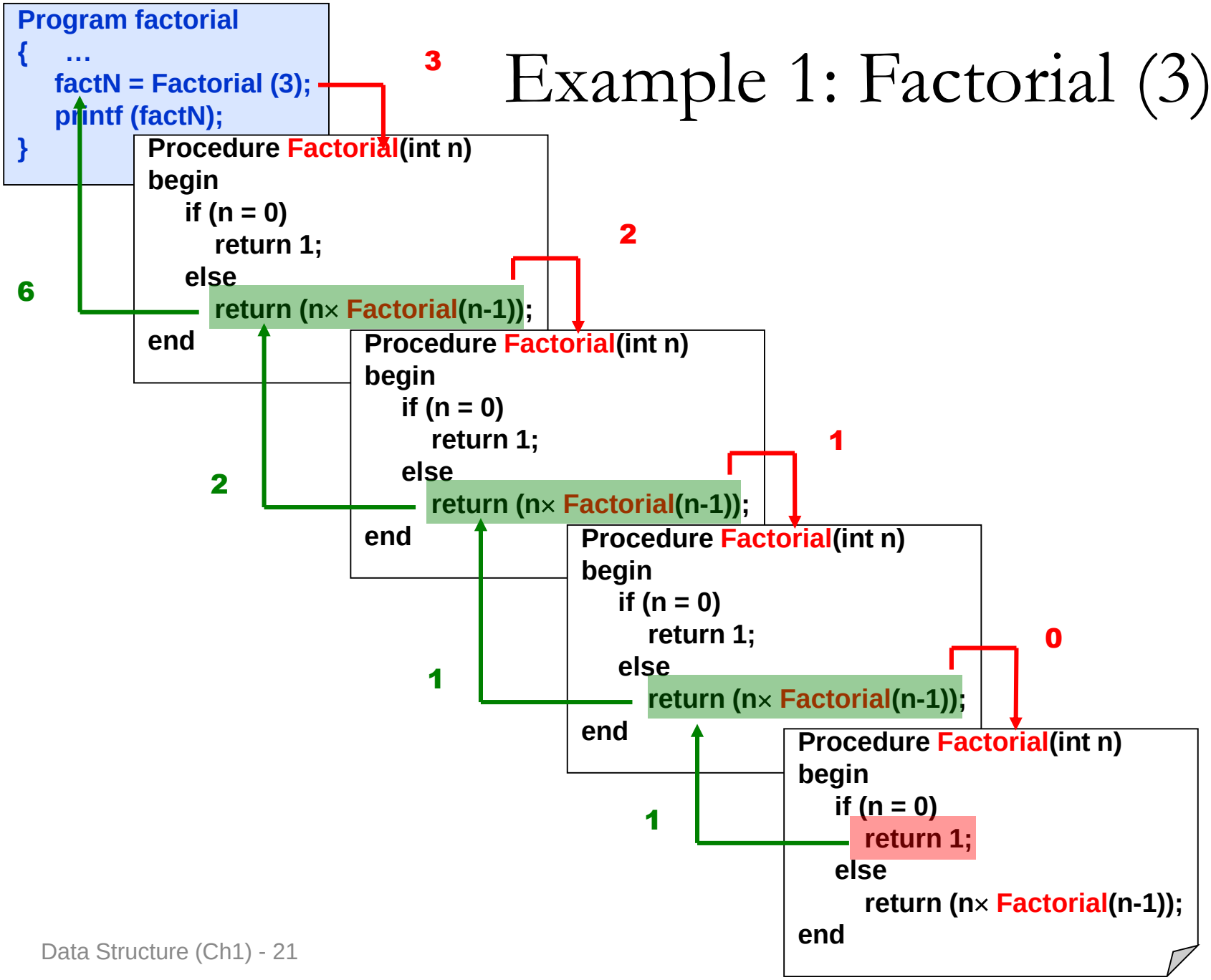
  - Recursion defined

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (\text{Factorial}(n-1)) & \text{if } n > 0 \end{cases}$$

Example 1: Factorial (2)



Example 1: Factorial (3)



Example 2: List Sum (1)

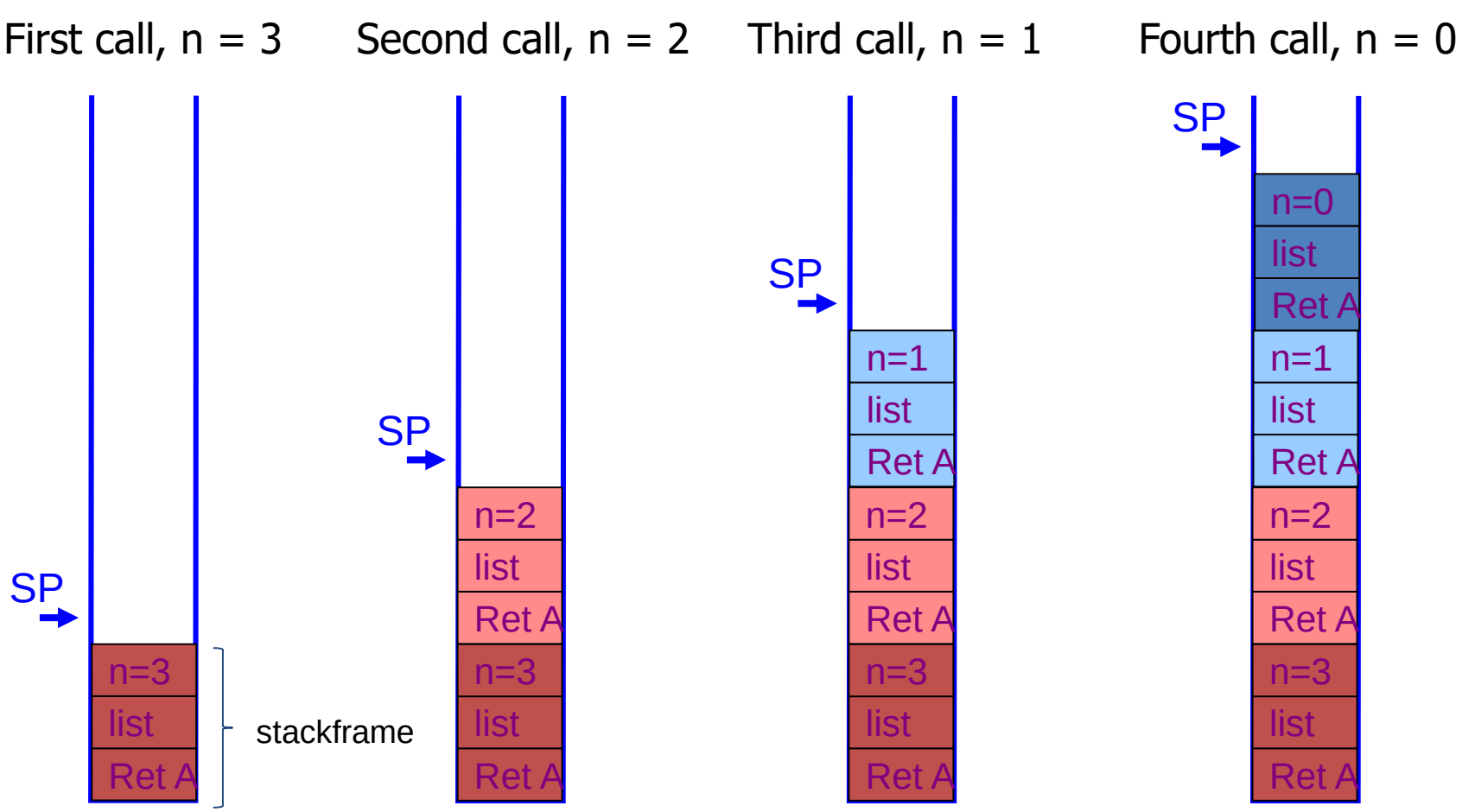
```
Int sum_list(int list[], int n)
{
/* This algorithm returns the summation of elements in the sublist. */

    if (n > 0)
        return( sum_list(list, n-1) + list[n-1] );
    else
        return( 0 );
}
```

- The space complexity in this algorithm is static or variable?
- Space requirement for each call
  - list[] 4 bytes // pointer to the array
  - n 4 bytes
  - return address 4 bytes

Example 2: List Sum (2)

List[3] = {1, 2, 3}



The space complexity in this algorithm is static or variable?

Stackframe

- The stackframe contains four different elements
  - The parameters to be processed by the called algorithm
  - The local variables in the calling algorithm
  - The return statement in the calling algorithm
  - The expression that is to receive the return value (if any)

The Design Methodology

- Determine the base case
  - Every recursive algorithm must have a base case
- Determine the general case
- Combine the base case and general case into an algorithm

- Every recursive call must either solve a part of the problem or reduce the size of the problem

Factorial(2) → 2 * factorial(1)

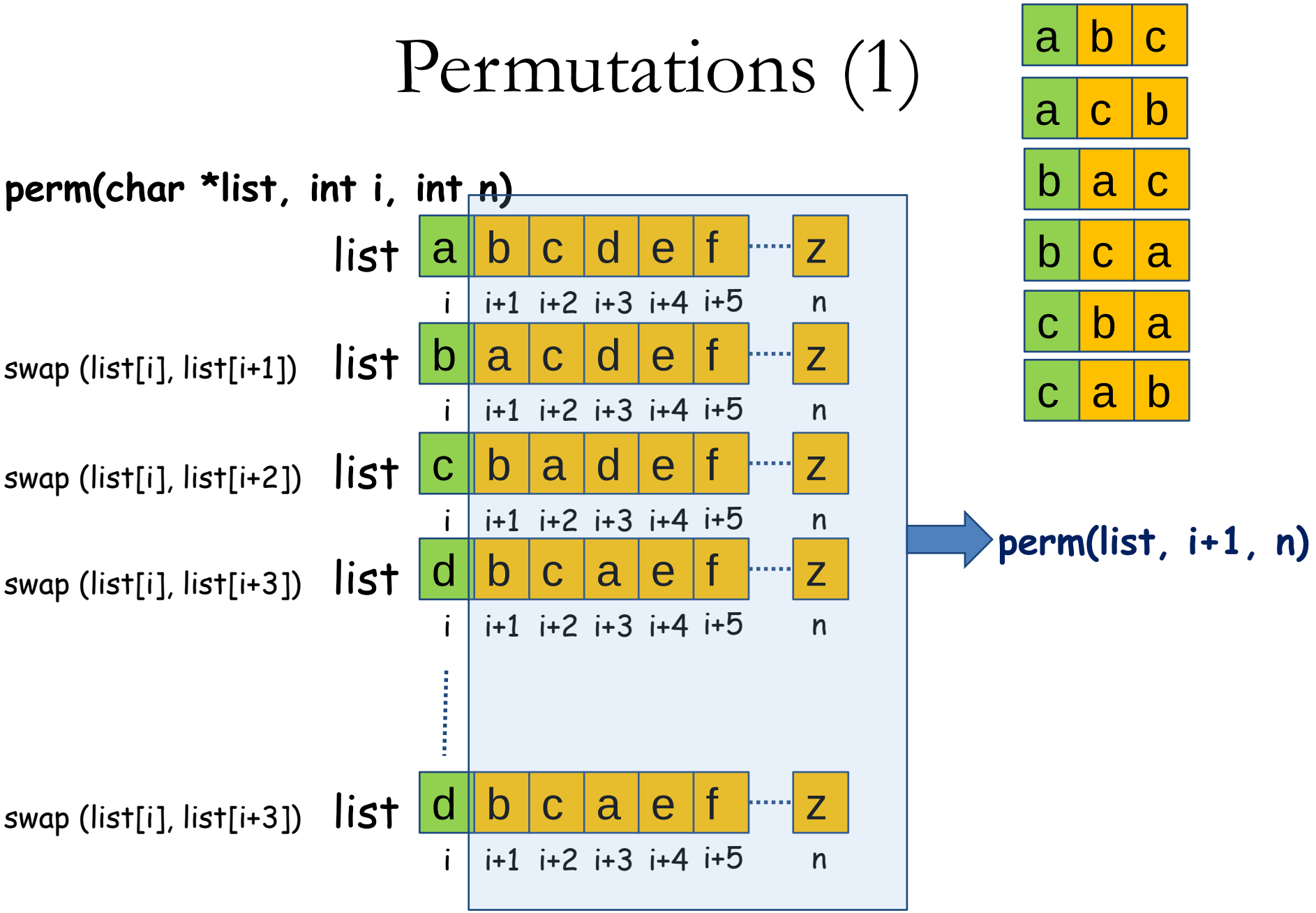
Limitations of Recursion

- Recursive solutions may involve extensive overhead because they use call
  - When a call is made, it takes time to build a *stackframe* and push it into stack
  - When a return is executed, the *stackframe* must be popped from the stack and the local variables reset to their previous values
- You should not use recursion if the answer to any of the following question is no
  - Is the algorithm or data structure naturally suited to recursion?
  - Is the recursive solution shorter and more understandable?
  - Does the recursive solution run in acceptable time and space limits?

Binary Search

```
int binsearch(int list[], int searchnum, int left,
                                         int right)
{
/* search list[0] <= list[1] <=  · · ·  <= list[n-1] for
searchnum. Return its position if found. Otherwise
return -1 */
   int middle;
   if (left <= right) {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: return
            binsearch(list, searchnum, middle + 1, right);
         case 0 : return middle;
         case 1 : return
            binsearch(list, searchnum, left, middle - 1);
      }
   }
   return -1;
}
```

Permutations (1)



Permutations (2)

Call Stack:

main

Perm (string , 0 , 2)

Perm (string , 1 , 2)

Perm (string , 2 , 2)

```
void p
/* ge                                    */
{
   int
   if
      }
   els
   /*
   gene
      for (j = i; j <= n; j++) {
         SWAP(list[i],list[j],temp);
         perm(list,i+1,n);
         SWAP(list[i],list[j],temp);
      }
   }
}
```

First,
We Permutations the
char *string :
by call perm(string,0,2);
0 is start index
2 is end index

Print The String
"bca"

SWAP (list[1],list[2], temp)
SWAP 'a' ⇔ 'c'

I=1 J=3 N=2

Call : perm (list,2, 2)

Program 1.8: Recursive permutation generator

Permutations (3)

```
void perm(char *list, int i, int n)
/* generate all the permutations of list[i] to list[n] */
{
   int j, temp;
   if (i == n) {
      for (j = 0; j <= n; j++)
         printf("%c", list[j]);
      printf("    ");
   }
   else {
   /* list[i] to list[n] has more than one permutation,
   generate these recursively */
      for (j = i; j <= n; j++) {
         SWAP(list[i],list[j],temp);
         perm(list,i+1,n);
         SWAP(list[i],list[j],temp);
      }
   }
}
```

Program 1.8: Recursive permutation generator

```
lv0 perm: i=0, n=2 abc
lv0 SWAP: i=0, j=0 abc
lv1 perm: i=1, n=2 abc
lv1 SWAP: i=1, j=1 abc
lv2 perm: i=2, n=2 abc
print: abc
lv1 SWAP: i=1, j=1 abc
lv1 SWAP: i=1, j=2 acb
lv2 perm: i=2, n=2 acb
print: acb
lv1 SWAP: i=1, j=2 abc
lv0 SWAP: i=0, j=0 abc
lv0 SWAP: i=0, j=1 bac
lv1 perm: i=1, n=2 bac
lv1 SWAP: i=1, j=1 bac
lv2 perm: i=2, n=2 bac
print: bac
lv1 SWAP: i=1, j=1 bac
lv1 SWAP: i=1, j=2 bca
lv2 perm: i=2, n=2 bca
print: bca
lv1 SWAP: i=1, j=2 bac
lv0 SWAP: i=0, j=1 abc
lv0 SWAP: i=0, j=2 cba
lv1 perm: i=1, n=2 cba
lv1 SWAP: i=1, j=1 cba
lv2 perm: i=2, n=2 cba
print: cba
lv1 SWAP: i=1, j=1 cba
lv1 SWAP: i=1, j=2 cab
lv2 perm: i=2, n=2 cab
print: cab
lv1 SWAP: i=1, j=2 cba
lv0 SWAP: i=0, j=2 abc
```

DATA ABSTRACTION

Data Abstraction (1)

- Data Type
  A data type is *a collection of objects* and *a set of operations* that act on those objects.
  - For example, the data type `int` consists of the objects `{0, +1, -1, +2, -2, …, INT_MAX, INT_MIN}` and the operations `+, -, *, /, and %`.
- The data types of C
  - The basic data types: `char, int, float and double`
  - The group data types: `array` and `struct`
  - The pointer data type
  - The user-defined types

Data Abstraction (2)

- Abstract Data Type
  - An **abstract data type (ADT)** is a data type that is organized in such a way that the specification of the objects and the operations on the objects is separated from the representation of the objects and the implementation of the operations.
  - We know what is does, but not necessarily how it will do it.

Abstraction

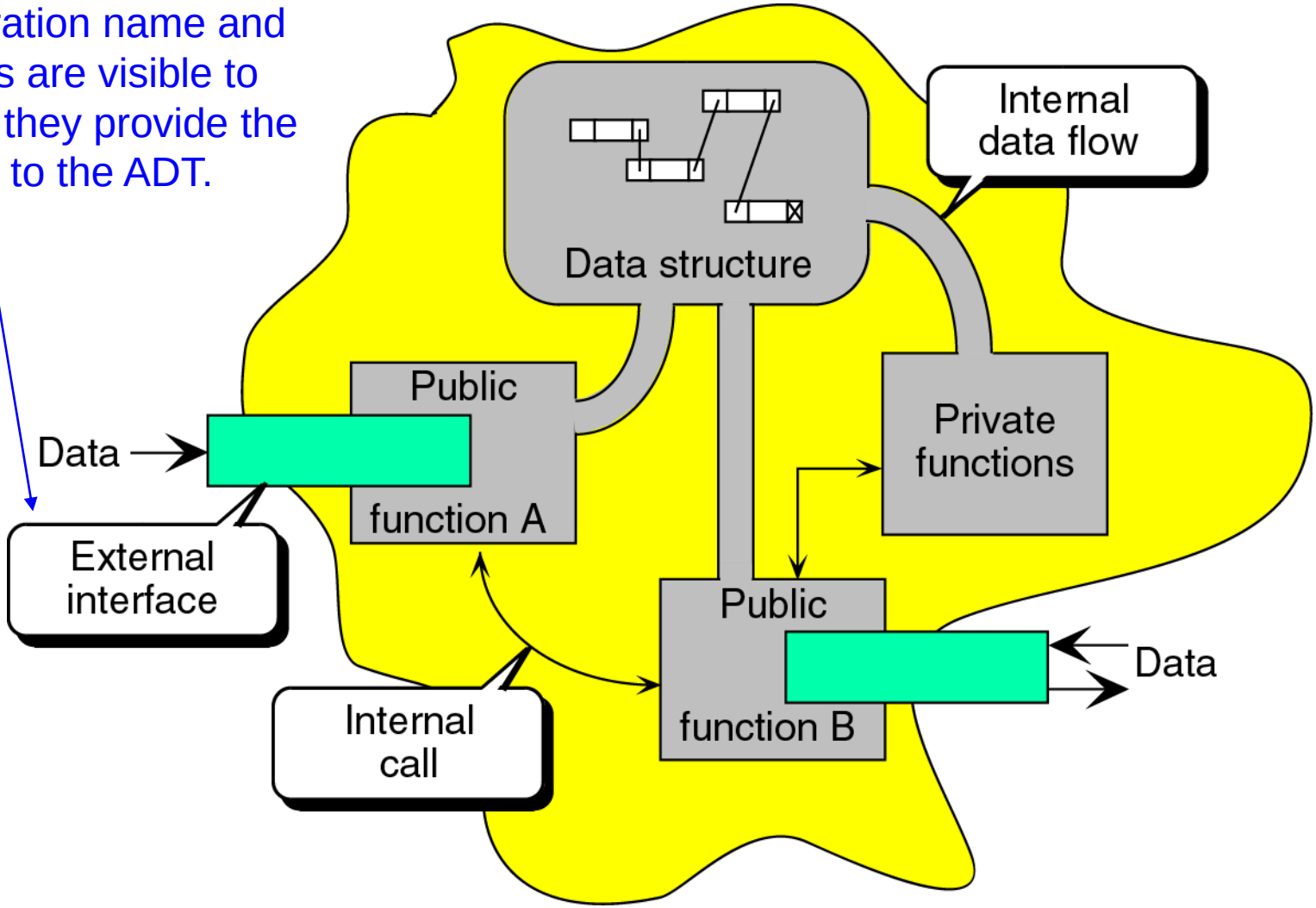
- The concept of abstraction means
  - We know what a data type can do.
  - How it is done is hidden.
- Example: list
  - Data structures to support list
    - array, linked list, file, …
  - User should not be aware of the structure we use
  - Operations on list are all the same
    - insert, retrieve, …

Specification vs. Implementation

- An ADT is implementation independent
- Operation specification
  - function name
  - the types of arguments
  - the type of the results
- The functions of a data type can be classify into several categories:
  - creator / constructor
  - transformers
  - observers / reporters

Abstract data type model

Only the operation name and its parameters are visible to the user, and they provide the only interface to the ADT.



Example : Natural_Number

structure *Natural_Number* is

objects: an ordered subrange of the integers starting at zero and ending at the maximum integer (*INT_MAX*) on the computer

functions:

for all $x, y \in$ *Nat_Number*; *TRUE*, *FALSE* $\in$ *Boolean*
and where +, −, <, and == are the usual integer operations

Nat_No Zero( )	::=	0
Boolean Is_Zero(x)	::=	**if** (x) **return** *FALSE*
		else return *TRUE*
Nat_No Add(x, y)	::=	**if** (($x + y$) <= *INT_MAX*) **return** $x + y$
		else return *INT_MAX*
Boolean Equal(x, y)	::=	**if** (x == y) **return** *TRUE*
		else return *FALSE*
Nat_No Successor(x)	::=	**if** (x == *INT_MAX*) **return** x
		else return $x + 1$
Nat_No Subtract(x, y)	::=	**if** ($x < y$) **return** 0
		else return $x - y$

::= is defined as

end *Natural_Number*

PERFORMANCE ANALYSIS

Performance Analysis

- 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

Examples (1)

$S_{abc}(i)=0$

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

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

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

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

Examples (2)

- The program is a recursive function for addition. Figure shows the number of bytes required for one recursive call.

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

Program: Recursive function for summing a list of numbers

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

Figure: Space needed for one recursive call of Program

$S_{sum}(i)=S_{sum}(n)=6n$

Time Complexity (1)

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

- The time, $T(P)$, taken by a program, P, is the sum of its compile time C and its run (or execution) time, $T_P(i)$
- Fixed time requirements
  - Compile time (C), independent of instance characteristics
- Variable time requirements
  - Run (execution) time T_P
  - *Example: a simple program that adds and subtracts numbers*
    - $T_P(n)=c_aADD(n)+c_sSUB(n)+c_lLDA(n)+c_{st}STA(n)$
    - c_a, c_s, c_l, c_{st} are constants that refer to the time needed to perform each operation

Time Complexity (2)

- A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics
  - Example
    (Regard as the same unit machine independent)
    - 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

Example

- Iterative summing of a list of numbers

```
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

Example

- Tabular Method
  - Step count table

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

Example

- Recursive summing of a list of numbers

```
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 steps

Example

- For recursive summing function

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

Asymptotic notation (O, Ω, Θ)

- Consider the 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 c_1, c_2, and c_3, the n beyond which c_3n is always faster than $c_1n^2+c_2n$

Performance Analysis

- 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, $n \geq n_0$
- Definition: [Omega]
  - $f(n) = \Omega(g(n))$ (read as "f of n is omega of g of n") *iff* there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all n, $n \geq n_0$
- Definition: [Theta]
  - $f(n) = \Theta(g(n))$ (read as "f of n is theta of g of n") *iff* there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$

Performance Analysis

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

Examples

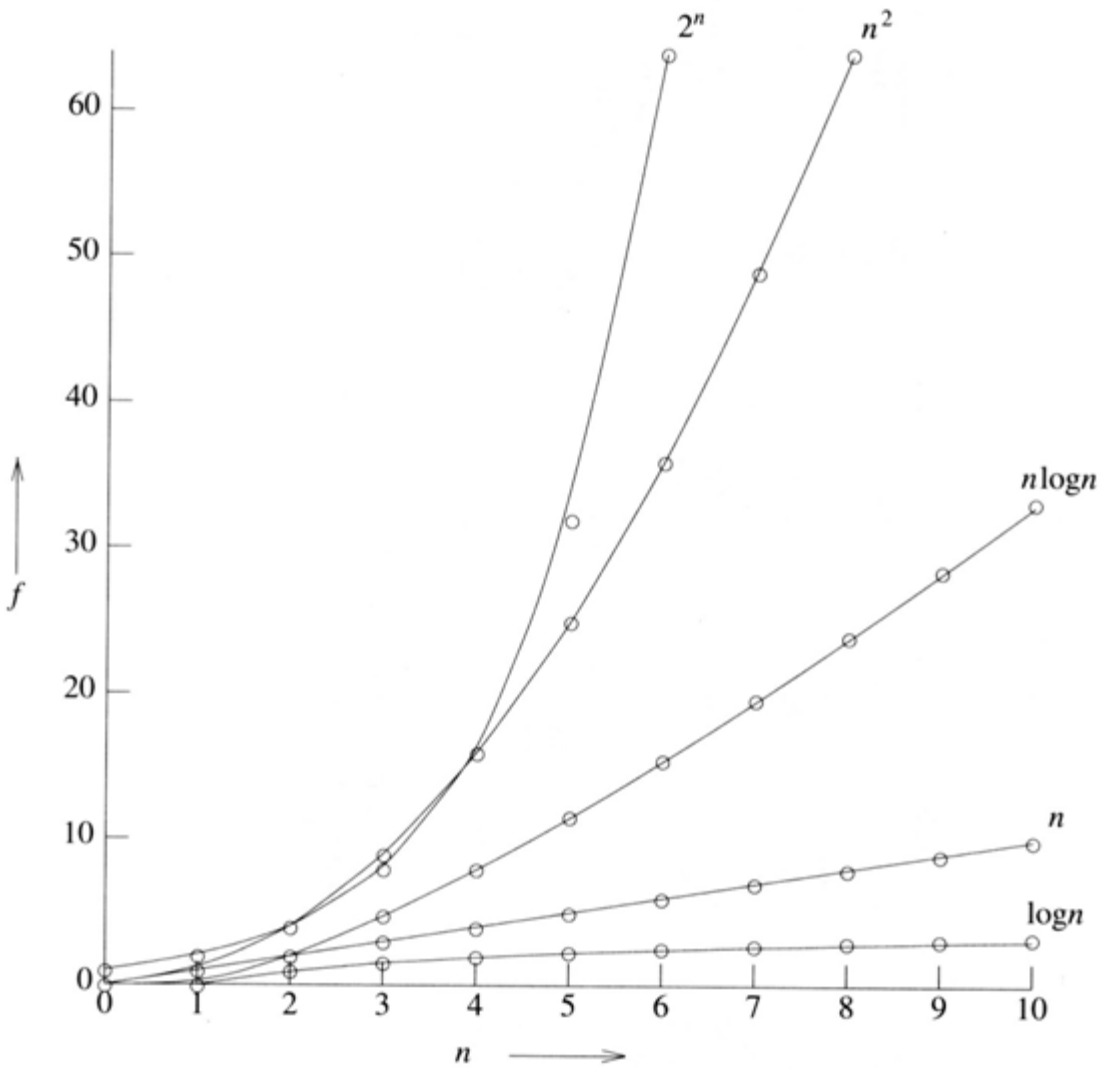
- $f(n) = 3n+2$
  - $3n + 2 <= 4n$, for all $n >= 2$, $\therefore 3n + 2 = O(n)$
    $3n + 2 >= 3n$, for all $n >= 1$, $\therefore 3n + 2 = \Omega(n)$
    $3n <= 3n + 2 <= 4n$, for all $n >= 2$, $\therefore 3n + 2 = \Theta(n)$
- $f(n) = 10n^2+4n+2$
  - $10n^2+4n+2 <= 11n^2$, for all $n >= 5$, $\therefore 10n^2+4n+2 = O(n^2)$
    $10n^2+4n+2 >= n^2$, for all $n >= 1$, $\therefore 10n^2+4n+2 = \Omega(n^2)$
    $n^2 <= 10n^2+4n+2 <= 11n^2$, for all $n >= 5$, $\therefore 10n^2+4n+2 = \Theta(n^2)$
- $100n+6=O(n)$ /* $100n+6 \leq 101n$ 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 $n \geq 4$ */

Practical complexity (1)

- To get a feel for how the various functions grow with *n*, you are advised to study the following two figures very closely.

$\log n$	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

Practical complexity (2)



Practical complexity (3)

- The following figure gives the time needed by a 1 billion instructions per second computer to execute a program of complexity $f(n)$ instructions.

1.9 Times on a 1 billion instruction per second computer

	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

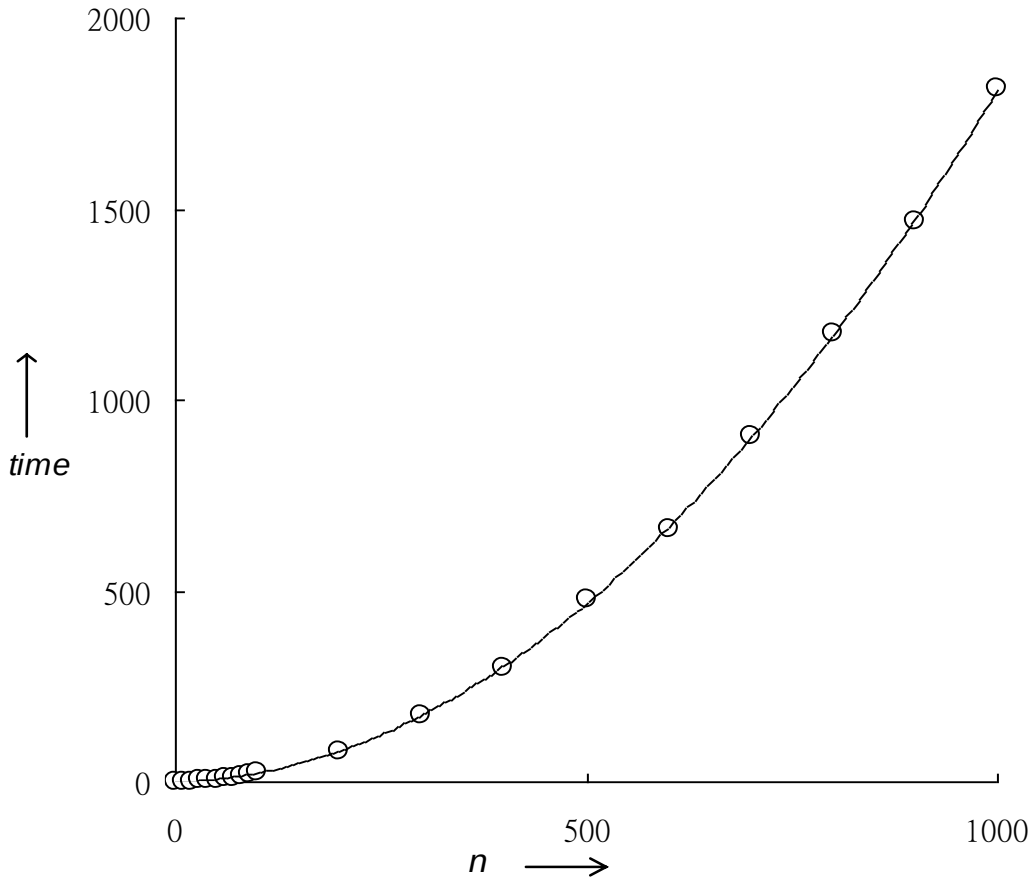
Practical complexity (4/4)

- Although performance analysis gives us a powerful tool for assessing an algorithm's space and time complexity, at some point we also must consider how the algorithm executes on our machine
  - This consideration moves us from the realm of analysis to that of measurement

	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);

Example

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



Worst case performance of the selection sort