1. Basic Concepts

h. choi

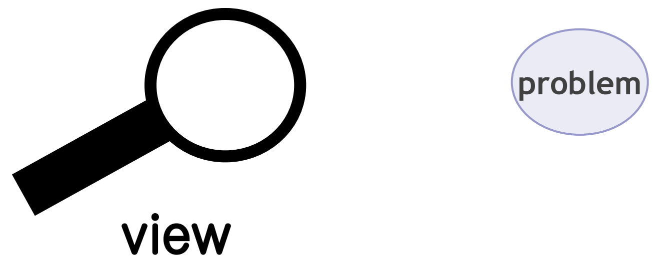
hchoi@handong.edu

Agenda

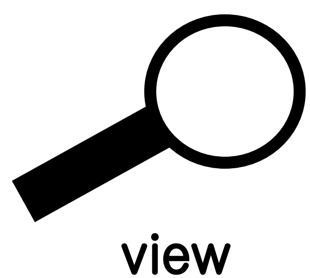
- System life cycle
- Pointers and dynamic memory allocation
- Algorithm specification
- Recursion
- Data abstraction
- Performance analysis

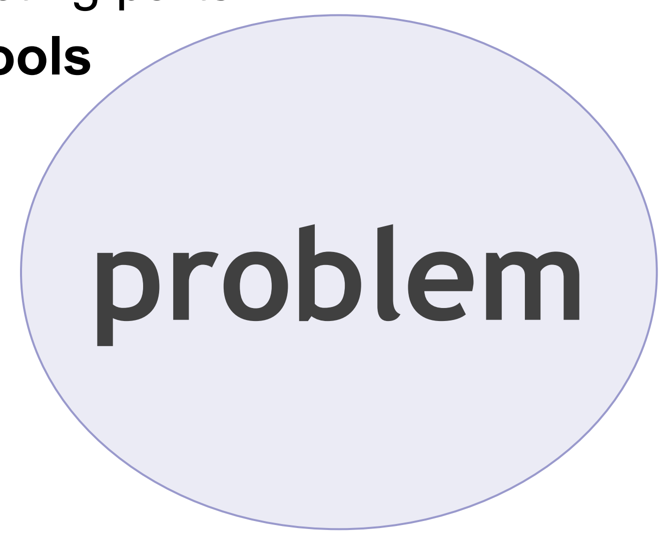
Overview

- Building a small program
  - Just do it !

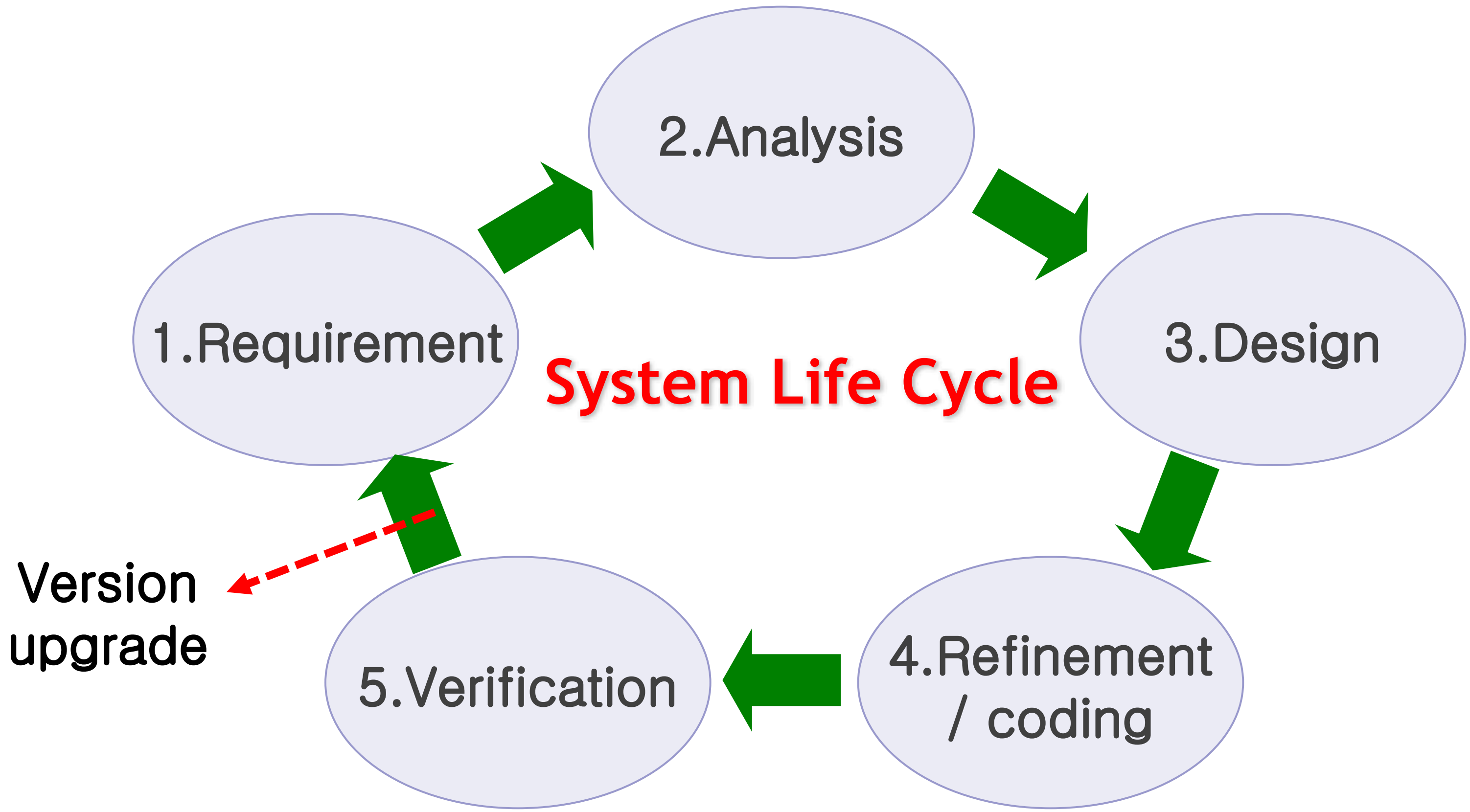


- Building a large-scale system
  - A system composed of complex interacting parts
  - Requires **systematic approach** and **tools**
    → **Objective of Data Structures**



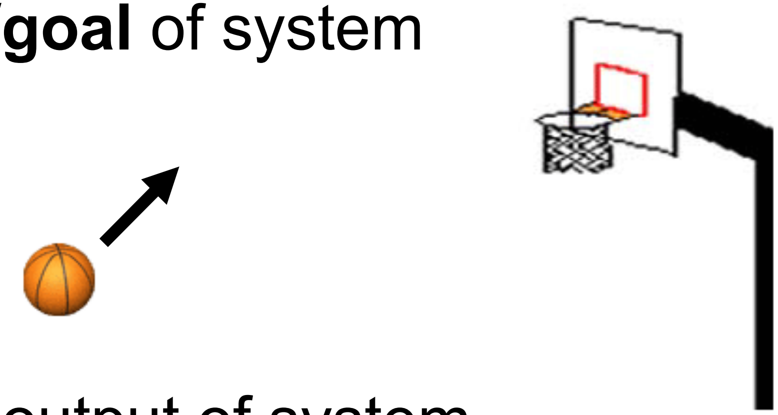


System Life Cycle

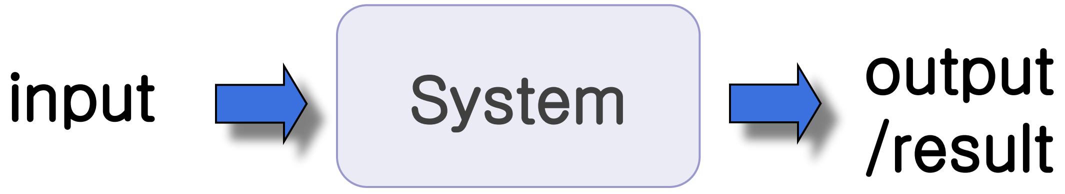


1. Requirement

- Define **purpose/goal** of system

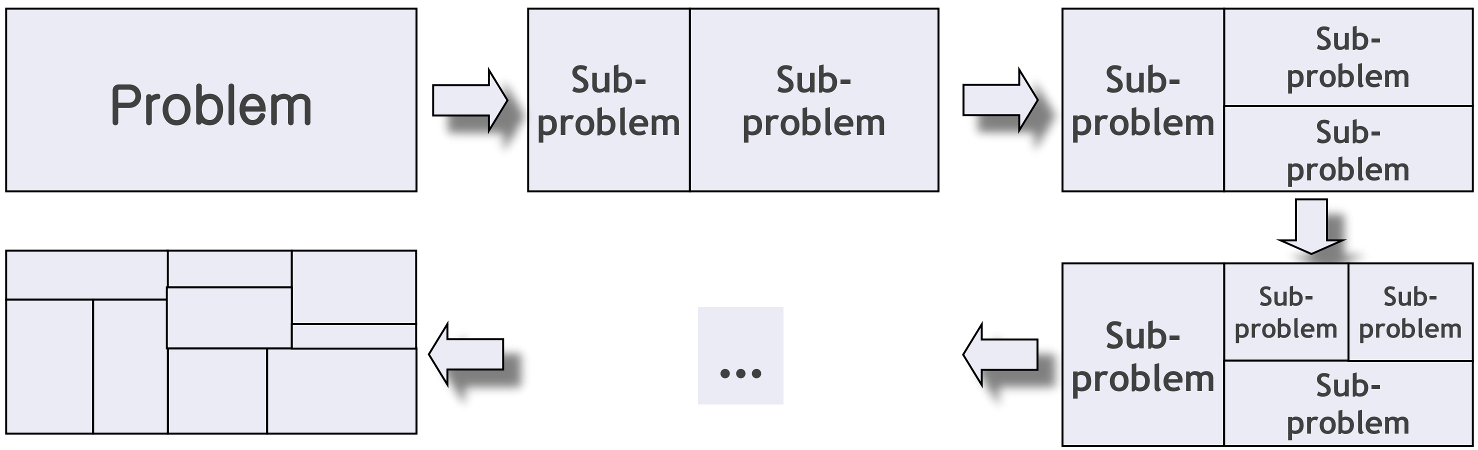


- Define input, output of system
  - Covers all cases
  - Definite/detailed description



2. Analysis

- Break down a problem into manageable pieces
  - Top-down approach: desirable
    - Purpose-driven approach



< Broken into manageable pieces >

cf. Bottom-up approach: old, unstructured strategy

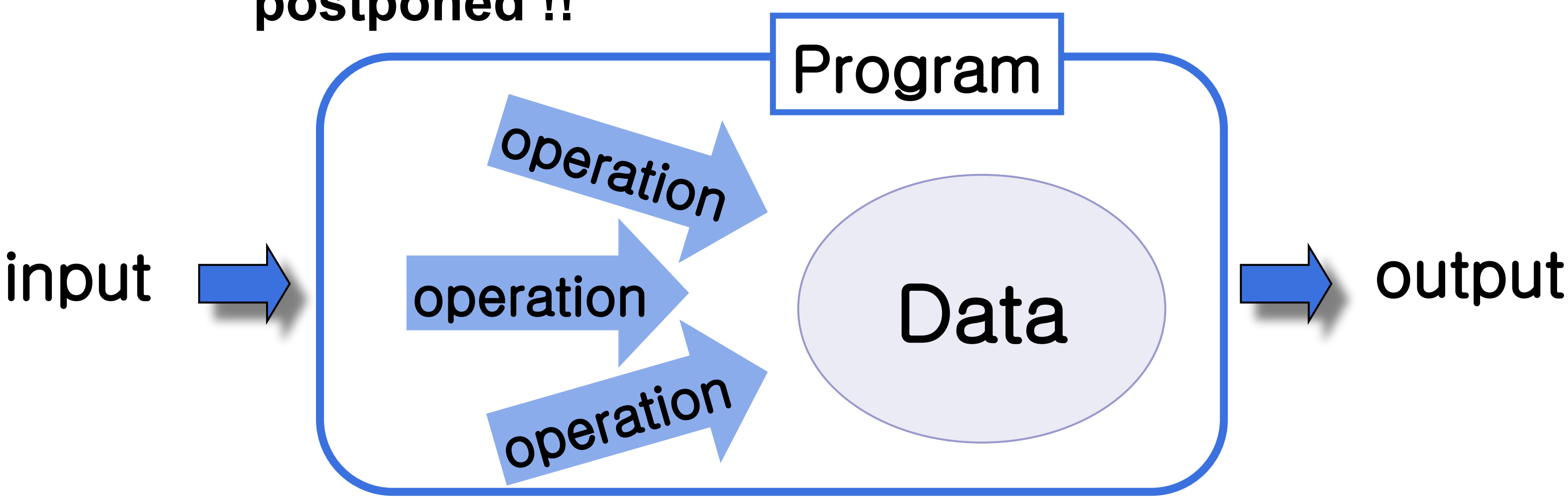
- Building from general walls, roof, plumbing, heating

3. Design

- Find solution from perspective of **data objects** and **operations** on them
  - Data objects: ADT (Abstract Data Type)
  - Operations: **specification** of algorithm

  → If the problem is broken-down into manageable pieces, design is easy.

  Note: Language dependent, implementation decisions are postponed !!



4. Refinement and Coding

- Implementation
  - Actual representations for data objects
  - Algorithms for each operation

  → data representation first, and then algorithms

  - Refinement with data representations and algorithms
  - Coding

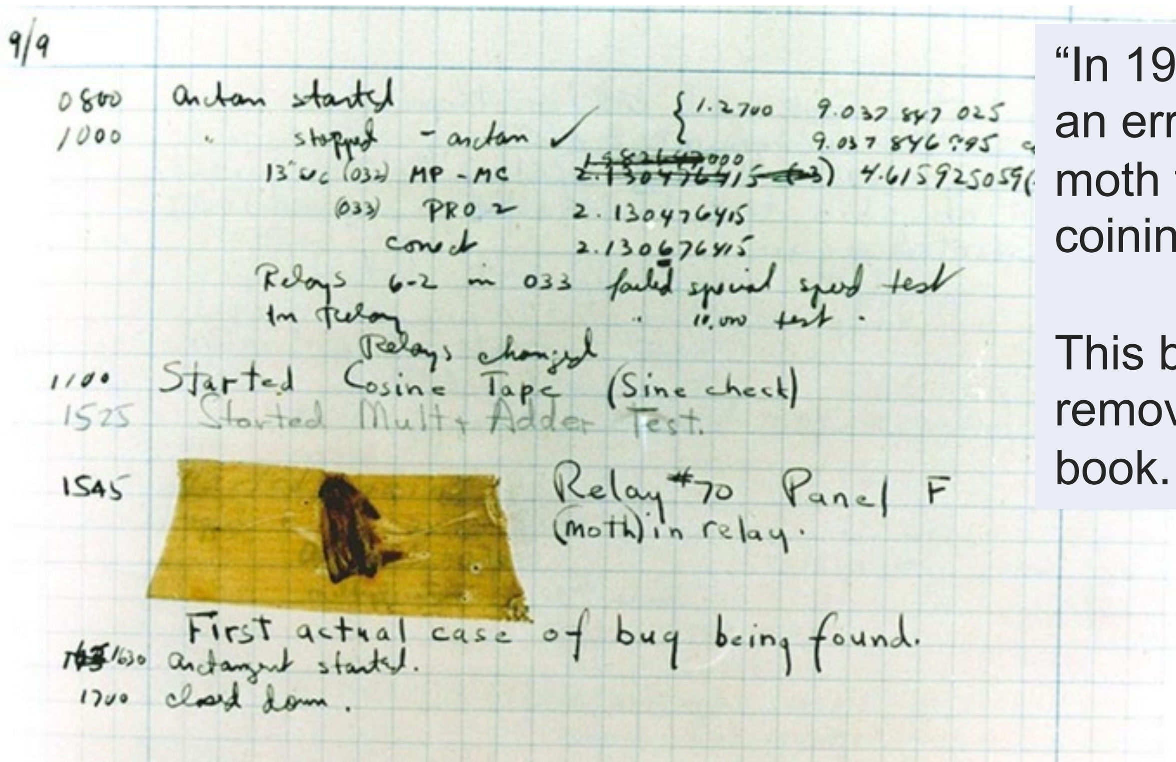
- Backup, and backup!
- If possible, use a version control program like github.

- Leave some comments and documentations

5. Verification

- Checking validity of system
  - Correctness proof (mathematical) → usually, very difficult
    - Employing proved algorithms can reduce the number of errors
  - Testing
    - with working code and **well-developed test data**
    - running time is another issue.
  - Error removal
    - Documentation, well-divided structure are very helpful
    - it is hard to find a tiny little thing from spaghetti
- Verification is very important for industry-strong code
  - TDD (Test-Driven Development)
    - Define test data before start to develop a system
    - A new trend of software development

"bug"



"In 1946, … Operators traced an error in the Mark II to a moth trapped in a relay, coining the term *bug*.

This bug was carefully removed and taped to the log book. "

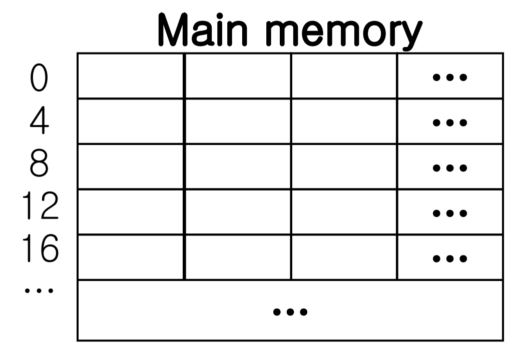
"Stemming from the first bug,
today we call errors or glitches in a program a *bug*" (from wiki)

Agenda

- System life cycle
- **<u>Pointers and dynamic memory allocation</u>**
- Algorithm specification
- Recursion
- Data abstraction
- Performance analysis

Memory and Variables

- Main memory
  - **List of cells** to store data or instruction
  - Each cell is identified by its **address**.

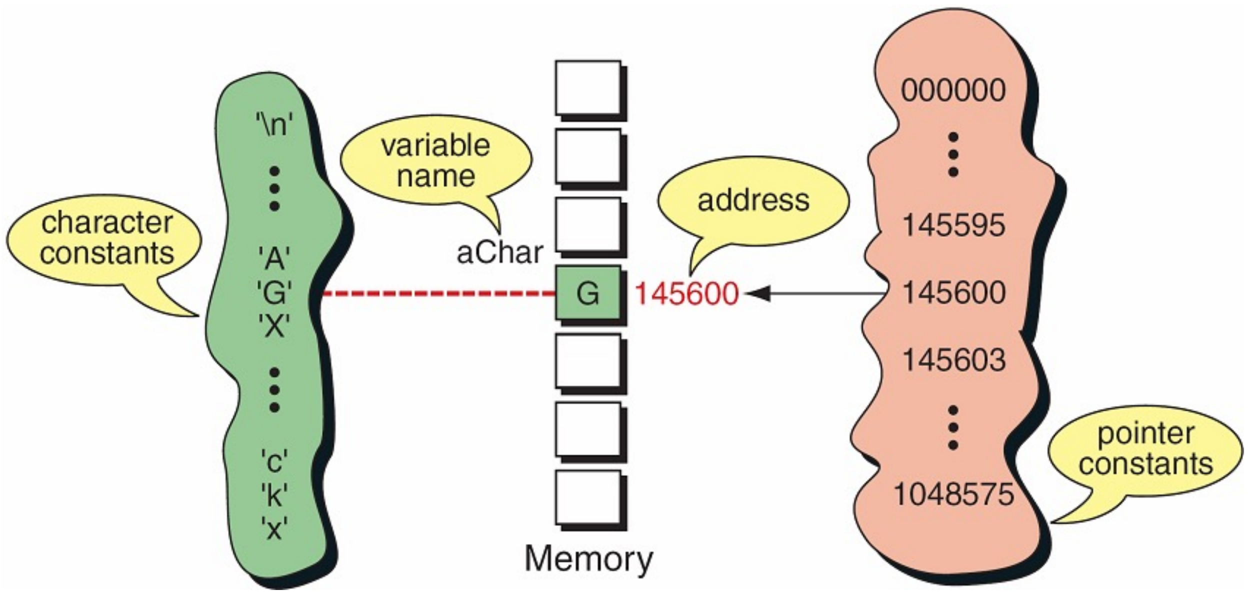


- Variable
  - A variable is a block of memory with a symbolic name
    Ex) int a = 100;
  - A variable has name, value(content), and **address**
  - A variable can store a piece of data of a particular type.

Pointers

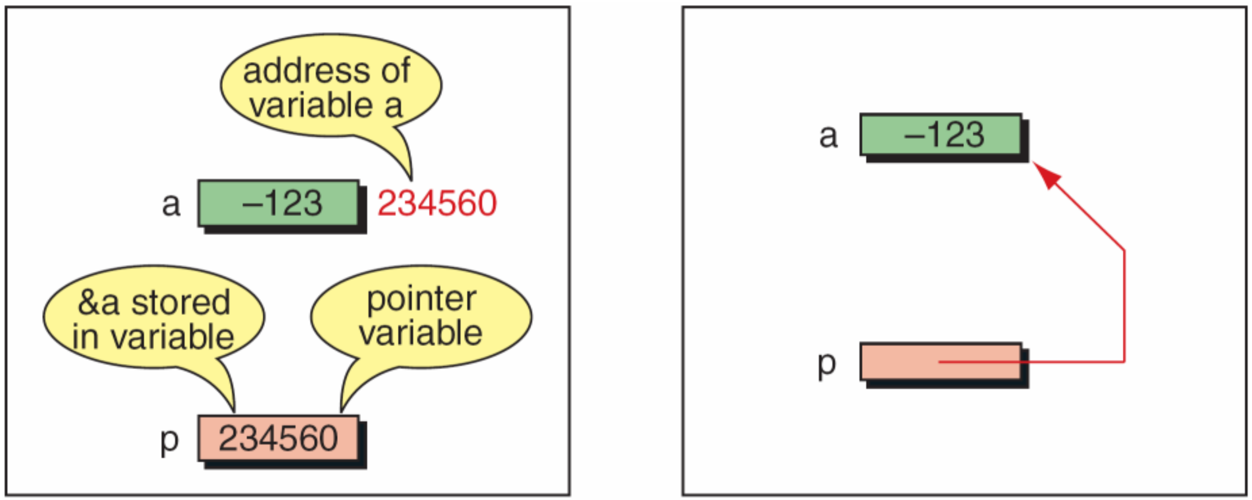
- Pointer: constant or variable that contains an address that can be used to access data
  - Range of pointer: address space of computer

    Ex) char aChar = ‘G’;   // assume &aChar == 1465600



Pointer Variables

- Pointer variable: a variable to store an address



Physical representation

Logical representation

Using Pointer Variables

- Declaration
  - int *pa;
- Extracting address of a variable (address operator &)
  - pa = &a;
- Dereferencing (dereferencing operator *)
  - *pa = 89;
  - c = *pa * 2;
- Address operator vs. dereferencing operator
  - & is inverse of *

    Ex) *&a $\equiv$ a;   // * and & cancel each other

    cf. How about &*a ?

- Arithmetic operations on pointers
  - addition, subtraction, multiplication, and division

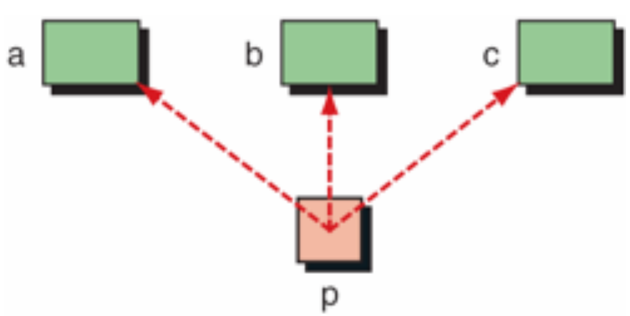
Flexibility of Pointer

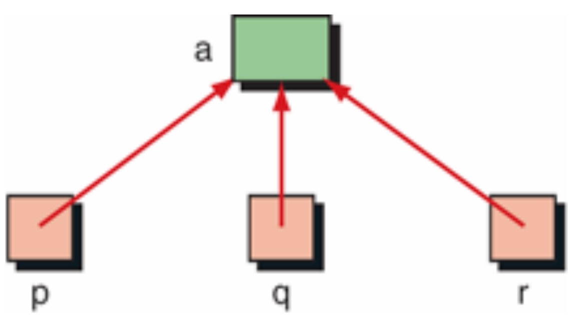
- Pointing different variables

```
int a = 10, b = 20, c = 30;
int *p;

p = &a;
printf("*p = %d\n", *p);
p = &b;
printf("*p = %d\n", *p);
p = &c;
printf("*p = %d\n", *p);
```

- Multiple pointers for a variables

```
int a = 10;
int *p, *q, *r;

p = q = r = &a;

printf("*p = %d\n", *p);
printf("*q = %d\n", *q);
printf("*r = %d\n", *r);
```

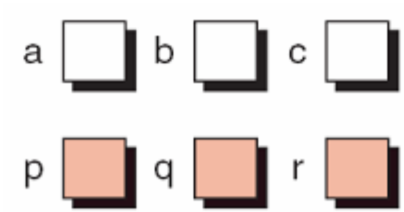




Example

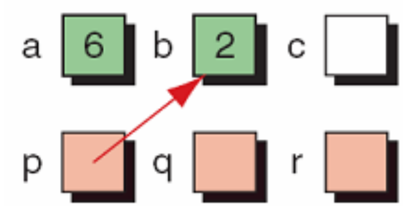
```
int a, b, c;
```

```
int *p, *q, *r;
```

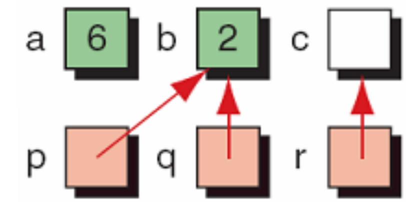



```
a = 6; b = 2;
```

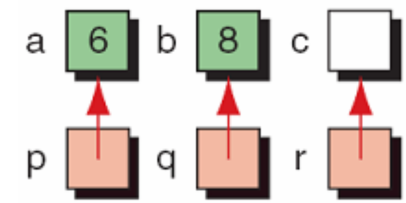
```
p = &b;
```



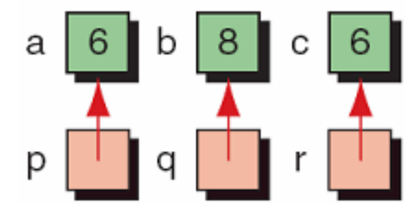

```
q = p; r = &c;
```



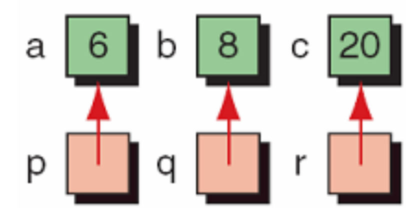

```
p = &a; *q = 8;
```




```
*r = *p;
```




```
*r = a + *q + *&c;
```

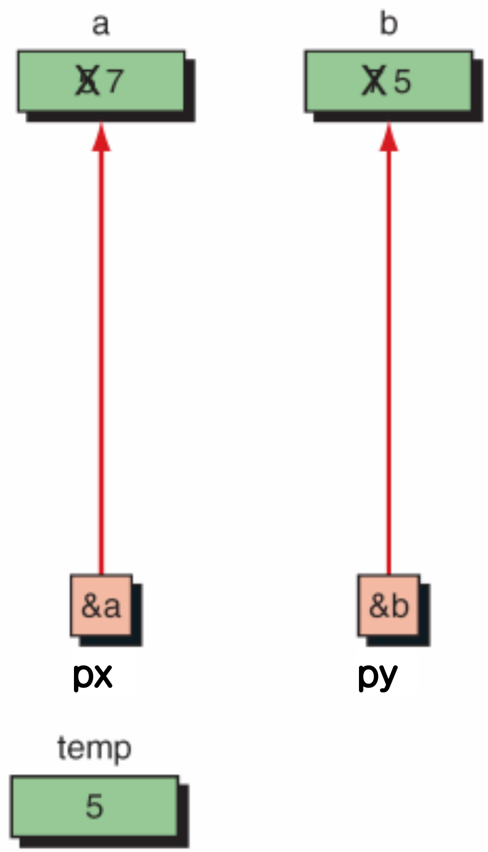


Pointers for Inter-Function Communication

- Passing addresses

```
// Function Declaration
void exchange (int*, int*);

int main (void)
{
  int a = 5;
  int b = 7;

  exchange (&a, &b);
  printf("%d %d\n", a, b);
  return 0;
} // main
```

```
void exchange (int* px, int* py)
{
  int temp;

  temp = *px;
  *px  = *py;
  *py  = temp;
  return;
} // exchange
```

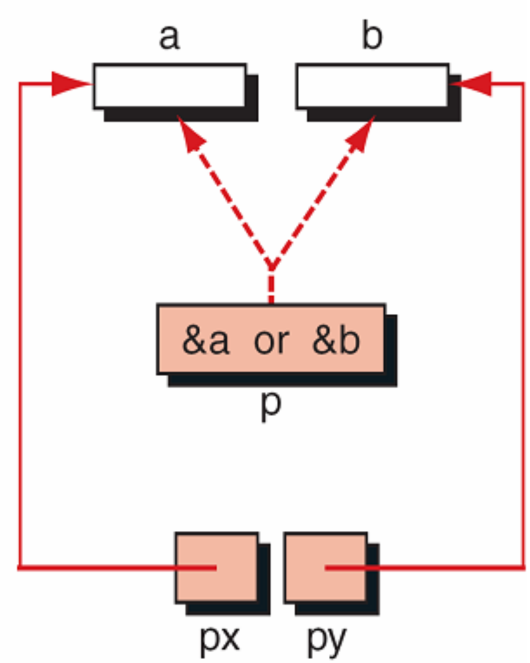


Pointers for Inter-Function Communication

- Functions returning pointers

```c
// Prototype Declarations
int* smaller (int* p1, int* p2);

int main (void)
  …
  int  a;
  int  b;
  int* p;
  …
  scanf ( "%d %d", &a, &b );
  p = smaller (&a, &b);
  …
```

```c
int* smaller (int* px, int* py)
{
  return (*px < *py ? px : py);
} // smaller
```



example

- what is the output of the following code segment?
  assume that the keyboard inputs are ”1 2 3 4 5”

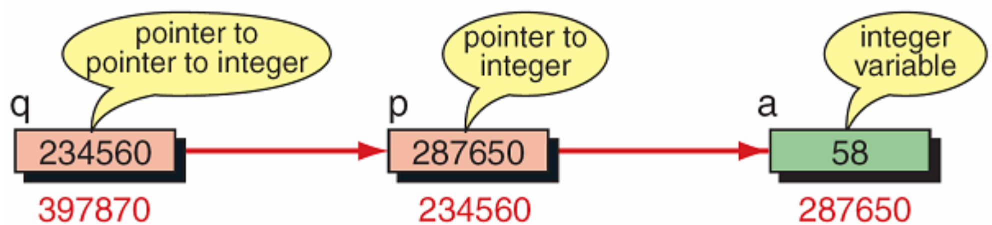
```
int main()
{
        int i, a[5];
        int *p=a;

        for(i=0; i<5; i++, p++)
                scanf("%d", p);

        for(i=0; i<5; i++)
                printf("a[%d]: %d\n", i, a[i]);

        return 0;
}
```

practice

Pointers to Pointers

- Pointer to pointer (double pointer): a pointer that points a pointer variable
  - Note! Pointer variable itself occupies memory space

```c
// Local Declarations
int    a;
int*   p;
int**  q;
```




```c
// Statements
a = 58;
p = &a;
q = &p;
printf(" %3d",   a);
printf(" %3d",  *p);
printf(" %3d", **q);
```

Example: Double Pointers

Exchange pointer variables

```
void ExchangePointers(int **pa, int **pb){
   int *temp = *pa;
   *pa = *pb;
   *pb = temp;
}

int main(){
   int a = 10, b = 20;
   int *p1 = &a, *p2 = &b;

   ExchangePointers(&p1, &p2);
   printf("*p1 = %d, *p2 = %d\n", *p1, *p2);
}
```

practice

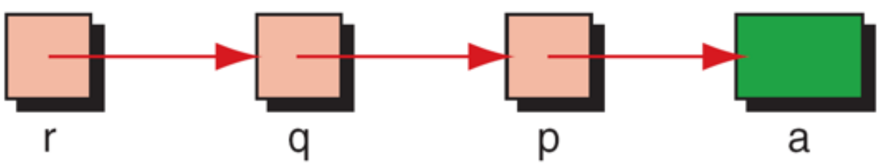
example (value, addr)

a: 10, 160	b: 20, 180
p1: 160, 240	p2: 180, 260
pa: 240, 400	pb: 260, 404

Pointers to Pointers

- Triple pointer

```
int a = 0;
int *p = &a;  // same with int *p; p = &a;
int **q = &p;
int ***r = &q;

// Note a ≡ *p ≡ **q ≡ ***r
```



Compatibility

- Pointer type compatibility
  - A pointer variable can store a pointer of the same type.

    Ex) char c, *pc;
    int a;
    pc = &c;   // no problem
    pc = &a;   // prohibited

- Pointer size compatibility
  - Although size of a variable vary with types, **size of all pointers are the same.**
    - int i, *pi;
    - char c, *pc;
    - float f, *pf;

    sizeof(i) $\neq$ sizeof(c) $\neq$ sizeof(f)

    sizeof(pi) = sizeof(pc) = sizeof(pf)

Pointer to Void

- void type pointer (void *) is just to store a generic address
  - A generic type that is not associated with a reference type

- void pointer can store any type of pointers

```
void *vp;
int a;
char c;
vp = &a;        // assigning integer pointer to vp
vp = &c;        // assigning character pointer to vp
```

- NULL pointer
  - **NULL** is defined by **(void*)0**, in stdio.h
  - Frequently used to initialize pointer variables

Pointer to Void

- void pointer cannot be dereferenced as it is

```
int a = 10;
void *pVoid = &a;
*pVoid = 10;  // illegal
```

  To be dereferenced, void pointer should be casted.

- void pointer can be dereferenced by casting

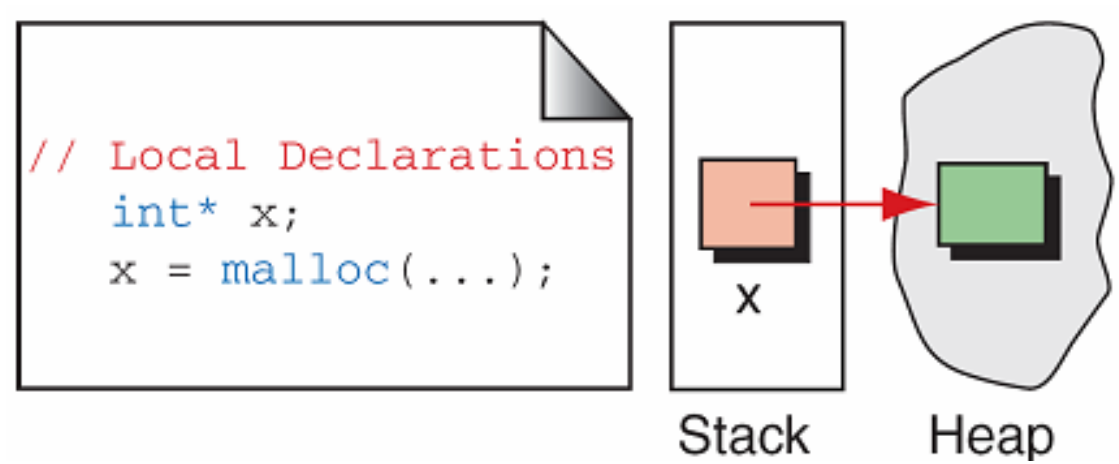
```
int a = 10;
void *pVoid = &a;
printf("*(int)pVoid = %d\n", *(int*)pVoid);
```

- Arithmetic operations are not available (+, -, [], …) **why not?**

```
vp = vp + 1;  // error
vp[2] = 0;    // error
```

Dynamic Memory Allocation

- Dynamic memory allocation: acquiring memory space to store information.
  - Memory allocation using predefined allocation functions
    - Size is dynamically determined
    - Allocated from heap



Dynamic Memory Allocation

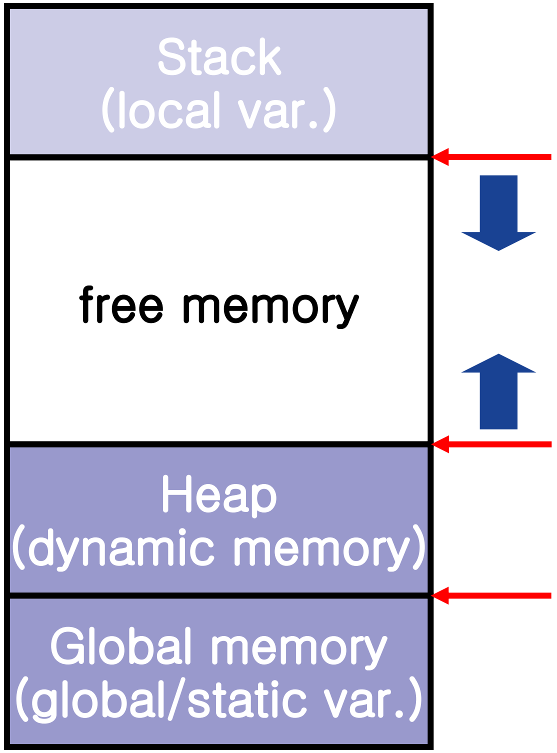
- Heap
  - Allocating memory
    - **malloc()** function
      - Size of memory block
  - Using memory
    - Address (pointer)
    - *** or [] operator**
  - Releasing memory
    - **free()** function
      - Address of the memory block

- Bank
  - Getting a loan
    - Loan application form
      - Amount of money
  - Using money
    - Account number
    - cash card
  - Repaying loan
    - Repayment application form
      - Borrowed money

Dynamic Memory Allocation

- Dynamic memory allocation: request for memory space
  - Declared in malloc.h
  - Allocation: *void *__malloc__(size_t NBYTES);*
    - *Usually, size_t is defined as unsigned int*
  - Deallocation: *void __free__(void *APTR);*

```
int *pi;
float *pf;
pi = (int*)malloc(sizeof(int));
pf = (float*)malloc(sizeof(float));

*pi = 1024;    // use of pi, pf
*pf = 3.14;

free(pi);
free(pf);
```



Static vs. Dynamic Memory Allocation

- Singleton variable

  int a = 0;

  int *pi = &a;

- Array

  float fa[100];

  float *pfa = fa;

- Singleton variable

  int *pi = (int*)malloc(sizeof(int));

- Array

  float *pfa =
      (float*)malloc(size*sizeof(float));

Static vs. Dynamic Memory Allocation

- Structure

```
struct Time {
    int hour, min, sec;
};

struct Time curTime;
struct Time *pCurTime =
    &curTime;

pCurTime->hour = 10;
```

- Structure

```
struct Time {
    int hour, min, sec;
};

struct Time *pCurTime = (struct Time*)
            malloc(sizeof(struct Time));

pCurTime->hour = 10;
```

Dynamic Memory Allocation

- Dynamically allocated memory can be used for any purpose.
  - Size and type should be specified properly

Ex) A singleton variable

```
int *pi = (int*)malloc(sizeof(int));
```

Ex) An array

```
float *pfa = (float*)malloc(size*sizeof(float));
// similar to "float pfa[size];", but not the same
```

Ex) A structure variable

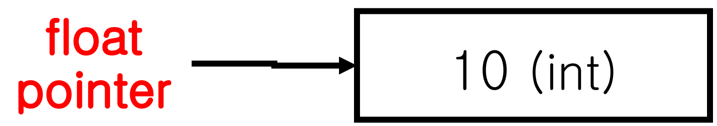
```
struct Time {                    // structure definition
    int hour, min, sec;
};
struct Time *pCurTime = (struct Time*)malloc(sizeof(struct Time));
pCurTime->hour = 10;             // same with (*pCurTime).hour = 10;
```

- **Dynamically allocated memory block must be deallocated eventually**

Invalid Use of Pointer

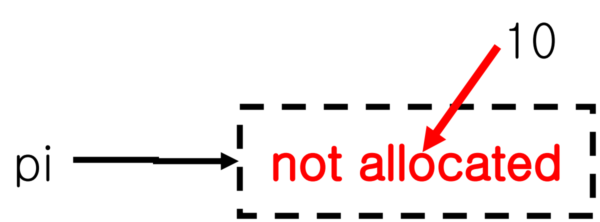
- Invalid type casting

```
Ex) int i = 10;
    int *pi = &i;
    float *pf = (float*) pi;        // semantic error
```



- Unassigned pointer

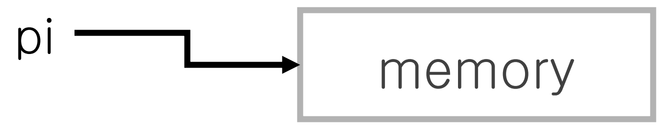
```
int *pi;
// pi = (int*) malloc(sizeof(int));        // forgot
*pi = 10;                                  // error
```



Invalid Use of Pointer

- Dangling pointer a pointer that points to dynamic memory that has been deallocated

```
int *pi = (int *)malloc(sizeof(int));
*pi = 10;        // valid use
…
free(pi);
…
```

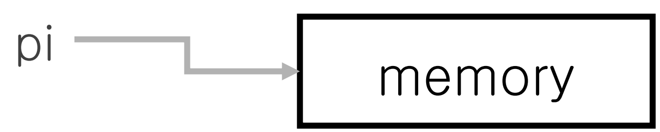


free(pi);   // error: pi is already deallocated

- Memory leak the loss of available memory space that occurs when dynamic data is allocated but never deallocated

```
{
    int *pi;
    pi = func(10);
    pi[0] = 10;
    …
    // free(pi);   // forgot
}
```

```
int *func(int len)
{
   int *a = malloc(len*sizeof(int));
   return a;
}
```



Safe Coding Practices

- Initialize every pointer at declaration

  Ex)

  ```
  int *pi;            // bad
  int *pi = NULL;     // good
  ```

- Set deallocated pointer variable by NULL

  ```
  free(pi);
  pi = NULL;          // free(NULL) is safe
  ```

string conversion

implement a function 'convert_case' to convert letter cases
ex) "Hello World" → "hELLO wORLD"

```
#include <string.h>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

#define MAX_LEN 1024
char* convert_case(char*);
int main()
{
    char aLine[MAX_LEN];
    char *p;

    fgets(aLine, MAX_LEN, stdin);

    p = convert_case(aLine);
    printf("%s\n", p);

    free(p);
    return 0;
}
```

practice

do not use any character array in convert_case()
→ use malloc()

HANDONG GLOBAL UNIVERSITY

Agenda

- System life cycle
- Pointers and dynamic memory allocation
- **Algorithm specification**
- Recursion
- Data abstraction
- Performance analysis

Algorithm Specification

- **Algorithm**: a finite set of instruction that accomplishes a particular task.
  - **Input**: zero or more quantities
  - **Output**: at least one quantity
  - **Definiteness**: clear and unambiguous instructions
  - **Finiteness**: for all cases, the algorithm terminates after a finite number of steps
    - Difference from program (but in this class, they are interchangeable)
  - **Effectiveness**: basic and feasible instructions

- Description of algorithm
  - Natural language
  - Programming language (C source code)
  - Etc.
    - Flow chart, pseudo code, …

Example

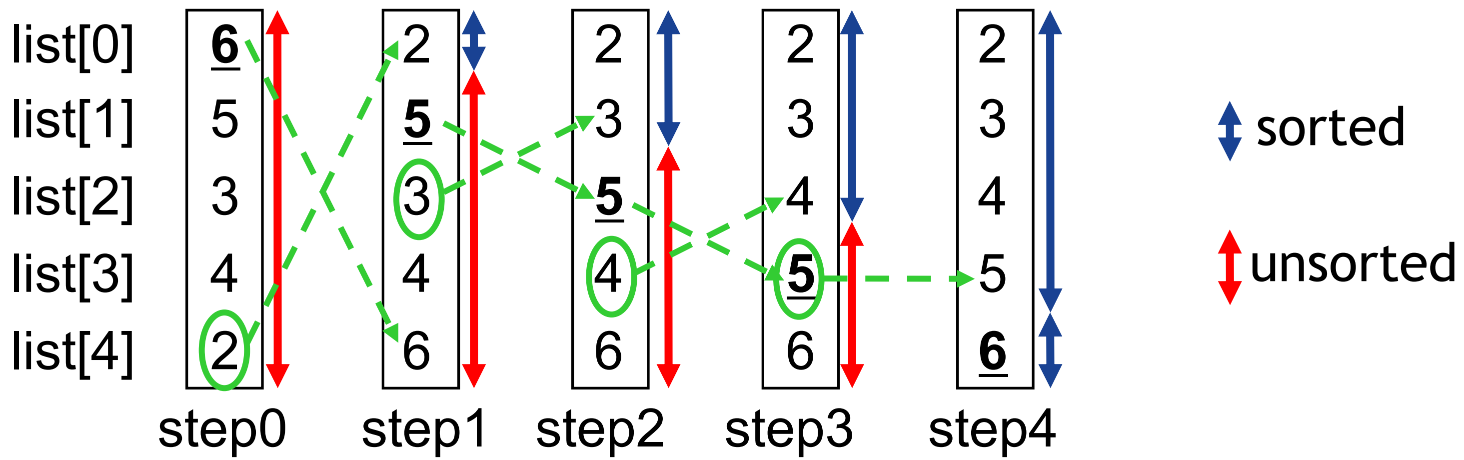
- How can one put an elephant in a refrigerator?
  - Open the fridge.
  - Push the elephant into the fridge. (effective?)
  - Close the door.

Example: Selection Sort

- Problem definition: sort *n* integers
- Simple solution
  - From those integers currently unsorted, find the smallest and place it next in the sorted list.

    → **Not clearly described.**

- Selection sorting



Example: Selection Sort

- Algorithm of selection sort

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

→ *Each step is clearly defined.*

Example: Selection Sort

- Implementation of selection sort in C language

```
void sort(int list[], int n)
{
    int i = 0, j = 0, min = 0, temp = 0;

    for(i = 0; i < n - 1; i++){
        // find the minimum of list[i] through list[n-1]
        min = i;
        for(j = i + 1; j < n; j++){
            if(list[j] < list[min])
                min = j;
        }

        // exchange list[i] and list[min]
        temp = list[i];
        list[i] = list[min];
        list[min] = temp;
    }
}
```

practice

```
swap(&list[i], &list[min]);

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

Example: Selection Sort

- how to practice?

```
#include <stdio.h>

void sort(int [], int);
int main()
{
    int aList[] = {0, 3, 1, 5, 7, 9, 2};

    sort(aList, 7);

    for(int i=0; i<7; i++)
        printf("%d ", aList[i]);

    return 0;
}
```

practice

Example: Selection Sort

- Prove that sort(list, n) works correctly.

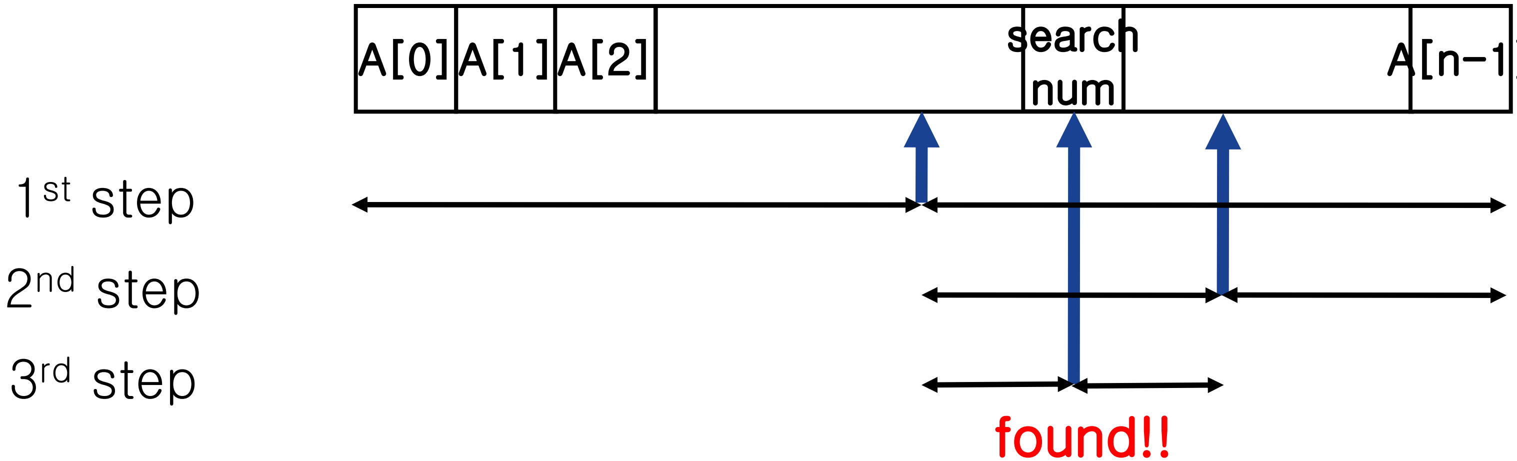
1. When the outer loop completes its iteration for i = q we have list[q] <= list[r], q < r < n.
2. On subsequent iterations, i > q, list[0] through list[q] are unchanged.

for any q between 0 and n−2

3. So, on the last iteration of the outer loop (i=n-2), we have list[0] <= list[1] <= … <= list[n-1].

Example: Binary Search

- Given
  - $n \geq 1$ distinct integers already sorted and stored in an array A.
    - $A[0] \leq A[1] \leq, \ldots, \leq A[n-1]$
  - An integer to find (searchnum)
- Problem: find an index, i, such thet A[i] = searchnum.
  - If searchnum is not present, return -1.



Example: Binary Search Algorithm

1. Let *left* and *right* denote the left and right ends of the list to be searched.
   - Initially, *left* = 0, *right* = *n* – 1
2. Let *middle* be the middle point of the list to be searched.
   - *middle* = (*left* + *right*) / 2
3. Compare *A[middle]* with *searchnum*
   - Case1: *searchnum* < *A[middle]*
     - If *searchnum* is present, it must be in the left half. [*left*, *middle* - 1]
     - Therefore, set *right* to *middle* - 1
   - Case2: *searchnum* == *A[middle]*
     - *Searchnum* is found. Return *middle*.
   - Case3: *searchnum* > *A[middle]*
     - If *searchnum* is present, it must be in the right half. [*middle* + 1, *right*]
     - Therefore, set *left* to *middle* + 1
4. If *left* <= *right*, go back to 2

Example: Binary Search Implementation

- Binary search in C

practice

```
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;
}
```

Example: Binary Search Implementation

practice

```
int main()
{
    int list[] = {1,2,3,4,5,6,7,10,11,17,20,23,25,29,31};
    int target, idx;                                   // 15 items
    scanf("%d", &target);

    idx = binsearch(list, target, 0, 14);

    if (idx < 0)
        printf("there is no target\n");
    else
        printf("the target is at %d\n", idx);
}
```

Example: Binary Search

- COMPARE: a function to compare x and y
  - If x < y, return -1
  - If x == y, return 0
  - If x > y, return +1

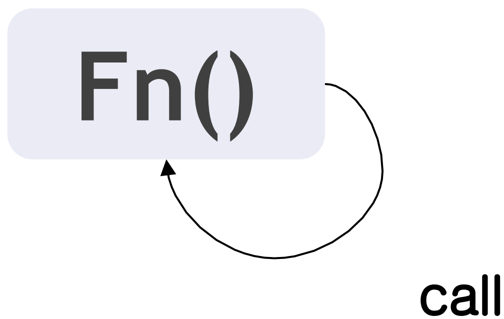
```
int COMPARE(int x, int y)
{
   if(x < y)
    return -1;
   else if(x == y)
    return 0;
   else
    return 1;
}
```

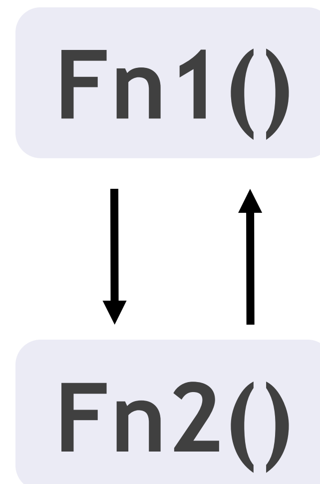
Agenda

- System life cycle
- Pointers and dynamic memory allocation
- Algorithm specification
- <u>**Recursion**</u>
- Data abstraction
- Performance analysis

Recursive Algorithm

- Recursion: a function call to itself





< direct recursion >

< indirect recursion >

- "Recursive definition" of recursion
  - Recursion:

    see ***Recursion***

Recursive Algorithm

- Why recursion?
  - Extremely powerful
  - Complex processes are often expressed in very clear terms with recursion.

- Example: Recursive implementation of Factorial (x!)

```
int Factorial(int x)
{
    if(x == 1)                    // termination condition
        return 1;
    else
        return x * Factorial(x-1);
}
```

- Theoretically, every iteration can be transformed to recursion and vice versa.

Recursive vs. Iterative Algorithm: Factorial

```
int Factorial(int n)
{ // recursive solution
  if (n==0) return 1;
  else
    return n*Factorial(n-1);
}
```

using a selection
with less local variable.

```
int Factorial(int n)
{ // iterative solution
  int factor, count;
  factor = 1;
  for(count=2;count <=n; count++)
    factor = factor * count;
  return factor;
}
```

using a loop
with more local variables.

Recursive Algorithm

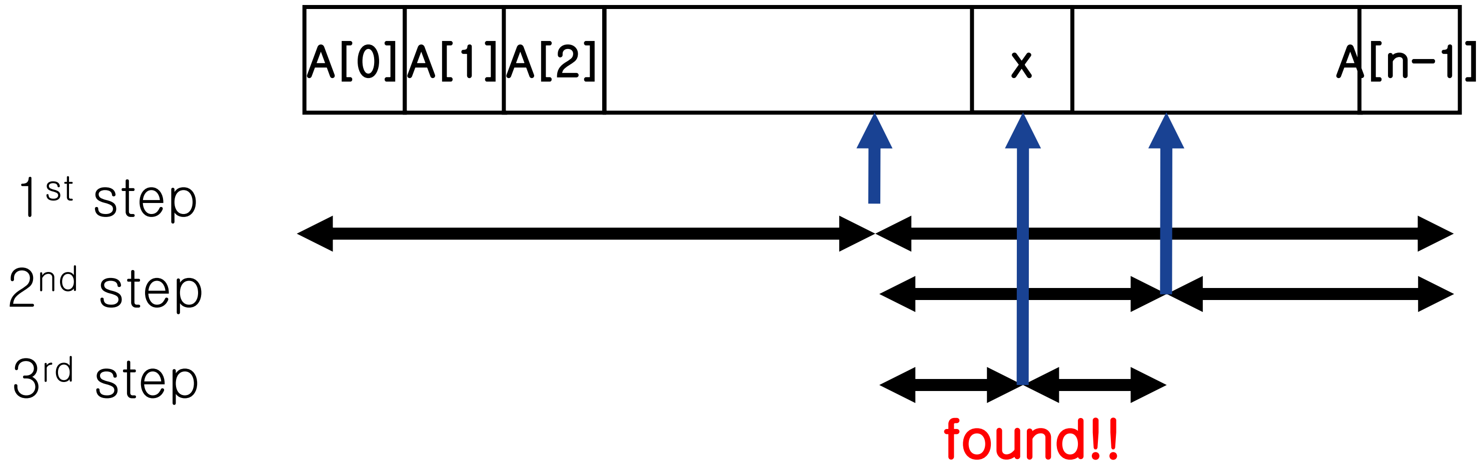
- Problems suitable for recursion
  - Recursively defined problems: problem that can be divided into the same problem with smaller size
    - Factorial
      - Factorial(x) = x*(x-1)*(x-2)*…*1 = x * Factorial(x-1)
    - Fibonachi numbers
      - Fibonachi(x) = Fibonachi(x-1) + Fibonachi(x-2)
      - Fibonachi(0) = Fibonachi(1) = 1 → termination condition
    - Binomial coefficients

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$$

$$\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1}$$

Example of Recursive Algorithm

- Binary search
  - Problem: find a number x from a sorted list A[]



Example of Recursive Algorithm

- Recursive algorithm for binary search

```
int binsearch(int A[], int x, int left, int right)
{
    if(left <= right){                          // termination condition 1
      int mid = (left + right) / 2;
      switch(COMPARE(A[mid], x)){
        case -1: return binsearch(A, x, mid+1, right);  // A[mid] < x
        case 0: return mid;         // termination cond. 2, A[mid] == x
        case +1: return binsearch(A, x, left, mid-1);   // A[mid] > x
      }
    }
    return -1;
}
```

practice.
compare this to the previous iterative version

An Example

- Given
  - An array A

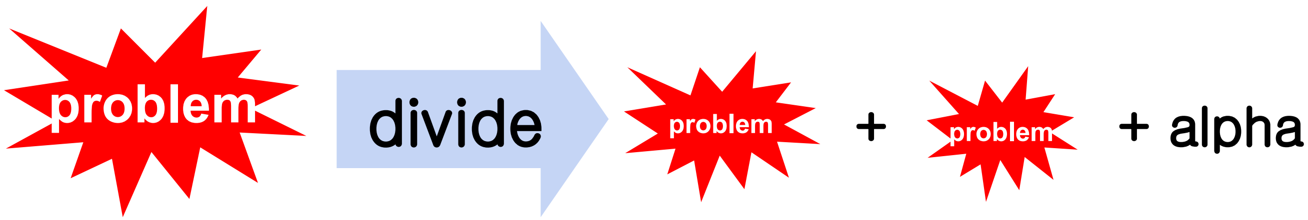
i	0	1	2	3	4	5	6	7	8
A[i]	2	5	6	8	9	13	15	30	54

  - A target number X = 8

- Procedure
  - Step1:
  - Step2:
  - Step3:

Design of Recursive Algorithm

1. Find a way to divide a problem into **sub-problems whose solution is the same** with the original problem with **reduced size**



2. Describe problem reduction algorithm with recursion general case

   Ex) F(n) = F(a) + F(b) + … + some other part

   - a, b, … should be smaller than n

3. Describe non-recursive solution(s) for very simple case(s) base case

   Ex) F(1), F(0)

   → termination

Example: permutation generator

- print out all possible permutations of the set (list)

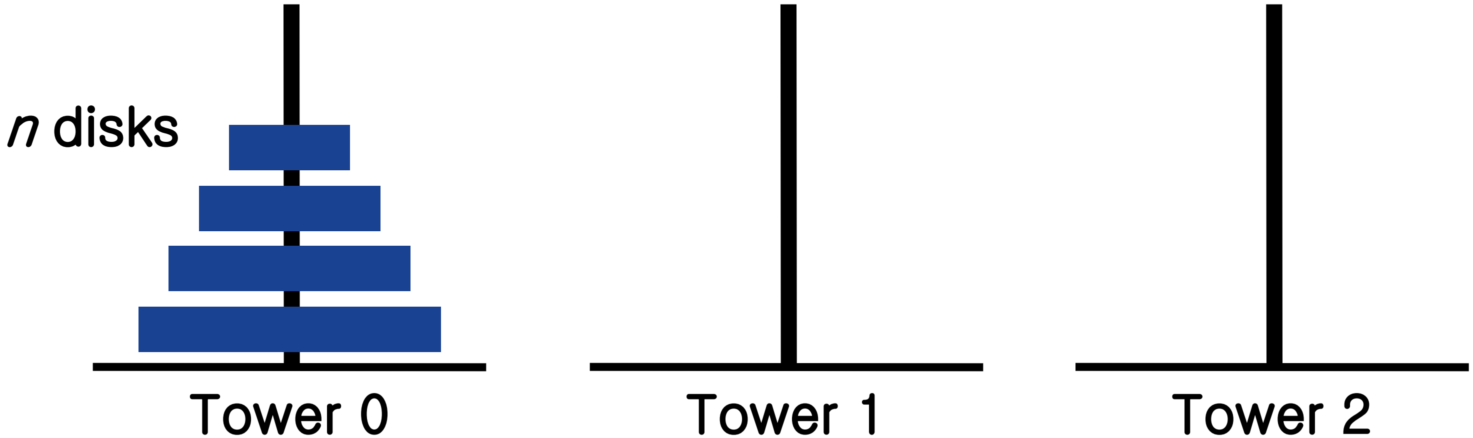
```
void perm(char *list, int i, int n)
{
   int j, temp;
   if (i == n){
     for (j=0; j<=n; j++)
        printf("%c", list[j]);
     printf("\n");
   }
   else{
     for (j=i; j<=n; j++){
        swap(&list[i], &list[j]);
        perm(list, i+1, n);
        swap(&list[j], &list[i]);
     }
   }
}
```

- initial call
`perm(list, 0, n-1);`

practice
understand how it works

Example: Tower of Hanoi

- Input
  - Three towers and ***n*** disks of different diameters
  - placed on the first tower in order of decreasing diameter
- Problem (mission)
  - Move all disks from the first tower to the third tower
- Restrictions
  - Only one disk can be moved at any time
  - No disk can be placed on top of a disk with a smaller diameter



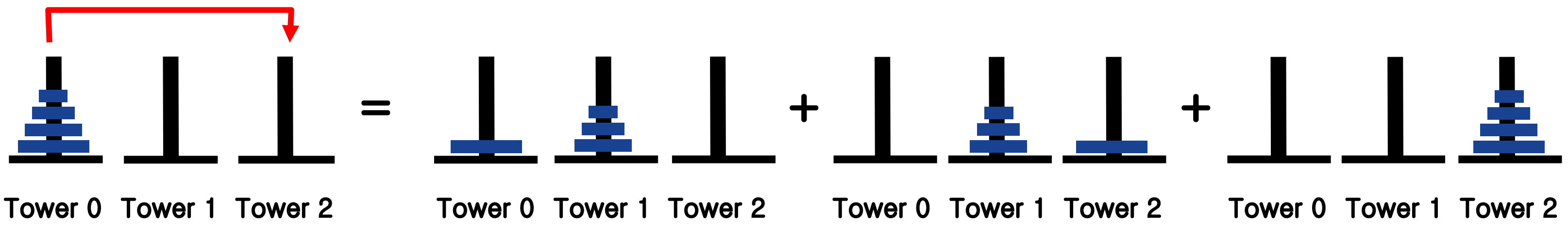
Towers of Hanoi

- Non-recursive solution: difficult and complex
- Solution using recursion
  - Divide the problem into smaller problems
    - Moving ***n*** disks from ***tower a*** to ***tower b*** =

      Move ***n-1*** disks from ***tower a*** to ***tower c***

      \+ Move ***1*** disk from ***tower a*** to ***tower b***

      \+ Move ***n-1*** disks from ***tower c*** to ***tower b***



  - Termination condition
    - If n == 1, just move a single disk from tower a to tower b

Towers of Hanoi

```
void Hanoi(int n, int begin, int aux, int end)
{
  if (n > 1)
  {
    Hanoi(n-1, begin, end, aux); // from begin to aux
    Hanoi(1, begin, aux, end);   // from begin to end
    Hanoi(n-1, aux, begin, end); // from aux to end
  }
  else
    printf("move the disk in %d to %d\n", begin, end);
}
```

practice
understand how it works

Agenda

- System life cycle
- Pointers and dynamic memory allocation
- Algorithm specification
- Recursion
- **Data abstraction**
- Performance analysis

Data Types

- Built-in data type of C/C++
  - Element type: char, int, float, double
  - Collection type: array, structure, …

    Ex)
    ```
    struct student {
            char last_name;
            int student_id;
            char grade;
    };
    ```
  - Pointer type: char*, int*, void*, …

- User defined type

Data Abstraction

- Data type: a collection of **objects** and a set of **operations** that act on those objects

  Ex) int type
  - Object: numbers in {INT_MIN, …, -1, 0, 1, …, INT_MAX}
    - 16bit integer
      - INT_MIN = -32768, INT_MAX = 32767
    - 32bit integer
      - INT_MIN = -2147483648, INT_MAX = 2147483647
  - Operation: +, -, *, /, %

Data Abstraction

- ADT (Abstract Data Type): data type organized by **specification of objects** and **specification of operation**, NOT including
  - Representation of objects
  - Implementation of operation

  - Focuses on what, not how
  - Necessary for managing large, complex software projects

- Specification of operation
  - Description of what the function does.
    - names, arguments, result of each functions
  - The function call depends on the function's specification (description), not its implementation (algorithm)
    - e.g., <u>double = sqrt(double)</u>, or <u>double = pow(double, double)</u>

Example of ADT

- Natural number (Nat_No)
  - Objects: integers from zero to INT_MAX
  - Functions (or operations)
    - Nat_No Zero() ::= return 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
    - Nat_No Subtract(x, y) ::= if(x < y) return 0;
      else return x – y;
    - Nat_No Power(x, y) ::= if (x^y < INT_MAX) return x^y
      else return INT_MAX
    - etc

ADT

- Why ADT?
  - Implementation-independent
- ADT frequently includes
  - Creator/constructor: create new instance
  - Transformers: create new instance from one or more other instances
  - Observers/reporters: provides information about instance
  - Destructor: discard instance of ADT (optional)
- we will discuss the specification first, then implementation

Agenda

- System life cycle
- Pointers and dynamic memory allocation
- Algorithm specification
- Recursion
- Data abstraction
- **Performance analysis**

Performance Analysis

- Criteria to evaluate a program
  - Does it meet specification of the task?
  - Does it work correctly?
  - Does it contain documentation about how to use and how it works?
  - Does it effectively use functions to create local units?
  - Is the code readable?

  - **Does it efficiently use primary/secondary storage?**
  - **Is the running time acceptable?**

Performance issues

Performance Evaluation

- Performance analysis
  - Mathematical analysis of algorithm
    - Complexity theory
    - Machine independent
- Performance measurement
  - Execution time on a specific computer
    - Machine-dependent

Complexities

- Space complexity of a program:
  - the amount of memory that it needs to run to completion
- Time complexity of a program:
  - the amount of computer time that it needs to run to completion

Space Complexity

- Fixed space requirements
  - Space requirement independent from number and size of input/output
    - Composed of instruction space, simple var., structures, constants

- Variable space requirements
  - Space requirement dependent on a particular instance $\boldsymbol{I}$
    - $S_p(I)$ - Variable space requirement of *a program* ***P*** working on *an instance* ***I***
      - $S_p(I)$ is usually a function of ***characteristics*** of $\boldsymbol{I}$
        - the number, size, values of input and output

        Ex) input is an array whose size is ***n***: *instance characteristic*
    - $S_p(I)$ can be replaced by $S_p(n)$ when $S_p(I)$ depends on only ***n***

- Total space requirement $S(P) = c + S_p(I)$

Examples

- Fixed space requirement

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

$S_{abc}(I) = 0$

- Variable space requirement

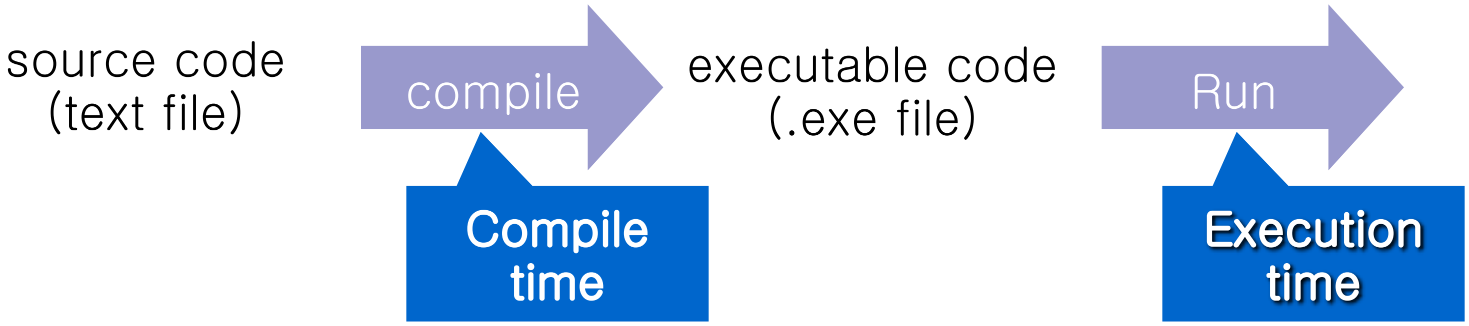
```
float sum(float list[], int n)
{
    float tempsum = 0;
    int i;
    for(i = 0; i < n; i++)
      tempsum += list[i];
    return tempsum
}
```

pass by value
pass by reference

$S_{sum}(I) = S_{sum}(n) =$ (***n in some languages like Pascal) (0 in C)***

Time Complexity

- Total time taken by a program P
  - T(P) = compile time + run time (execution time)
    - Compile time is not very important
      - Independent from instance characteristics
      - After development is finished, no need for recompile.
    - **Run time (T_p)**
      - **Major target of complexity analysis**



Time Complexity

- However, T_p does not depend only on P

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

    - c_a, c_s, c_l, c_{st}: constants

  - Requires knowledge about compiler and H/W

  For more details,
  computer architecture class

  - Rarely worthy

- Program step: alternative unit to measure execution time
  - Syntactically or semantically meaningful program segment,
  - whose **execute time is independent of the instance characteristics**

  Ex) Assignment , comparison, addition, …

  a = 2

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

  Both assignments are just one step

Measuring # of Steps Using Step Count Variable

- Iterative summing of a List of Numbers

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

- Iterative version with count 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;
}
```

- Number of executed steps: $2n + 3$

Measuring # of Steps Using Step Count Variable

- Recursive version with count statements

```
float rsum(float list[], int n)
{
   count++;        /* for if conditional */
   if (n > 1) {
      count++;  /* for return and rsum invocation */
      return rsum(list, n-1) + list[n -1];
   }
   count++;
   return list[0];
}
```

- Number of executed steps: $2n + 2$
- However, recursive version is usually slower than iterative version because of function-call overhead.

Measuring # of Steps Using Step Count Variable

- Matix Addition

```
void add_mat (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++){          // rows + 1
     // count++;
     for(j=0; j < cols; j++){          // rows × (cols + 1)
      // count ++;
      c[i][j] = a[i][j] + b[i][j];     // rows * cols
      // count ++;
     }
     // count ++;
    }
    // count ++;
}
// Total = 2 × rows × cols + 2 × rows + 1
```

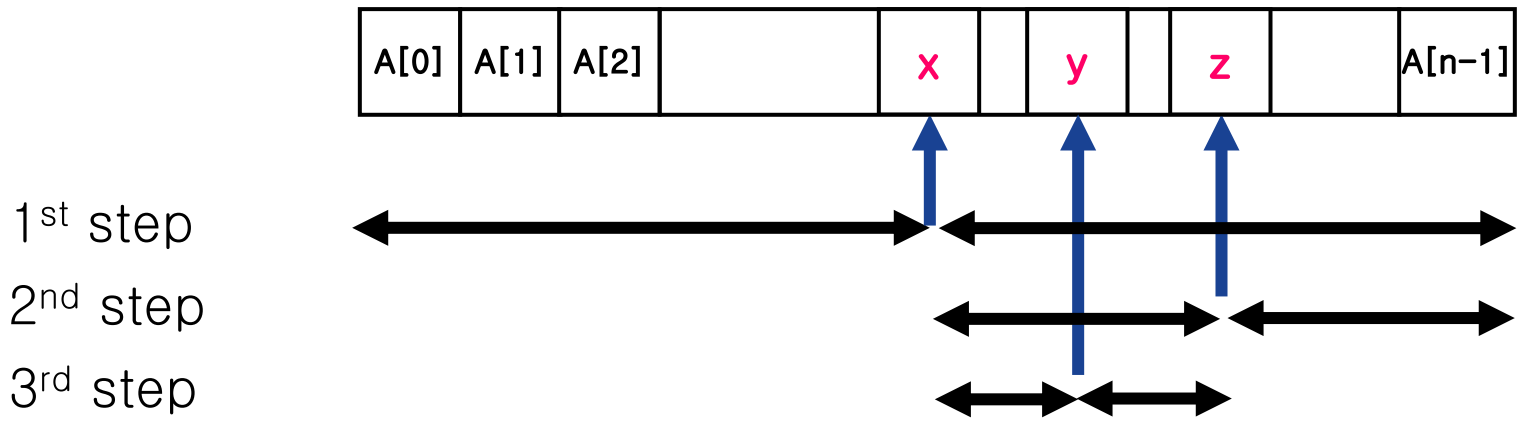
Measuring # of Steps Using Tabular Method

1. Determine **step count** for each statement (steps/execution or s/e)
2. Figure out number of times each statement is executed (**frequency**)
3. Multiply s/e by frequency to get total steps of each statement
4. Sum total steps of all statements

void add_mat (int a[ ] [MAX_SIZE], int b[ ][MAX_SIZE], int c[ ][MAX_SIZE], int rows, int cols)	s/e	Frequency	Total Step
{ 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*(cols+1)	rows*(cols+1)
c[I][j] = a[I][j] + b[I][j];	1	rows*cols	rows*cols
}	0	0	0

Three Kinds of Steps Counts

Ex) Step count of binarysearch depends on values of search target and contents of sorted list



- Step count should be estimated for three cases
  - Best case: minimum number of steps for execution
  - Worst case: maximum number of steps for execution
  - Average case: average number of steps for execution

Limits of Step Count

- Problems of step count for performance analysis
  - Increasingly difficult
  - Exact step count is not very necessary.
  - Step count itself is not exact
- Comparing programs
  - We can say 3n+3 is faster than 100n+10.
  - But, it is difficult to compare 80n +10 to 85n or 75n +20.
- So, we need asymptotic notations

Asymptotic Notation

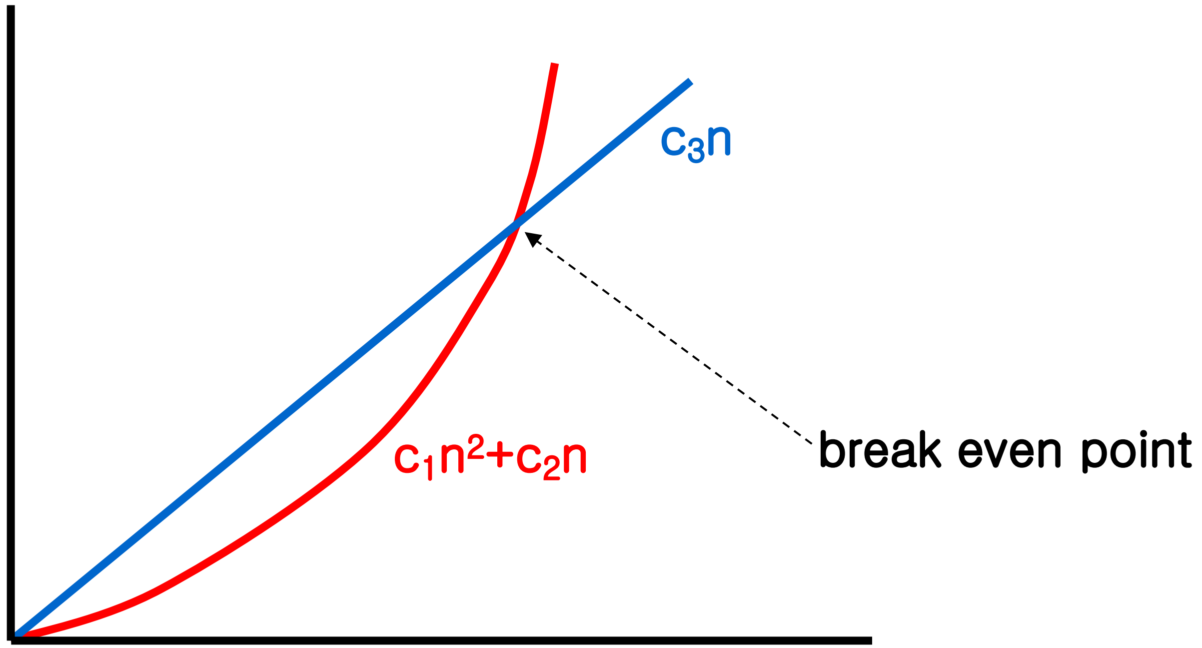
- More adequate method than step count
  - $c_1n^2 \le T_p(n) \le c_2n^2$ or $T_q(n,m) = c_1n+c_2m$
    - c_1, c_2 are non-negative constants

- We can compare $c_1n^2+c_2n$ with c_3n for sufficiently large n.
  - No mater how much c_3 is bigger than c_1 and c_2, $c_1n^2+c_2n > c_3n$ if n is very large

  Ex) $c_1 = 1$, $c_2 = 2$, $c_3 = 100$
    - for $n \le 98$, $c_1n^2+c_2n < c_3n$
    - for $n > 98$, $c_1n^2+c_2n > c_3n$

Asymptotic Notation

- Break even point: a value of n, beyond which c_3n is always faster than $c_1n^2+c_2n$, regardless of c_1, c_2, and c_3
  - Exact estimation of break even point is difficult and little advantage
  - Knowing whether a break even point exists is sufficient



Order of Complexities

- If the size of data is large, the order of complexity dominates the constant coefficient.

- Order of complexities
  - 3, 100, 35000, … → $O(1)$
  - $2\log_4(n)$, $6\log_2(n)$, $10\log_8(5n)$ → $O(\log n)$
  - $30n$, $65n+30$, $10000n + 329858$, … → $O(n)$
  - $9n^2$, $2n^2+100$, $582n^2+28$, … → $O(n^2)$
  - n^3+130, $9n^3+20$, $128n^3+32$, … → $O(n^3)$
  - 2^n, $5*2^n$, $100*2^n+n^3+100$, … → $O(2^n)$

Polynomial Complexities

- O(n): algorithms with single loops

```
for(i = 0; i < n; i++)
    sum += list[i];                          // executed n times
```

- O(n^2): algorithms with double loops

```
for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
      sum += list_2D[i][j];                  // executed n^2 times
```

- O(n^3): algorithms with triple loops

```
for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
      for(k = 0; k < n; k++)
         sum += list_3D[i][j][k];            // executed n^3 times
```

- O(n^m): algorithms with m^{th} order loop

Log Complexities

Ex) Worst case complexity of binary search

n	1	2	3	4	5	6	7	8	…	16	…	32	…
# of comp. (T)	1	2	2	3	3	3	3	4	4	5	5	6	…

- $n \approx 2^{(T-1)}$

→ $T \approx \log_2 n$

Exponential Complexities

Ex) # of possible patterns using n bits

n	0	1	2	3	4	5	6	7	8	9	10	11	12
# of possible patterns (T)	1	2	4	8	16	32	64	128	256	512	1024	2048	4096

- $T(n) \approx 2^n$

Asymptotic Notation

- Upper bound complexity

  Def) Big "Oh": $f(n) = O(g(n))$ iff there exist positive constants c and n_0 s.t. $f(n) \le cg(n)$ for all n, $n \ge n_0$

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

Note!
$3n+3=O(n)$
$3n+3=O(n^2)$
But,
$O(n) \neq O(n^2)$

Asymptotic Notation

- Big Oh notations
  - $O(1)$: constant
  - $O(n)$: linear
  - $O(n^2)$: quadratic
  - $O(n^3)$: cubic
  - $O(2^n)$: exponential

- Comparison

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

Note! $f(n) = O(g(n))$ doesn't mean $g(n) = O(f(n))$

Asymptotic Notation

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

- Proof

$$f(n) \le \Sigma^{m}_{i=0} |a_i|\, n^i$$

$$= n^m\, \Sigma^{m}_{0} |a_i|\, n^{i-m}$$

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

$$= c\, n^m, \text{ for } n \ge 1, \text{ where } c = \Sigma^{m}_{0} |a_i|.$$

So, $f(n) \le c\, n^m$ → $f(n) = O(n^m)$

For polynomial functions, simply find the largest degree

Asymptotic Notation

- Lower bound complexity

  Def) Omega: $f(n) = \Omega(g(n))$ iff there exist positive constant c and n_0 s.t. $f(n) \geq cg(n)$ for all n, $n \geq n_0$

  Examples

  - $3n+2 = \Omega(n)$
    - $3n+2 \geq 3n$ for $n \geq 1$
  - $10n^2+4n+2 = \Omega(n^2)$
    - $10n^2+4n+2 \geq n^2$ for $n \geq 1$
  - $6*2^n+n^2 = \Omega(2^n)$
  - $6*2^n+n^2 = \Omega(n)$
  - $6*2^n+n^2 = \Omega(1)$

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

Asymptotic Notation

- Lower and upper bound complexity

  Def) Theta: $f(n) = \Theta(g(n))$ iff there exist positive constant c_1, c_2 and n_0 s.t. $c_1 g(n) \le f(n) \le c_2 g(n)$ for all n, $n \ge n_0$

  Examples
  - $3n+2 = \Theta(n)$
    - $3n+2 \ge 3n$ for $n \ge$ **2** and $3n+2 \le 4n$ for all $n \ge 2$
  - $10n^2+4n+2 = \Theta(n^2)$
  - $6*2^n+n^2 = \Theta(2^n)$
  - $6*2^n+n^2 \ne \Theta(n)$
  - $6*2^n+n^2 \ne \Theta(1)$

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

Asymptotic Notation

- Example: Complexity of matrix addition

`void add(int a[][MAX_SIZE]..)`	0
`{`	0
`int i, j;`	0
`for(i=0; i<rows; i++)`	Θ (rows)
`for(j=0;j<cols;j++)`	Θ (rows*cols)
`c[i][j] = a[i][j]+b[i][j]`	Θ (rows*cols)
`}`	0
Total	Θ (rows*cols)

- Example: How about binsearch? log(n)

Practical Complexities

- Time complexity of a program:
  - A function of instance characteristics, e.g., f(n)
- Time complexity is useful in …
  - Determining how the time requirements vary as the instance characteristics change
  - Comparing two programs P and Q that perform the same task
    - If P has complexity Θ (n) and Q is of complexity Θ (n^2), we can assert that P is faster than Q for "sufficiently large" n

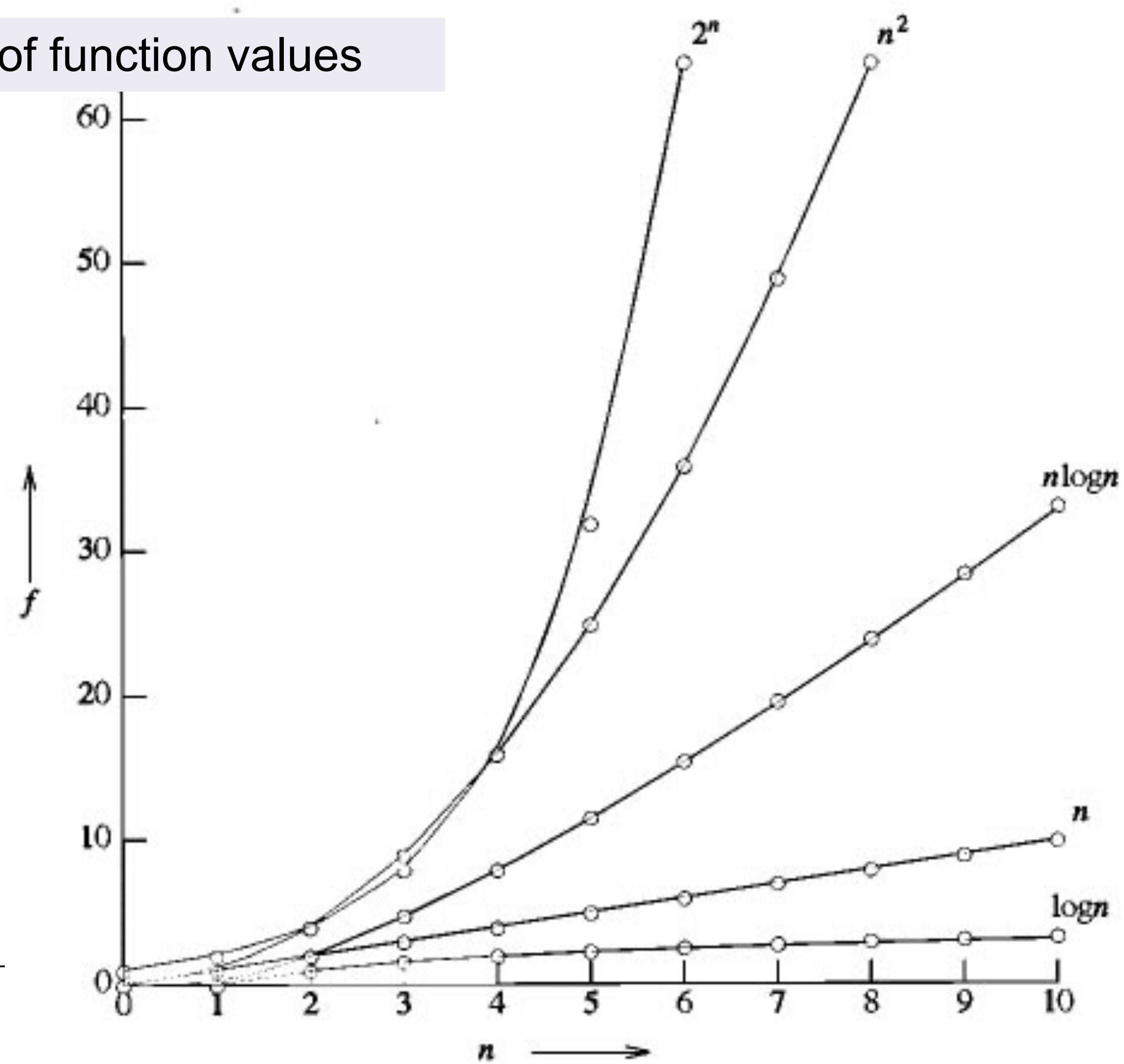
Practical Complexities

- Growth of function values

instance characteristic n								
time	name	1	2	4	8	16	32
1	constant	1	1	1	1	1	1
log n	logarithmic	0	1	2	3	4	5
n	linear	1	2	4	8	16	32
n log n	log linear	0	2	8	24	64	160
n^2	quadratic	1	4	16	64	256	1024
n^3	cubic	1	8	64	512	4096	32768
2^n	exponential	2	4	16	256	65536	4294967296
n!	factorial	1	2	24	40326	20922789888000	$26313*10^{33}$

Practical Complexities

Growth of function values



IG GLOBAL
ERSITY

Practical Complexities

- Time needed by a 1 billion instructions per second (GIPS) computer to execute a program of complexity f(n)

	Time for $f(n)$ instructions on a 10^9 instr/sec computer						
n	$f(n)=n$	$f(n)=n\log n$	$f(n)=n^2$	$f(n)=n^3$	$f(n)=n^4$	$f(n)=n^{10}$	$f(n)=2^n$
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	

Note that only programs of small complexity (such as n, n^2, n^3) are feasible (for reasonably large n, say n > 100).

Performance Measurement

- Performance measurement: measuring execution time on an actual machine
  - Measuring time using functions in C standard library (declared in time.h)

#include <time.h>

clock(): elapsed time since the program began
time(): elapsed time since Jan. 1, 1970

	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)
Remark	Internal processor time	Measured in second

*** note: exact syntax of functions and constant varies with systems**

Performance Measurement

- Example:

```
#include <time.h>
int main()
{
    clock_t start = 0, stop = 0;
    double duration = 0;
    start = clock();

    /// …
    // routines to measure execution time
    /// …

    stop = clock();
    duration = ((double) (stop – start) / CLOCKS_PER_SEC;
    return;
}
```

Performance Measurement

- Generating test data
  - Usually it is very difficult to generate worst-case data
  - Alternative way:
    - generate suitably large number of random test data
    - estimate the worst-case and the average-case.

questions or comments?

hchoi@handong.edu