1장 기본개념

1.1 개요: 시스템 생명주기 (시스템: large-scale program)

⇨ 시스템: Input + Processor + Output

⇨ 시스템 생명주기 (system life cycle): 프로그램 개발단계
(시스템 개발을 위한 일련의 단계)

(1) 요구사항 (requirements) 분석
- 문제에 대한 적절한 해를 구하기 위한 요구조건을 정의
- 프로젝트들의 목적을 정의한 명세(specification)들의 집합

(2) 명세 (specification)
- define "what the s/w is supposed to do"
- 기능적 명세 (functional specification)
  . 입력/출력 명세 - 구문적 제약조건 (비 절차적 명세)
  . 기능적 명세 - 의미적 제약조건 (절차적 또는 비절차적)

(3) 설계(design)
- 명세된 기능을 어떻게 달성하는가를 기술 (추상적 언어: Pseudo code)
- 전체 시스템 설계 : top-down, bottom-up
- 추상 자료형(abstract data type) - 자료객체와 연산
- 자료 객체(object)들과 수행될 연산(operation) 의 정의

(예: 수강등록 시스템 - student, course, 등을 object 라고 하고,
이 object 에 적용할 insert, remove 등이 operation 이다)

(4) 구현(implementation) - coding
- 구조화 프로그래밍 (assignment, conditional, loop)
- modular 프로그래밍 (function, entry/exit point)

(5) 검증(Verification)
- 테스트(testing): 프로그램의 수행 검증, 프로그램 성능 검사
  (모듈검사, 통합검사, 시스템 검사..)
- 정확성 증명(correctness proofs):
  수학적 기법들을 사용하여 프로그램의 정확성 증명
- debugging - 오류제거

(6) 운영 및 유지보수 (operation and maintenance)
- 시스템 설치(installation), 운영, 유지보수
- 새로운 요구조건 - 변경
- 수정 내용 기록 유지

1.2 소프트웨어 설계 방법론

(1) 하향식(top-down): 크고 복잡한 시스템 개발시 사용
- 문제들을 실제 다룰 수 있을 정도의 작은 단위들로 나눔
- 프로그램을 독립된 기능을 수행하는 작은 세그먼트로 분리

abstract program (initial solution)
↓ decomposition (refinement)
concrete program (executable by a target machine)

(2) 상향식(bottom-up): 하위레벨 부터 상세히 프로그램을 작성

Module 개발(하나의 기능수행)
↓. 작은 문제를 해결, 프로그래밍언어로 직접 개발, 하나의 기능을 수행

module 통합(시스템 통합)
- 전체 시스템: 하나의 완전한 프로그램

1.3 데이터 추상화 와 캡슐화

- 데이타 캡슐화(information hiding)
  - 외부에 대해 데이타 객체의 자세한 구현을 은폐
  - 내부 표현을 사용자에게 은폐 (예: VCR 의 play, ffw..)
  => 외부 세계로부터 데이터 객체의 자세한 구현을 은폐

- 데이타 추상화 (data abstraction)
  - 데이터객체의 명세와 구현을 분리
  - 무엇(what)과 어떻게(how)를 독립화
  => 데이터 객체의 명세와 구현을 분리

*** 추상화 와 캡슐화의 장점**

⑴ 소프트웨어 개발의 간소화: 복잡한 작업을 부분작업들로 분해
⑵ 검사와 디버깅의 단순화: 부분 작업
⑶ 재 사용성: 자료 구조가 시스템에서 별개의 개체로 구현

1.4 알고리즘 명세

알고리즘(Algorithm)의 정의: (An algorithm is a finite set of instructions that accomplishes a particular task)

. 특정한 일을 수행하기 위한 명령어의 유한 집합
. 동일한 문제에 여러 개의 알고리즘이 존재함

(예: 전화번호부 -> 무순서(순차적방법),
사전식배열(이진탐색)

조건 (criteria)

i. 입력(input): 0 or more are externally provided

ii. 출력 (Output) : 적어도 한개이상의 결과가 생성됨

iii. 명확성(definiteness) : 모호하지 않은 명확한 명령

iv. 유한성(finiteness) : 종료

v. 유효성(effectiveness) : 기본적, 실행가능 명령

ex. program ≒ algorithm

(알고리즘은 유한 단계를 거친 후 반드시 종료, 프로그램은 반드시 종료는 아님, 예: 운영체제는 실행할 job 이 없으면 대기상태로 감)

- Algorithm 기술방법
  - . 자연어 : (정확성 결여)
  - . flowchart : (명백성과 모호성의 결여)
  - . 프로그래밍 언어 : (문법상의 복잡성)
  - . 의사코드(pseudocode) : best way

예제1 [Selection Sort(선택 정렬)의 알고리즘 설계]

문제: n >= 1 개의 서로 다른 정수를 정렬하는 알고리즘을 설계하라

1) "*정렬되지 않은 정수들 중에서 가장 작은 값을 찾아서 정렬된 리스트 다음 자리에 놓는다*" ≠algorithm

2) (Pseudocode)

```
for (i = 0; i < n;  i++) {
    list[i]에서부터 list[n-1]까지의 정수 값을 검사한 결과
    list[min]이 가장 작은 정수 값이라 하자;
    list[i]와 list[min]을 서로 교환;
}
```

3) 실제 코드

```
void selectionSort(int list[], int n)
 {
   int i, j, min;
   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]);
  }
 }
```

[문제 #1] 다음 코드를 완성하고 실행결과를 보이시오

```
const int arraySize = 10;
void main(void)
{
  int list[] = {90, 60, 80, 100, 10, 40, 30, 70, 20, 50};
  int n;
   void selectionSort (int list[], int n);
   n = arraySize;
   dump(list, n);
   cout << "==============================" << endl;
   selectionSort(list, n);
}
void swap(int &a,  int &b)
{
  int temp = a;
  a = b;
  b = temp;
}
```

예제 2 [이진탐색] : 정수 searchnum이 배열 list에 있는지 검사

=> list[0] <= list[1] <= ... <= list[n-1] /* 미리 정렬되어 있음 */

=> list[i] = searchnum 인 경우 인덱스 i 를 반환, 없는 경우는 -1 반환

(초기 값 : left = 0, right = n-1;

list 의 중간 위치 : middle = (left + right) / 2)

* list[middle] 과 searchnum 비교 시 3 가지중 하나를 선택

1) searchnum < list[middle]: /* search again between left and moddile-1 */

2) searchnum = list[middle]: /* middle 을 반환 */

3) searchnum > list[middle]: /* search again between middle+1 and right */

- *code version 1*

```
int binarysearch(int list[], int searchnum, int left, int right)
{
int middle;
while (left <= right) {
      middle = (left + right) / 2;
      if (searchnum < list[middle])          right = middle - 1;
      else if (searchnum == list[middle])       return middle;
      else left = middle + 1;
}
```

- code version 2

```
int binsearch (int list[], int searchnum, int left, int right)
{
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;      break; }
```

```
  }
  return -1;
}

char compare (int x, int y)
{
  If (x > y) return 1;
  Else if (x < y) return -1;
  Else return 0;
}
```

[문제 #2] 다음 코드를 완성하고 실행결과를 보이시오

```
#include <iostream.h>

int binarySearch(int data[], int num, int left, int right);

void main()
{
    int data[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    int num, found;

    cout << "Enter an integer to search : ";
    cin >> num;
    found = binarySearch(data, num, 0, 9);

    if (found == -1)
            cout << "Not in the list" << endl;
    else
            cout << "Found at position " << found << endl;
}
```

* 순환 알고리즘 (Recursive Algorithm)

- 수행이 완료되기 전에 자기 자신을 다시 호출
- 순환 알고리즘의 장단점.
  . 장점 : 대단히 강력한(powerful) 알고리즘 표현 방법일 뿐 아니라 복잡한 알고리즘의 과정을 명료하게 표현 가능
  . 단점 : 비순환 알고리즘보다 시간(time)과 공간(space)면에 있어서 비효율적이다.

-순환 알고리즘의 구성
  . 반드시 순환호출을 끝내는 종료 조건이 있어야 한다.
  . 종료 조건에 접근하는 다음 단계의 순환호출이 있다.

*** Factorial function**
- 반복적 정의: **n! = n*(n-1)*(n-2) ……*2*1**
- 순환적 정의: **n! = 1 if n=1**
  n*n(-1)! if n>1

ex) recursive factorial

```
int factorial (int n)
{
    if (n == 0)   return 1     /* anchor (종료조건)*/
          else return
    (n*factorial(n-1));   /* recursive step(순환호출) */
}
```

3! = 3*2! = 3*2*1! = 3*2*1*0! (0! =1)

* Factorial 의 반복 프로그램 (non-recursive factorial)

```
int factorial(int n)
{
   int fact = 1;
   for (int i=0;   i<n;   i++)
        fact = fact*i;
   return fact;
 }
```

ex) 이진 탐색

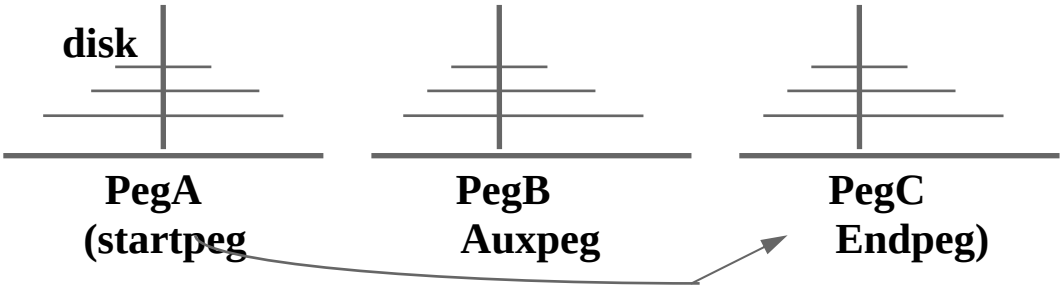
```
int binsearch(int list[], int searchnum, int left, int right)
{
  /* search list[0]    list[1]   ...    list[n-1] for searchnum*/

 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;
}
```

ex) Towers of Hanoi

$m_k = 2m_{k-1} + 1$ $(k >= 2)$, $m_1 = 1$ (k= 1)

then $m_n = 2^n - 1$ for n>= 1

requirement: 1) move one at a time 2) smaller disk on top of larger disk



- Algorithm: move n disk from startpeg to endpeg using auxpeg)

1) if only one disk, output “move disk from startpeg to endpeg”
2) else
    move (n-1 disk from startpeg to auxpeg using endpeg)
    output “Move disk from startpeg to endpeg”
    move (n-1 disk from auxpeg to endpeg using startpeg)

```
void towerHanoi (char from, char to, char aux, int n)
{
 if (n==1)
     printf(“move disk 1 from peg%c to peg%c\n”, from , to);

 else {
    towerHanoi(from, aux, to , n-1);
    printf(“move disk %d from peg %c to peg %c\n “, n, from, to);
    towerHanoi(aux, to, from, n-1);
    }
 }
 .............
  towerHanoi('A', 'C', 'B', num);
```