제 1 장
기본 개념

시스템 생명 주기(System Life Cycle)

- **요구사항(requirement)**
  - 프로젝트들의 목적을 정의한 명세들의 집합
  - 입력과 출력에 관한 정보를 기술

- **분석(analysis)**
  - 문제들을 다룰 수 있는 작은 단위들로 나눔
  - 상향식(bottom-up)/하향식(top-down) 접근 방법

- **설계(design)**
  - 추상 데이타 타입(abstract data type) 생성
  - 알고리즘 명세와 설계 기법 고려

시스템 생명 주기(System Life Cycle)

- **정제와 코딩(refinement and coding)**
  - 데이타 객체에 대한 표현 선택
  - 수행되는 연산에 대한 알고리즘 작성

- **검증(verification)**
  - 정확성 증명(correctness proof)
    - 수학적 기법들을 이용해서 증명
  - 테스트(testing)
    - 프로그램의 정확한 수행 검증
    - 프로그램의 성능 검사
  - 오류 제거(error removal)
    - 독립적 단위로 테스트 후 전체 시스템으로 통합

객체 지향 설계

- 구조적 프로그래밍 설계와의 비교
  - 유사점
    - 분할-정복 기법 : 복잡한 문제를 여러개의 단순한 부분 작업으로 나누어 각각을 개별적으로 해결
  - 차이점
    - 과제 분할 방법

알고리즘적 분해와 객체 지향적 분해

- 알고리즘적 분해(함수적 분해)
  - 고전적 프로그래밍 기법
  - 소프트웨어를 기능적 모듈로 분해
  - Pascal의 프로시저, FORTRAN의 서브프로그램, C의 함수

- 객체 지향적 분해
  - 응용 분야의 개체를 모델링하는 객체의 집합
  - 소프트웨어의 재사용성 조장
  - 변화에 유연한 소프트웨어 시스템
  - 직관적

객체 지향 프로그래밍의 기본 정의와 개념

- **객체(object)**
  - 계산을 수행하고 상태를 갖는 개체
  - 데이타 + 절차적 요소
- **객체 지향 프로그래밍(object-oriented programming)**
  - 객체는 기본적인 구성 단위(building block)
  - 각 객체는 어떤 타입(클래스)의 인스턴스(instance)
  - 클래스는 상속 (inheritance)관계로 연관됨
- **객체 지향 언어(object-oriented language)**
  - 객체 지원
  - 모든 객체는 클래스에 속함
  - 상속 지원
- **객체 기반 언어(object-based language)**
  - 객체와 클래스는 지원하되 상속은 지원하지 않는 언어

프로그래밍 언어의 발전과 C++의 역사

- 고급 프로그래밍 언어
  - 1세대 언어 - FORTRAN
    - 수식 계산 가능
  - 2세대 언어 - Pascal, C
    - 알고리즘을 효과적으로 표현
  - 3세대 언어 - Modula
    - 추상 데이타 타입 개념 도입
  - 4세대 언어(객체 지향 언어) - Smalltalk, Objective-C, C++
    - 상속 기능 지원

- C의 장점
  - 효율성(efficient) : 하드웨어를 직접 제어할 수 있는 기능
  - 유연성(flexible) : 대부분의 응용 분야에 사용 가능
  - 가용성(available) : 모든 컴퓨터에 C 컴파일러 존재

데이타 추상화와 캡슐화

- **데이타 캡슐화(data encapsulation)**
  - 정보 은닉(information hiding)
  - 외부로부터 데이타 객체의 자세한 구현을 은닉

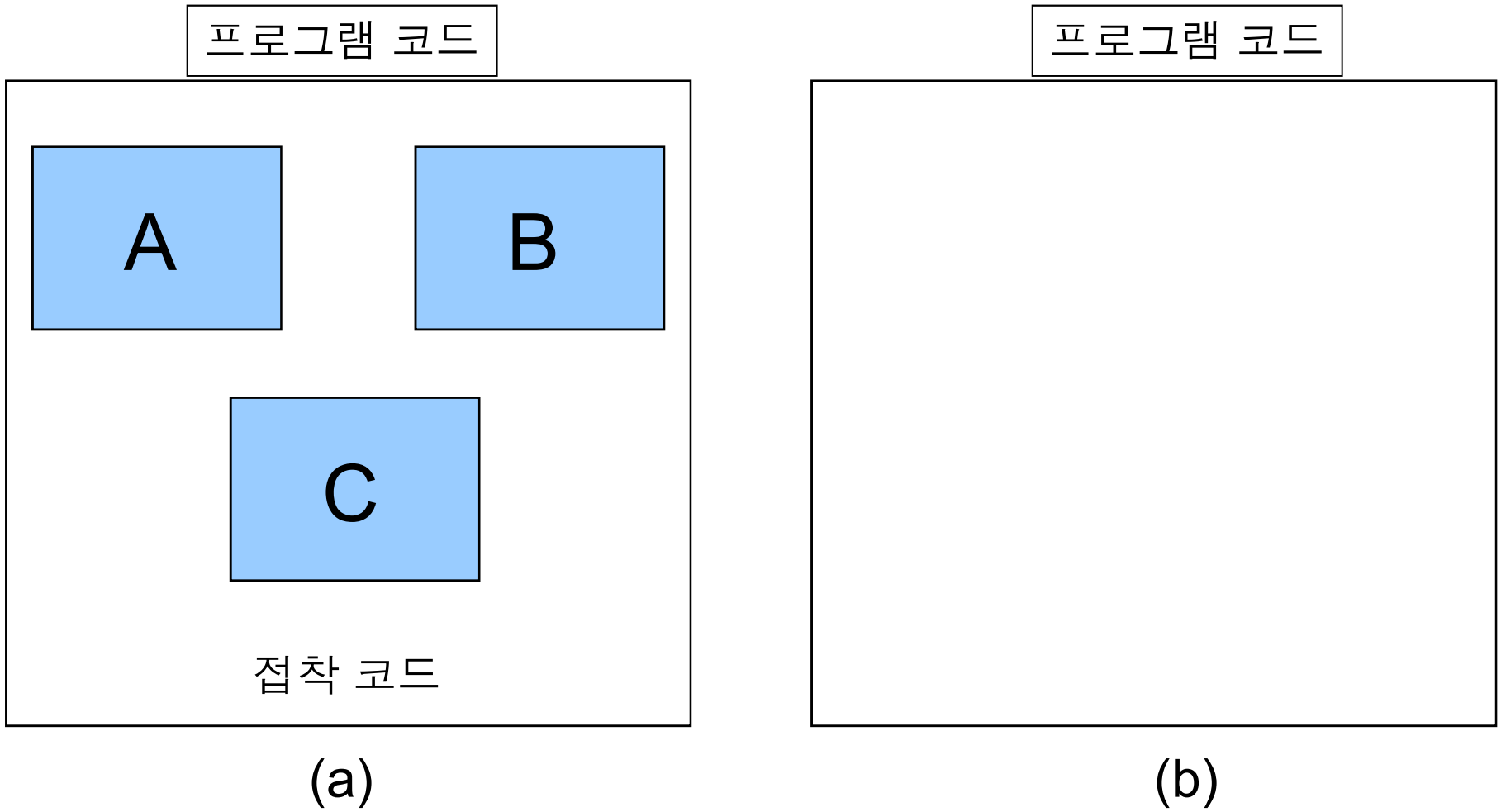
- **데이타 추상화(data abstraction)**
  - 객체의 명세(specification)와 구현(implementation)을 분리
  - 무엇(what)과 어떻게(how)를 명확하게 구분

데이타 타입

- **데이타 타입(data type)**
  - 객체(object)들과 이 객체들에 대한 연산(operation)의 집합
- **C++의 데이타 타입**
  - 기본 데이타 타입
    - char, int, float, double 등
    - 타입 수식어 : short, long, signed, unsigned
  - 파생 데이타 타입
    - 포인터(pointer) 타입, 참조(reference) 타입
  - 데이타를 묶어주는 구조
    - 배열(array), 구조(struct), 클래스(class)
  - 사용자 정의 데이타 타입
- **추상 데이타 타입(abstract data type; ADT)**
  - 객체와 연산에 대한 명세가 객체의 표현과 연산의 구현으로부터 분리된 방식으로 구성된 데이타 타입

데이타 추상화와 데이타 캡슐화의 장점

- 소프트웨어 개발의 간소화
  - 복잡한 작업을 부분 작업들로 분해
- 테스트와 디버깅
  - 각 부분 작업을 독자적으로 테스팅, 디버깅
- 재사용성
  - 자료 구조에 대한 코드와 연산을 추출해서 다른 소프트웨어 시스템에서도 사용
- 데이타 타입의 표현에 대한 수정
  - 데이타 타입의 내부 구현에 직접 접근하는 연산들만 수정



흰색이 버그 검사 대상 부분

(a)에서는 데이타 추상화를 사용, (b)에서는 데이타 추상화를 사용하지 않음.

C++의 기초

- **C++ 프로그램의 구성**
  - 헤더 화일
    - .h 접미사
    - 선언 저장에 사용
    - 시스템/사용자 정의 화일
    - 전처리기 명령 #include를 사용하여 화일에 포함시킴
  - 소스 화일
    - .c 접미사
    - 소스 코드 저장에 사용
- **프로그램의 실행**
  - 컴파일 → 링크 → 적재
- **헤더 화일 중복 방지**

```
#ifdef FILENAME_H
#define FILENAME_H
//헤더 화일 내용 삽입
        …
#endif
```

C++에서의 영역

- 화일 영역
  - 함수 정의에 속하지 않은 선언,클래스 정의, 네임스페이스
- 네임스페이스 영역
  - 네임스페이스(namespace)
    - 논리적으로 연관된 변수나 함수를 한 그룹으로 만드는 기법
  - 영역 정보를 사용해서 네임스페이스 안의 개체에 접근
    - 예) std::cout
- 지역 영역
  - 블록 안에서 선언된 이름
- 클래스 영역
  - 클래스 정의에 관련된 선언

C++ 명령문과 연산자

- **C++ 명령문**
  - C와 같은 문법과 의미
- **C++ 연산자**
  - new, delete 제외하고 C의 연산자와 동일
- **C++의 입출력**
  - 왼편/ 오른편 시프트 연산자(<<, >>) 사용
- **C와 C++의 중요한 차이점**
  - 연산자 다중화(operator overloading) 허용

C++ 데이타 선언

- 데이타 선언 : 한 데이타 타입을 어떤 이름에 연결
  - 상수 값
  - 변수 : 데이타 타입의 인스턴스
  - 상수 변수 : 선언시에 내용이 고정

    const int MAX=500;
  - 열거 타입 : 일련의 정수 상수를 선언

    enum *semester*{SUMMER, FALL, SPRING};
  - 포인터 : 객체의 메모리 주소 저장

    int i=25;
    int *np;
    np = &i;
  - 참조 타입 : 객체에 대한 또 다른 이름을 제공

    int i=5;
    int& j=i;
    i=7;
    printf("i=%d, j=%d", i, j); i=7, j=7

C++의 주석문

- 복수 행 주석문
  - C의 주석문과 동일
  - /*
    ……
    ……
    */

- 단일 행 주석문
  - C++에만 있음
  - // ……

C++에서의 입출력 (1)

- 시스템 정의 헤더 화일 **iostream**이 필요

- **cout**
  - 표준 출력 장치로 출력
  - <<연산자는 cout 키워드와 출력될 개체를 구분

```
#include <iostream>
main()
{
  int n=50; float f=20.3;
  cout<<"n:"<<n<<endl;
  cout<<"f:"<<f<<endl;
}
```

```
출력:  n:50
       f:20.3
```

C++에서의 입출력 (2)

- **cin**
  - 입력을 위해 사용
  - >> 연산자는 입력될 변수들을 구분
  - 여백 문자로 입력값을 구분

```
#include <iostream>

main(){
  int a,b;
  cin>>a>>b;
}
```

```
입력 1:
5 10 <Enter>

입력 2:
 5 <Enter>
10 <Enter>
```

C++에서의 입출력 (3)

- **C++에서의 입출력의 장점**
  - 자유 형식 : 포맷 기호 불필요
  - 입출력 연산자도 다중화 가능
- **화일 입출력**
  - 헤더 화일 fstream 필요

```
#include <iostream>
#include <fstream>
main(){
   ofstream outFile("my.out", ios::out);
   if (!outFile){
      cerr << "cannot open my.out" << endl; //표준 오류 장치
      return;
   }
   int n = 50;  float f = 20.3;
   outFile << "n: " << n << endl;
   outFile << "f: " << f << endl;
}
```

C++의 함수

- 정규 함수
- 멤버 함수: 특정 클래스와 연관된 함수
- 함수의 구성
  - 함수 이름
  - 인자 리스트 (시그너처(입력))
  - 반환 타입(출력)
  - 몸체(함수를 구현한 코드)

```
int Max(int a, int b){
   if (a > b) return a;
   else return b;
}
```

Max : 함수 이름
int *a*, **int** *b* : 인자 리스트
int : 반환 타입
{와}사이 : 함수 몸체

C++의 매개변수 전달

- **값(value)에 의한 전달**
  - 전달된 객체는 그 함수의 지역 저장소에 복사
  - 실인자에 영향을 주지 않음
  - 실인자로 제공되는 변수가 함수에 의해 변경되지 않음
- **참조(reference)에 의한 전달**
  - 인자 리스트의 타입 명세자에 & 첨가
  - 전달된 객체의 주소만 함수의 지역 저장소에 복사
  - 실인자에 접근
  - 전달된 객체가 큰 메모리를 요구하는 경우 더 빨리 수행
- **상수 참조 (constant reference)**
  - 인자 변경 불가

    const T& a
- **배열은 참조로 전달**

C++의 함수 이름 다중화

- 함수 다중화(**function overloading**)
  - 함수의 시그너처(인자 리스트)가 다르기만 하면 같은 이름을 가진 함수가 둘 이상 존재할 수 있음
  - e.g.

    int *Max*(**int**, **int**);
    int *Max*(**int**, **int**, **int**);
    int *Max*(**int***, **int**);
    int *Max*(**float**, **int**);
    int *Max*(**int**, **float**);

인라인 함수

- 함수 정의에 키워드 **inline**을 첨가해서 선언
  - 함수 호출과 인자 복사의 오버헤드를 줄임

- **e.g.**

```
inline int Sum(int a, int b)
{ return a + b;ㅁ }
```

  - $i = Sum(x,12)$; 명령문은 $i = x+12$; 로 대체

C++에서의 동적 메모리 할당

- **new / delete 연산자**
  - 실행 시간에 자유 저장소를 할당받거나 반환
- **new 연산자**
  - 원하는 타입의 객체를 생성하고 그에 대한 포인터를 반환
  - 생성할 수 없으면 예외 발생
- **delete 연산자**
  - new로 생성된 객체에게 할당된 기억 장소를 반환
- **e.g.**

```
int *ip = new int;
.
.
delete ip;
```

```
int *jp = new int[10]; //jp는 정수 배열
.
.
delete [] jp;
```

예외 발생

- 오류와 다른 특별한 조건이 발생했음을 알리는데 사용
- 다양한 예외 각각에 대해 예외 클래스를 정의
- **e.g**

```
int DivZero(int a, int b, int c)
{
  if (a <= 0 || b <= 0  || c <= 0)
      throw "All parameters should be >0”;
  return a + b * c + b / c;
}
```

예외 처리

- **try** 블록에 발생될 수 있는 예외를 포함시켜서 처리
- **try** 블록 뒤에는 **0**개 이상의 **catch** 블록
- 각 **catch** 블록은 예외 타입을 나타내는 한 인자를 가짐

catch (**char*** *e*){}
catch (*bad_alloc e*){}
catch (...){}

알고리즘 명세

- **알고리즘 (algorithm)**
  - 특정 작업을 수행하는 명령어들의 유한 집합

- **알고리즘의 요건**
  - 입력 : 외부에서 제공되는 데이타가 0개 이상
  - 출력 : 적어도 한 개 이상의 결과 생성
  - 명확성 : 각 명령은 명확하고 모호하지 않아야 함
  - 유한성 : 알고리즘대로 수행하면 어떤 경우에도 반드시 종료
  - 유효성 : 반드시 실행 가능해야 함

- **알고리즘 기술 방법**
  - 자연어
  - 흐름도(flowchart)
  - C++언어

선택정렬

- **n ≥ 1** 개의 서로 다른 정수의 집합을 정렬
  - “ 정렬되지 않은 정수들 중에서 가장 작은 값을 찾아서 정렬된 리스트 다음 자리에 놓는다 ”

```
for (int i = 0; i < n; i++) [
  a[i]에서부터 a[n-1]까지의 정수 값을 검사한 결과 a[j]가 가장 작은 값
 a[i]와 a[j]를 서로 교환;
}
```

```
 1 void SelectionSort(int *a, const int n)
 2 {// n개의 정수 a[0]부터 a[n-1]까지 비감소 순으로 정렬한다.
 3   for (int i = 0; i < n; i++)
 4   {
 5     int j = i;
 6     // a[i]와 a[n-1] 사이에 가장 작은 정수를 찾는다.
 7     for (int k = i + 1; k < n; k++)
 8       if (a[k] < a[j]) j = k;
 9     swap(a[i], a[j])
10   }
11 }
```

선택 정렬의 정확성 증명

- **정리 1.1**
  - 함수 *SelectionSort*(a, n)는 $n \geq 1$ 개의 정수를 정확하게 정렬한다. 그 결과는 $a[0], ..., a[n-1]$로 되고 여기서 $a[0] \leq a[1] \leq ... \leq a[n-1]$이다.

- **증명**
  - $i=q$ 에 대해, 외부 for가 수행되면 $a[q] \leq a[r]$, $q < r \leq n-1$ 이 된다.
  - $i>q$ 이면 $a[0]$에서 $a[q]$가 변하지 않는다.
  - 따라서 for를 마지막으로 수행하면(즉, $i=n-1$), $a[0] \leq a[1] \leq ... \leq a[n-1]$가 된다.

이원 탐색

- 이미 정렬된 배열 $a[0]...a[n-1]$에서 $x = a[j]$인 j를 반환
  - *left, right*: 탐색하고자 하는 리스트의 왼쪽, 오른쪽 끝
  - 초기 값으로 $left = 0$, $right = n-1$
  - 리스트의 중간 위치 $middle = (left + right) / 2$로 설정
  - $a[middle]$과 x 비교
    - ⑴ $x < a[middle]$
      → $right = middle-1$
    - ⑵ $x = a[middle]$
      → $middle$ 반환
    - ⑶ $x > a[middle]$
      → $left = middle+1$

Recursive 알고리즘

- 수행이 완료되기 전에 자기 자신을 다시 호출
  - 직접 재귀(direct recursion) : 함수가 그 수행이 완료되기 전에 자기 자신을 다시 호출
  - 간접 재귀(indirect recursion) : 호출 함수를 다시 호출하게 되어 있는 다른 함수를 호출

- 재귀 알고리즘 기술 방법
  1. 주어진 입력에 대해 함수가 마땅히 수행해야 할 작업에 대한 명령문 구성
  2. 자신에 대한 재귀적 호출이 제대로 이루어질 때 목적을 달성할 수 있는지 검증
  3. 함수를 유한한 횟수로 순환 호출하면 완료 조건을 만족시키는지 확인
  4. 완료 조건을 만족시켰을 때 함수가 올바른 계산을 하는지 확인

재귀적 이원 탐색(Recursive Binary Search)

- **BinarySearch(a, x, 0, n-1)**

```
int BinarySearch(int *a, const int x, const int left, const int right)
{//정렬된 배열 a[left], ..., a[right]에서 x 탐색
  if(left <= right){
    int middle = (left + right) /2;
    if(x < a[middle]) return BinarySearch(a, x, left, middle-1);
    else if(x > a[middle]) return BinarySearch(a, x, middle+1, right);
    return middle;
  }//if의 끝
  return -1;//발견되지 않음
}
```

순열 생성기

- 주어진 집합의 모든 가능한 순열 출력
- **Permutations(a,0,n-1)**

```
void Permutations(char *a, const int k, const int m)
{// a[k], ..., a[m]에 대한 모든 순열 생성
  if(k == m){//순열을 출력
    for (int i = 0 ; i <= m; i++) cout << a[i] << " ";
    cout << endl;
  }
  else {//a[k:m]에는 하나 이상의 순열이 있다. 이 순열을 재귀적으로 생성
    for(i=k;i<=m;i++){
      swap(a[k], a[i]);
      Permutations(a, k+1, m);
      swap(a[k], a[i]);
    }
  }
}
```

표준 템플릿 라이브러리

- **표준 템플릿 라이브러리(STL:standard templates library)**
  - 컨테이너(container)
  - 어댑터(adapter)
  - 반복자(iterator)
  - 함수 객체(function)
  - 알고리즘

  - #include<algorithm> 첨가

accumulate STL 알고리즘

- 순차에 있는 원소들을 합산
- **accumulate(start, end, initialValue)**
  - accumulate(a, a+n, initialValue)

$$\longrightarrow \quad initialValue + \sum_{i=0}^{n-1} a[i]$$

- **accumulate(start, end, initialValue, operator)**
  - operator : 합산 과정에서 사용될 연산을 정의하는 함수
  - e.g 정수로 된 배열의 곱셈

```
int Product(int *a, int n)
{//a[0:n-1]의 수의 곱셈을 반환
  int initVal = 1;
  return accumulate(a, a+n, initVal, multiplies<int>());
}
```

STL 알고리즘 copy와 next_permutation

- **copy**
  - 원소 순차를 한 장소에서 또 다른 장소로 복사
  - copy(start, end, to)
    - start, start+1,..., end-1에서 to, to+1, ..., to+end-start로 복사
- **next_permutation**
  - [start,end) 범위에 있는 원소들의 다음으로 큰 사전식 순열 생성
  - 그러한 다음 순열이 존재하면 true 반환
  - next_permutation(start,end)

```
void Permutations(char *a, const int m)
{//a[0:m]의 모든 순열 생성
  //모든 순열을 하나씩 출력
  do{
     copy(a, a+m+1, ostream_iterator<char>(cout," "));
     cout << endl;
  }while(next_permutation(a, a+m+1));
}
```

STL 알고리즘 sort와 count와 fill

- **sort(start, end)**
  - sorts elements in the range [start, end) into ascending order.

- **count(start, end, value)**
  - returns the number of occurrences of value in the range [start,end)
- **fill(start, end, value)**
  - sets all positions in the range [start,end) to value.

STL sort(start, end) algorithm example

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
bool myfunction (int i,int j) { return (i<j); }

struct myclass {
  bool operator() (int i,int j) { return (i<j);}
} myobject;

int main () {
  int myints[] = {32,71,12,45,26,80,53,33};
  vector<int> myvector (myints, myints+8); // 32 71 12 45 26 80 53 33
  vector<int>::iterator it; // using default comparison (operator <):
  sort (myvector.begin(), myvector.begin()+4); //(12 32 45 71)26 80 53 33

  // using function as comp
  sort (myvector.begin()+4, myvector.end(), myfunction); // 12 32 45 71(26 33 53 80)

  // using object as comp
  sort (myvector.begin(), myvector.end(), myobject); //(12 26 32 33 45 53 71 80)

  // print out content:
  cout << "myvector contains:";
  for (it=myvector.begin(); it!=myvector.end(); ++it)
    cout << " " << *it; cout << endl;
  return 0;
}
```

성능 분석과 측정 (1)

- 프로그램의 평가 기준
  - 우리가 원하는 작업을 하는가?
  - 원래 작업의 명세에 부합해서 정확히 작동하는가?
  - 문서화가 되어 있는가?
  - 논리적 작업 단위를 수행하는 기준으로 함수가 생성되었는가?
  - 코드가 읽기 쉬운가?

- **성능 평가(performance evaluation)**
  - 성능 분석 (performance analysis)
    - 사전 예측
  - 성능 측정 (performance measurement)
    - 이후 검사

공간 복잡도 (1)

- **공간 복잡도(space complexity)**
  - 프로그램을 실행시켜 완료하는 데 필요한 메모리 양
  - 고정 부분
    - 보통 명령어 공간, 단순 변수, 집합체, 상수를 위한 공간
  - 가변 부분
    - 변수, 참조된 변수가 필요로 하는 공간, 재귀스택 공간
  - 프로그램 P의 공간 요구 $S(P) = c + S_P$
  - c : 상수
  - S_P : 인스턴스 특성
    - 문제에 따라 달라지는 값
    - 인스턴스로 인해 필요로 하는 공간

공간 복잡도 (2)

```
float Abc(float a, float b, float c)
{
  return a+b+b*c+(a+b-c)/(a+b)+4.0;
}
```

- a,b,c 값 각각을 저장하고 Abc에서 값을 반환하는데 한 워드가 필요하다고 가정
- $S_P = 0$

```
line float Sum(float *a, const int n)
{
  float s = 0;
  for(int i = 0 ; i < n ; i++)
    s+= a[i];
  return s;
}
```

- $S_{Sum}(n) = 0$

공간 복잡도(3)

```
line float Rsum(float *a, const int n)
1   {
2     if(n<=0) return 0;
3     else return(Rsum(a, n-1) + a[n-1]);
4   }
```

- R_{Sum}을 호출할 때마다 적어도 네 개의 워드 (n과 a 값, 반환 값, 반환 주소에 필요한 공간 포함)
- 재귀 깊이가 n+1이므로 재귀 스택 공간에 4(n+1)

시간 복잡도 (1)

- 시간 복잡도
  - 프로그램을 완전히 실행시키는데 필요한 시간
  - T(P) = 컴파일 시간 +실행시간 t_P
  - 실행시간에 주로 관심
  - $t_P(n) = c_aADD(n)+c_sSUB(n)+c_mMUL(n)+c_dDIV(n) + \ldots$
    - n : 인스턴스 특성
    - C_a, C_s, C_m, C_d : +, -, x, / 연산을 위한 상수 시간
    - ADD, SUB, MUL, DIV : 특성 n인 인스턴스에서 각 연산의 실행 횟수

시간 복잡도 (2)

- **프로그램 단계 수(number of steps)**
  - 주석 : 0
  - 선언문 : 0
    - 변수, 상수 정의(int, long, short, char,...)
    - 사용자 데이타 타입 정의(class, struct, union, template)
    - 접근 결정(private, public, protected, friend)
    - 함수 타입 결정(void, virtual)
  - 산술식 및 지정문 : 1
    - 예외 : 함수 호출을 포함하는 산술식
  - 반복문 : 제어 부분에 대해서만 단계수 고려
  - 스위치 문
    - switch(<expr>)의 비용 = <expr>의 비용
    - 각 조건의 비용 = 자기의 비용 + 앞서 나온 모든 조건의 비용
  - if-else 문
    - <expr>, <statement1>, <statement2>에 따라 각각 단계수가 할당

시간 복잡도 (3)

- **프로그램 단계 수(number of steps)**
  - 함수 호출
    - 값에 의한 전달 인자 포함하지 않을 경우 : 1
    - 값에 인한 전달 인자 포함할 경우 : 값 인자 크기의 합
    - 재귀적인 경우 : 호출되는 함수의 지역 변수도 고려
  - 메모리 관리 명령문 : 1
  - 함수문 : 0
    - 비용이 이미 호출문에 할당
  - 분기문 : 1

단계 수 테이블

- 명령문의 실행당 단계 수(**s/e : step per execution**) 결정
  - s/e : 그 명령문의 실행 결과로 count가 변화하는 양
- 명령문이 실행되는 총 횟수 계산

```
line float Sum(float * a, const int n)
1    {
2      float s = 0;
3      for(int i = 0 ; i < n ; i++)
4        s+= a[i];
5      return s;
6    }
```

line	s/e	freq	total steps
1	0	1	0
2	1	1	1
3	1	n+1	n+1
4	1	n	n
5	1	1	1
6	0	1	0
		총 단계 수	2n + 3

피보나치 수

- 피보나치 수열
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
  - $F_0 = 0, F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}, n \geq 2$

```
1 void Fibonacci(int n)
2 {//피보나치 수 Fn 계산
3   if(n<=1) cout << n << endl; //F0 =0와 F1=1
4   else { // Fn을 계산한다
5       int fn; int fnm2 = 0; int fnm1 = 1;
6       for(int i = 2 ; i <=n ;i++)
7       {
8         fn = fnm1 + fnm2;
9         fnm2 = fnm1;
10        fnm1 = fn;
11      }//for문의 끝
12      cout << fn << endl;
13   }//else의 끝
14 }
```

line	s/e	freq	total steps
3a	1	1	1
3b	1	1	1
5	1	2	2
6	1	n	n
8-10	1	n-1	n-1
12	1	1	1

(case)n=0, 1인 경우 : 총 2
  3a, 3b 총 2회
(case)n>1인 경우 : 총 4n+1
 3a, 12에서 각 1회, 5에서 2회,
 6에서 n회, 8-10에서 각n-1회

점근 표기법 (1)

- 빅오(**big 'oh'**)
  - 모든 n, $n \geq n_0$에 대해 $f(n) \leq cg(n)$인 조건을 만족시키는 두 양의 상수 c와 n_0가 존재하면 $f(n)=O(g(n))$
  - 예
    - $3n + 2 = O(n)$ ($c=4, n_0=2$)
    - $3n + 3 = O(n)$ ($c=4, n_0=3$)
    - $100n + 6 = O(n)$ ($c=101, n_0=10$)
    - $10n^2 + 4n + 2 = O(n^2)$ ($c=11, n_0=5$)
    - $1000n^2 + 100n - 6 = O(n^2)$ ($c=1001, n_0=100$)
    - $6*2^n + n^2 = O(2^n)$ ($c=7, n_0=4$)
    - $3n + 3 = O(n^2)$ ($c=3, n_0=2$)
    - $10n^2 + 4n + 2 = O(n^4)$ ($c = 10, n_0=2$)

    - $3n + 2 \neq O(1)$
    - $10n^2 + 4n + 2 \neq O(n)$

점근 표기법 (2)

- 연산 시간
  - 상수시간 : $O(1)$
  - 로그시간 : $O(\log n)$
  - 선형시간 : $O(n)$
  - n로그시간 : $O(n\log n)$
  - 평방시간 : $O(n^2)$
  - 입방시간 : $O(n^3)$
  - 지수시간 : $O(2^n)$
  - 계승시간 : $O(n!)$

- 연산 시간의 크기 순서

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

점근 표기법 (3)

- 오메가 **(omega)**
  - 모든 n, $n \geq n_0$에 대해 $f(n) \geq cg(n)$을 만족시키는 두 양의 상수 c와 n_0가 존재하면 $f(n) = \Omega(g(n))$
  - 예
    - $3n + 2 = \Omega(n)$ ($n_0=1$, $c=3$)
    - $3n + 3 = \Omega(n)$ ($n_0=1$, $c=3$)
    - $100n + 6 = \Omega(n)$ ($n_0=1$, $c = 100$)
    - $10n^2 + 4n + 2 = \Omega(n^2)$ ($n_0=1$, $c = 1$)
    - $6*2^n + n^2 = \Omega(2^n)$ ($n_0=1$, $c = 1$)

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

점근 표기법 (4)

- 세타(theta)
  - 모든 n, $n \geq n_0$에 대해 $c_1 g(n) \leq f(n) \leq c_2 g(n)$을 만족시키는 세 양의 상수 c_1, c_2와 n_0가 존재하면, $f(n) = \Theta(g(n))$
  - 예
    - $3n + 2 = \Theta(n)$ ($c_1=3$, $c_2=4$, $n_0=2$)
    - $3n + 3 = \Theta(n)$
    - $10n^2 + 4n + 2 = \Theta(n^2)$
    - $6*2^n + n^2 = \Theta(2^n)$
    - $10*\log n + 4 = \Theta(\log n)$
    - $3n + 2 \neq \Theta(1)$
    - $3n + 3 \neq \Theta(n^2)$
    - $10n^2 + 4n + 2 \neq \Theta(n)$
    - $10n^2 + 4n + 2 \neq \Theta(1)$
    - $6*2^n + n^2 \neq \Theta(n^{100})$
    - $6*2^n + n^2 \neq \Theta(1)$

매직 스퀘어(magic square)

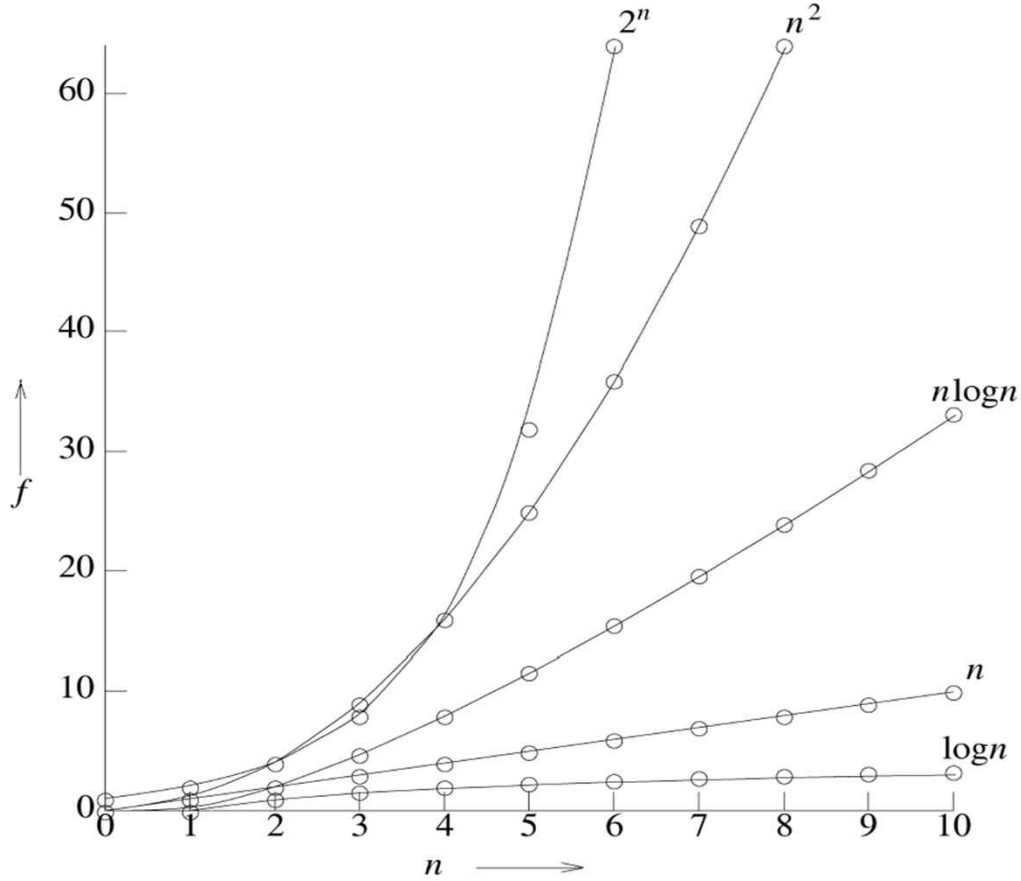
- **1~n^2**까지의 정수로 된 **n×n** 행렬
- 각 행의 합, 열의 합, 주대각선의 합이 모두 동일

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

첫번 째 행의 중앙에 1을 넣는 것부터 시작한다. 빈 정방형에 1씩 큰 수를 할당하면서 왼쪽 대각선 방향으로 올라간다. 만약 정방형 밖으로 벗어나면 정방형의 반대편 자리에서 계속한다. 즉 상단을 벗어나면 같은 열의 최하단으로, 왼쪽에서 벗어나면 같은 행의 제일 오른쪽으로 이동한다. 만약 정방형이 채워져 있으면 밑으로 움직여서 계속한다.

실용적인 복잡도

$\log n$	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



성능 측정(1)

- 프로그램의 공간 및 시간 요구량을 구하는 것

- 순차 탐색 함수에 대한 최악의 경우 성능 측정
  - 시간을 측정할 n의 값들 결정
  - n의 값들에 대해서 최악의 성질을 가진 데이타 결정

```
int SequentialSearch (int *a, const int n, const int x)
{// a[0:n-1]에 대해 탐색
  int i;
  for(i=0; i<n && a[i] != x; i++);
  if(i==n) return -1;
  else return i;
}
```

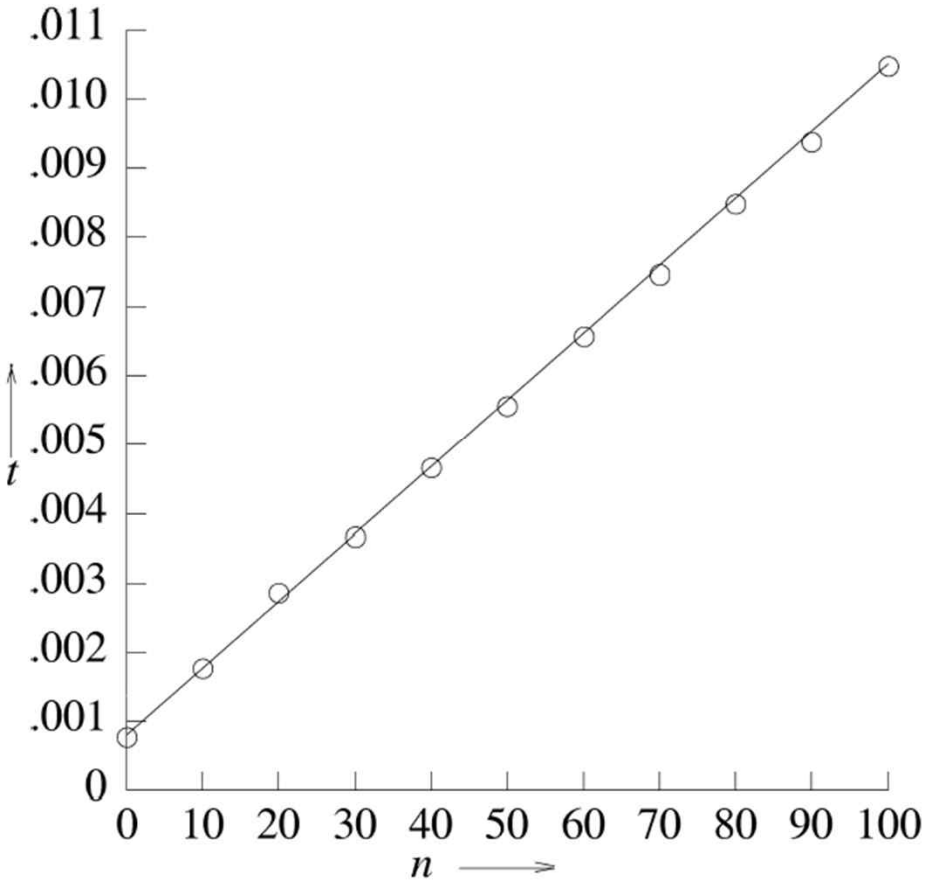
  - 점근적 해석 : Θ(n)
    - 작은 n 값, 무시된 저차 항들의 영향에 의해 정확하지 않을 수 있음

성능 측정 (2)

- 시간 측정 프로그램 수행 결과

n	*total*	*runTime*	*n*	*total*	*runTime*
0	241	0.0008	100	527	0.0105
10	533	0.0018	200	505	0.0202
20	582	0.0029	300	451	0.0301
30	736	0.0037	400	593	0.0395
40	467	0.0047	500	494	0.0494
50	565	0.0056	600	439	0.0585
60	659	0.0066	700	484	0.0691
70	604	0.0075	800	467	0.0778
80	681	0.0085	900	434	0.0868
90	472	0.0094	1000	484	0.0968

100분의 1초로 잰 시간



테스트 데이타의 생성

- 최악의 성능을 초래하는 데이타 세트 생성
  - 관심 있는 인스턴스 특성 값들에 대해 적당한 대규모의 무작위 테스트 데이타 생성
  - 생성한 테스트 데이타 각각에 대해 실행 시간 측정
  - 이들 시간의 최대 값을 최악의 경우의 시간으로 사용
- 평균인 경우의 시간 측정
  - 주어진 특성의 모든 가능한 인스턴스에 대한 평균
  - 위에 언급된 기법을 사용해서 평균 시간의 어림값 계산
- 실험을 위해 생성되어야 할 데이타들을 결정하기 위해 사용되는 알고리즘을 분석하는 것이 바람직