강의 소개

1. 교재

   C로 쓴 자료구조론, Horowitz, Sahni, Anderson-Freed

   보조: introduction to algorithms, cormen 외 3명, MIT press

2. 강의 자료

   E 강의실 게시판

3. 교수

   컴퓨터공학부 지상문

   연구실: 8관 606호

   전화: 607-5146

   E-mail: smchiks@ks.ac.kr

4. 성적

   퀴즈 20%, 중간 40%, 기말 40%

시스템 생명 주기

(1) 요구사항(requirements)
프로젝트들의 목적을 정의한 명세(specification)들의 집합
입력과 출력 정보의 기술

(2) 분석(analysis)
문제들을 실제 다룰 수 있을 정도의 작은 단위들로 나눔
상향식(bottom-up) / 하향식(top-down)

(3) 설계(design)
자료 객체들과 수행될 연산들의 관점
- 추상 자료형 (abstract data type)
- 알고리즘의 명세와 설계 기법 고려

시스템 생명 주기

(4) 정제와 코딩(refinement & coding)
자료 객체에 대한 표현 선택
수행될 연산에 대한 알고리즘 작성

(5) 검증(Verification)
정확성 증명(correctness proofs)
- 수학적 기법들을 사용하여 프로그램의 정확성 증명

테스트(testing) : 테스트 데이타와 수행 가능한 코드
- 프로그램의 정확한 수행 검증
- 프로그램의 성능 검사

오류(error) 제거
- 독립적 단위 테스트
- 통합 테스트

1.2 알고리즘 명세

알고리즘(Algorithm)

특정한 일을 수행하기 위한 명령어의 유한 집합

다음 조건(criteria) 만족하여야 함

i. 입력 : 외부에서 제공되는 데이터가 0개 이상

ii. 출력 : 적어도 한 가지의 결과

iii. 명확성(definiteness) : 모호하지 않은 명확한 명령

iv. 유한성(finiteness) : 한정된 수의 단계 뒤에는 종료

v. 유효성(effectiveness) : 기본적, 실행가능 명령

iodef

- 프로그램은 유한성을 만족하지 않아도 됨
- 알고리즘을 자연어로 기술할 경우에는 명확성에 유의

예제 1.1 n >= 1개의 서로 다른 정수 정렬 프로그램

- 알고리즘이 아닌 예 : "정렬되지 않은 정수들 중에서 가장 작은 값을 찾아서 정렬된 리스트 다음 자리에 놓는다"

- 알고리즘 기술을 위한 첫번째 예:

정수들이 배열(array), list에 저장: i 번째 정수는 list[i] 에 저장

```
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;
}
```

최소 정수를 찾는 작업

① 최소 정수가 list[i]라 가정

② list[i]와 list[i+1] ~ list[n-1] 비교

- 더 작은 정수를 만나면 새로운 최소정수로 선택

최소 정수 값을 list[i] 값과 교환하는 작업

최소 정수 값을 list[i] 값과 교환하는 작업

- 함수 사용 : swap(&a, &b) //a, b는 정수형 변수

```
void swap(int *x, int *y)
/* 매개 변수 x, y는 정수형을 갖는 포인터 변수이다 */
{
   int temp = *x;  /* temp 변수를 int로 선언하고  x가
                      가리키는 주소의 내용을 지정한다 */
       *x = *y;     /* y가 가리키는 주소의 내용을 x가
                      가리키는 주소에 저장한다 */
       *y = temp;  /* temp의 내용을 y가 가리키는 주소에
                      저장한다 */
}
```

- 매크로 정의

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

- 프로그램 1.3 선택 정렬

```
#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");
    exit(1);
  }
  for (i=0; i<n; i++) { /* randomly generate numbers*/
    list[i] = rand() % 1000;
    printf(%d  ", list[i]);
 }
```

```
  sort(list, n);
  printf("Sorted array:");
  for (i=0; i<n; i++)    { /* print out sorted
    numbers */
     printf("%d  ", list[i]);
  printf("");
}

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);
 }
}
```

정확성 증명

정리 1.1

함수 SORT(list, n)는 n >= 1 개의 정수를 정확하게 정렬한다. 그 결과는 list[0], ..., list[n-1]로 되고 여기서 list[0] <= list[1] <= ... <= list[n-1]이다.

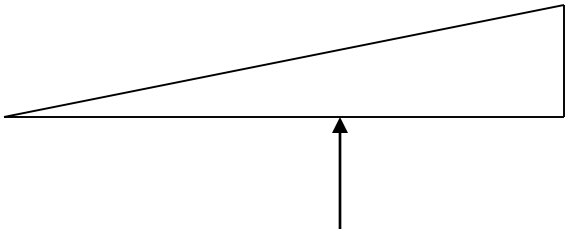
증명

i=q 에 대해, 외부 for문이 완료되면 list[q] <= list[r],
q < r < n 이다.

다음 반복에서는 i>q 이고 list[0]에서 list[q]까지는 변하지 않는다.

따라서 for문을 마지막으로 수행하면(즉, i=n-2),
list[0] <= list[1] <= ... <= list[n-1]가 된다. □

- 예제 [이진탐색] : 정수 searchnum이 배열 list에 있는지 검사
  - list[0] <= list[1] <= ... <= list[n-1]로 이미 정렬된 상태임
  - 있다면 list[i] = searchnum인 인덱스 i를 반환
  - 없다면 -1 반환



- 기본 아이디어

초기 값 : left = 0, right = n-1

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

list[middle]과 searchnum을 비교한 후

1) searchnum < list[middle] :

   list[0] <= searchnum <= list[middle - 1]

   right = middle-1

2) searchnum = list[middle] : middle을 반환

3) searchnum > list[middle] :

   list[middle+1] <= searchnum <= list[n-1]

   left = middle+1

탐색 전략 기술

```
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;
}
```

비교 연산 : 함수, 매크로

```
#define COMPARE(x,y) ((x) < (y)) ? -1 : ((x) == (y)) ? 0 : 1)
```

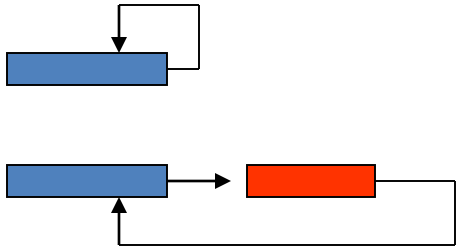
- 프로그램 1.6 순서 리스트 탐색

```
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;
        }
  }
  return -1;
}
```

순환 알고리즘

수행이 완료되기 전에 자기 자신을 다시 호출

- 직접 순환 : direct recursion
- 간접 순환 : indirect recursion



- factorial, 거듭제곱, 이항 계수

$$n! = n * (n-1) * \cdots * 1$$

$$\begin{bmatrix} n \\ m \end{bmatrix} = \frac{n!}{m!(n-m)!}, \begin{bmatrix} n \\ m \end{bmatrix} = \begin{bmatrix} n-1 \\ m \end{bmatrix} + \begin{bmatrix} n-1 \\ m-1 \end{bmatrix}$$

- 이후의 장에서 필요: 리스트, 트리 등

Factorial 계산

- 반복 (iterative) 계산

```
int fac_iter (int n) {
        int fact=1, k ;
        for (k=n; k>0; --k)
                fact *= k ;
        return fact ;
}
```

- 순환 (recursive) 계산

```
int fac_rec (int n) {
        if (n <= 1) return 1 ;  // 순환을 멈추는 부분
        else return n * fac_rec (n-1);//작은 부분으로 분할
}
```

거듭제곱 계산

- 반복 (iterative) 계산

```
double pow_iter (double x, int n) {
        double pow = 1;
        for (int k=0; k<n; ++k)
                pow *= x ;
        return pow ;
}
```

- 순환 (recursive) 계산 : 이경우는 더 빠름

```
double pow_rec (double x, int n) {
        if (n == 0) return 1 ;  // 순환을 멈추는 부분
        else if (n%2==0) { // 짝수
                return pow_rec(x*x, n/2) ;
        } else { //홀수
                return x*pow_rec(x*x, (n-1)/2) ;
        }
}
```

- 예제 [이진 탐색에 대한 순환 구현]

```
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)  { //순환호출이 종결될 수 있도록 경계 조건 설정
        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;
}
```

예제 [순열] : n >= 1 개의 원소를 가진 집합에서 모든 가능한 순열을 출력하는 함수

- a, b, c : (a,b,c),(a,c,b),(b,a,c), (b,c,a),(c,a,b),(c,b,a)
- n 원소 : n! 개의 상이한 순열
- a, b, c, d
  1) a로 시작하는 b, c, d의 모든 순열
  2) b로 시작하는 a, c, d의 모든 순열
  3) c로 시작하는 a, b, d의 모든 순열
  4) d로 시작하는 a, b, c의 모든 순열

초기 함수 호출 : *perm(list, 0, n-1);*

```
void perm(char *list, int i, int n)
/* generate all the permutations of list[i] to list[n]*/ {
  int j, temp;
  if (i == n)  { // 경계조건
    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);
    }
  }
}
```

하노이 탑 (P15 연습문제 11)

```
#include <stdio.h>
void hanoi (int n, char from, char tmp, char to) {
    if (n==1) {
            printf ("disc 1 from %c to %c₩n", from, to) ;
    } else {
            hanoi (n-1, from, to, tmp) ;
            printf ("disc %d from %c to %c₩n", n, from, to) ;
            hanoi (n-1, tmp, from, to) ;
    }
}
// tower A, B, C  and discs from A to C
int main () {
    hanoi (64, 'A', 'B', 'C') ;
}
```

하노이탑 실행결과

1. hanoi (1, 'A', 'B', 'C') ;
disc 1 from A to C

2. hanoi (2, 'A', 'B', 'C') ;
disc 1 from A to B
disc 2 from A to C
disc 1 from B to C

3. hanoi (3, 'A', 'B', 'C') ;
disc 1 from A to C
disc 2 from A to B
disc 1 from C to B
disc 3 from A to C
disc 1 from B to A
disc 2 from B to C
disc 1 from A to C

4. hanoi (4, 'A', 'B', 'C') ;
disc 1 from A to B
disc 2 from A to C
disc 1 from B to C
disc 3 from A to B
disc 1 from C to A
disc 2 from C to B
disc 1 from A to B
disc 4 from A to C
disc 1 from B to C
disc 2 from B to A
disc 1 from C to A
disc 3 from B to C
disc 1 from A to B
disc 2 from A to C
disc 1 from B to C

1. 3 데이타 추상화

- C 언어의 기본 데이타 타입 : char, int, float, double
  - 키워드 short, long, unsigned에 의해 변경

- 자료의 그룹화 :배열(array), 구조체(structure)
  - int list[5] : 정수형 배열,
  - 구조체

```
struct student  {
        char last_name;
        int student_id;
        char grade;
    }
```

- 포인터 데이타 타입 : 정수형, 실수형, 문자형, float형 포인터

```
int i, *pi;
```

- 만들어진 새로운 데이타 타입 : 사용자 정의 데이타 타입

- 정의 : 데이타 타입(data type)은 객체(object)와 그 객체 위에 작동하는 연산(operation)들의 집합

- 데이터 타입 int의 예
  - 객체 : 0, +1, -1, +2, -2, ..., INT_MAX, INT_MIN
  - 연산 : +, -, *, /, %, 테스트 연산자, 치환문 ...
    - atoi와 같은 전위(prefix) 연산자, +와 같은 중위(infix) 연산자
    - 이름, 매개 변수, 결과가 명세 되어야 함

- 데이터 타입의 객체 표현
  - char 형 : 1 바이트 비트 열
  - int 형 : 2 또는 4바이트
  - 구체적인 내용을 사용자가 모르도록 하는 것이 좋은 방법
    객체 구현 내용에 대한 사용자 프로그램의 독립성

- 정의 : 추상 자료형(ADT : abstract data type)
  - 객체의 명세와 그 연산의 명세가 그 객체의 표현과 연산의 구현으로부터 분리된 자료형
  - 연산의 구현이나 객체의 표현에 독립적으로 객체의 필수요소들을 이해

- 명세와 구현을 명시적으로 구분
  Ada - package, C++ - Class

- ADT 연산의 명세
  - 함수 이름, 매개 변수형, 결과형, 함수가 수행하는 기능에 대한 기술
  - 내부적 표현이나 구현에 대한 자세한 설명은 필요 없음
    ADT가 구현에 독립

- ADT의 정의가 완전히 설명되어지면, 그 후 구현과 표현방법에 대해 논의

예 [추상 자료형 Natural_Number]

Structure Natural_Number

객체(objects): 0 에서 시작해서 컴퓨터상의 최대 정수 값 (INT_MAX)까지 순서화 된 정수의 부분 범위이다.

함수(functions): for all x, y ∈ Nat_Number, TRUE, FALSE ∈ Boolean에 대해, 여기서 +, -, <, 그리고 == 는 일반적인 정수 연산자이다.

```
Nat_No Zero()              ::=    0
Boolean Is_Zero(x)         ::=    if (x) then 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
```

end Natural_Number

성능 분석 및 측정

- 1.4장 성능 분석(*performance analysis*)
  - 시간과 공간의 추산, 복잡도 이론(complexity theory)
  - a priori estimates
- 1.5장 성능 측정(*performance measurement*)
  - 컴퓨터 의존적 실행 시간
  - a posteriori testing

- 성능분석의 두개의 복잡도 정의
  - 공간 복잡도(space complexity) : 프로그램을 실행시켜 완료하는데 필요한 공간의 양
  - 시간 복잡도(time complexity) : 프로그램을 실행시켜 완료하는데 필요한 컴퓨터 시간의 양

공간 복잡도

- 프로그램에 필요한 공간

  1) 고정공간요구 *c* : 프로그램 입출력의 횟수나 크기와 관계없는 공간 요구, 명령어 공간, 단순 변수, 고정 크기의 구조체 변수, 상수

  2) 가변공간요구 *Sp(I)* : 특정 인스턴스 *I*에 의존하는 크기를 가진 가변 공간, 스택 공간

  예) 입력이 *n*개의 요소를 갖는 배열이라면 n은 인스턴스 특성, n이 유일한 인스턴스 특성이면 *Sp(I)* 표현을 위해 *Sp(n)* 사용

- 총 공간 요구량 : *S(P) = c + Sp(I)*

예제 : 고정 공간 요구만을 가지는 함수이므로 *Sabc(I) = 0*

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

- 예제 1.7 : 가변 공간 요구, 배열 (크기 n)
  - 함수에 대한 배열의 전달 방식
  - Pascal : 값 호출(call by value)
    .*Ssum(I)=Ssum(n)=n* : 배열 전체가 임시기억장소에 복사
  - C : 배열의 첫번째 요소의 주소 전달
    . *Ssum(n) = 0*

```
float sum(float list[], int n) {
   float tempsum = 0;
   int i;
   for (i = 0; i < n; i++)
      tempsum += list[i];
     return tempsum;
 }
// 프로그램 1.10
```

- 예제 1.8
  - rsum : 컴파일러가 매개 변수, 지역 변수, 매 순환 호출시에 복귀 주소를 저장

```
float rsum(float list[], int n) {
   if (n) return rsum(list, n-1) + list[n-1];
   return 0;
} //프로그램 1.11
```

- 하나의 순환 호출을 위해 요구되는 공간 (80386 예)
  - 두개의 매개 변수, 복귀 주소를 위한 바이트 수
    = 2 + 2 + 2 = 6
- 배열이 n = MAX_SIZE 만큼의 기억장소를 가진다면, 가변공간은 *Srsum(MAX_SIZE) = 6 * MAX_SIZE* 으로 순환함수는 반복함수보다 훨씬 큰 오버헤드를 가짐

1.4.2 시간 복잡도

- 프로그램 P에 의해 소요되는 시간 : T(P)
  - 컴파일 시간 + T_P (실행 시간)

- *T_P(n)=Ca ADD(n) + Cs SUB(n) + Cl LDA(n)+ Cst STA(n)*
  - *Ca, Cs, Cl, Cst* : 각 연산을 수행하기 위해 필요한 상수 시간
  - *ADD, SUB, LDA, STA(n)* : 특성 n에 대한 연산 실행 횟수

- 실제 컴퓨터 의존적인 실행시간을 구할 때는 시스템 클럭을 이용한 성능측정이용

프로그램 단계(program step)

- 컴퓨터에 독립적인 견적은 연산의 횟수를 계산
- 정의 : 프로그램 단계(program step)
  - 실행 시간이 인스턴스 특성에 상관없이 구문적으로 또는 의미적으로 독립성을 갖는 프로그램의 단위

- *1 step 예*
  - *a = 2*
  - *a = 2*b+3*c/d-e+f/g/a/b/c*
  - 한단계 실행에 필요한 시간이 인스턴스 특성에 독립적이어야 함.

- 프로그램 단계의 계산 방법 1
  - 전역 변수 *count*의 사용

예제 [수치 값 리스트의 합산을 위한 반복 호출]

```
float sum(float list[], int n)
{ float tempsum = 0; count++;    /* 배정문을 위한 선언 */
   int i;
   for (i = 0; i < n; i++) {
     count++;                    /* for 루프를 위한 연산 */
     tempsum += list[i]; count++;  /* 배정문을 위한 연산 */
   }
   count++;    /* for문의 마지막 실행 */
   count++;     /* 반환을 위한 문장 */
   return tempsum;
}   // ----------------------------------------------
float sum(float list[], int n)  /* 단순화된 프로그램 */
{ float tempsum = 0;
   int i;
   for (i = 0; i < n; i++)
      count += 2;
   count += 3;
   return 0;
}
```

프로그램 단계수 *2n + 3 steps*

예제 [수치 값 리스트의 합산을 위한 순환 호출]

```
float rsum(float list[], int n)
{
   count++;          /* if문을 위한 문장 */
   if (n)  {
      count++;       /* 반환과 rsum의 호출을 위한 문장 */
      return rsum(list, n-1) + list[n-1];
   }
   count++;
   return list[0];
 }
```

---

n = 0 -> 2 (if, 마지막 return)

n > 0 -> 2 (if, 처음 return) : n회 호출

2n + 2 steps

2n + 3 (iterative) > 2n + 2 (recursive)

Titerative > Trecursive ??

단계수가 많지만, 각 단계가 실행에 걸리는 시간은 순환적인 것이 더 느리다.

- 단계의 계산 (방법 2)

  테이블 방식(*tabular method*) : 단계수 테이블

  ① 문장에 대한 단계수 : *steps/execution, s/e*

  ② 문장이 수행되는 횟수 : 빈도수(*frequency*)

  – 비실행 문장의 빈도수 = 0

  ③ 총 단계수 = 빈도수 x *s/e*

- [수치 값 리스트의 합산을 위한 반복 호출]

	s/e	*빈도수*	*총 단계수*
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
합계			2n+3

예제 [수치 값 리스트의 합산을 위한 순환 호출]

	s/e	*빈도수*	*총 단계수*
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
합계			2n+2

- 프로그램 1.6의 함수 binsearch를 고려
  - 순서화된 리스트를 탐색, 매개변수는 원소수 n
  - 탄색시간은 searchnum의 위치에 따라 다르다.
  - 단계수를 유일하게 결정하지 않고, 최상, 최악, 평균을 정의한다.
  - 최상 단계수: 주어진 매개변수에 대해 실행될 수 있는 단계수가 최소
  - 최악 단계수: 주어진 매개변수에 대해 실행될 수 있는 단계수가 최대
  - 평균 단계수: 주어진 매개변수를 갖는 인스턴스에 대해 실행되는 평균 단계수

1.4.3 점근 표기법

- 단계수 (step count)
  - 두 프로그램의 시간 복잡도 비교에 사용
  - 인스턴스의 특성에 따른 실행시간의 증가 예측에 사용
  - 그러나, 한 step이 정확한 실행시간을 가지지 않으므로 두 프로그램을 비교하려는 목적에는 유용하지 않다.
    - x=y, x=y+z+(x/y)+(x*y*z-x/z)를 한단계로 계산

- step count 대신에 점근적 복잡도를 주로 사용
  - 대략적인 단계수를 의미
  - 100 n + 10, 3n + 3 등을 대략 n의 복잡도로 표현
  - 1000n과 n^2 + 2n을 비교하면 n <= 998에서는 1000n이 크지만 더 큰 n에 대해서는 n^2 + 2n이 더 크다.
  - n의 복잡도보다는 n^2의 복잡도가 n이 커짐에 따라 복잡도 증가

점근 표기법

- Big "oh" : (f of n은 big-oh of g of n)
  - f(n) = O(g(n)) iff ∃positive constants c and n_0 s.t. $f(n) \leq cg(n)$ for all n, $n \geq n_0$
- ex : 3n + 2 =O(n)
  - $3n + 2 \leq 4n$ for all $n \geq 2$
  - 즉, c=4, n_0=2, g(n)=n의 경우이다.

- 예제 1.15

  n ≥ 2, 3n + 2 ≤ 4n 3n + 2 = O(n)

  n ≥ 3, 3n + 3 ≤ 4n 3n + 3 = O(n)

  n ≥ 10, 100n + 6 ≤ 101n 100n + 6 = O(n)

  n ≥ 5, 10n^2 + 4n + 2 ≤ 11n^2 10n^2 + 4n + 2 = O(n^2)

  n ≥ 4, 6*2^n + n^2 ≤ 7*2^n 6*2^n + n^2 = O(2^n)

  n ≥ 2, 3n + 3 ≤ 3n^2 3n + 3 = O(n^2)

  n ≥ 2, 10n^2 + 4n + 2 ≤ 10n^4 10n^2 + 4n + 2 = O(n^4)

n ≥ n0인 모든 n과 임의의 상수 c에 대해

3n + 2 <= c 가 false인 경우가 존재하면 3n+2≠O(1)

10n^2 + 4n + 2 ≠ O(n)

- order of magnitude (오름차순)
  - O(1) : 상수(constant)
  - O(log n) : logarithmic
  - O(n) : 선형(linear)
  - O(nlog n) :log linear
  - $O(n^2)$: 평방형(quadratic)
  - $O(n^3)$: 입방형(cubic)
  - $O(2^n)$: 지수형(exponential)
  - O(n!) : factorial

- *f(n) = O(g(n))*
  - n ≥ n0인 모든 n에 대해 g(n) 값은 f(n)의 상한값
  - g(n)은 조건을 만족하는 가장 작은 함수여야 함

- Therem 1.2: $f(n) = a_m n^m + \cdots a_1 n + a_0$
  - f(n)은 지수가 제일 큰 $O(n^m)$
  - Proof:

$$f(n) \le \sum_{i=0}^{m} |a_i| n^i \le n^m \sum_{i=0}^{m} |a_i| n^{i-m} \le n^m \sum_{i=0}^{m} |a_i|$$

- 주의
  - $10n^2 + 4n + 2 = O(n^2)$
  - $O(n^3)$ 라고는 않함
  - 마찬가지로 $3n + 2 = O(n^2)$ 라고는 않함
  - $O(g(n))$ 중 가장 차수가 낮은 것을 사용

- 정의 [Omega] [f(n) = Ω(g(n)) (하한값)

f(n) = Ω(g(n)) iff c,n_0 > 0 존재, s.t f(n) >= cg(n) 모든 n, n >= n0

예제

n ≥ 1, 3n + 2 ≥ 3n 3n + 2 = Ω(n)

n ≥ 1, 3n + 3 ≥ 3n 3n + 3 = Ω(n)

n ≥ 1, 100n + 6 ≥ 100n 100n + 6 = Ω(n)

n ≥ 1, 10n^2 + 4n + 2 ≥ n^2 100n^2 + 4n + 2 = Ω(n^2)

n ≥ 1, 6*2n + n^2 ≥ 2^n 6*2^n + n^2 = Ω(2^n)

g(n) : f(n)의 하한 값(가능한 큰 함수)

- 정의 [Theta] [f(n) = Θ(g(n))

f(n) = Θ(g(n)) iff C1, C2, n0 > 0 존재, s.t C1g(n) ≤ f(n) ≤ C2g(n), 모든 n, n >= n0

- 예제
- n ≥ 2, 3n <= 3n + 2 <= 4n ➔ 3n + 2 = Θ(n)

  - c1 = 3, c2 = 4, n0 = 2

  3n + 3 = Θ(n)
- 10n^2 + 4n + 2 = Θ(n^2)
- 6*2^n + n^2 = Θ(2^n)
- 10*log n + 4 = Θ(log n)

- g(n)이 f(n)에 대해 상한 값과 하한 값을 모두 가지는 경우
- g(n)의 계수는 모두 1 !!

- 예제
- *Tsum = 2n + 3 Tsum(n) = Θ(n)*
- *Trsum(n) = 2n + 2 = Θ(n)*
- *Tadd(rows,cols) = 2rows cols + 2rows + 1 = Θ(rows cols)*

- 점근적 복잡도(asymptotic complexity:O,Ω,Θ)는 정확한 단계수의 계산 없이 쉽게 구함

- 예제 [행렬 덧셈의 복잡도]

문장	점근적 복잡도
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
합계	*Θ(rows cols)*

- 예제 [이진 탐색]
  - 프로그램 1.6, while loop는 $\lceil \log_2(n+1) \rceil$ 이므로, 최악 *Θ(log n)*

1.4.4 실용적인 복잡도

- 두 프로그램의 성능비교를 위해서는, 복잡도와 n 고려
  - P는 10^6 n, Q는 n^2 일 때, n <= 10^6일 경우는 Q가 더 빠르므로 Q를 사용

- 현실적으로 볼 때, 작은 복잡도 (n, nlog n, n^2, n^3)이 유용하고, n^10, 2^n등은 n=100경우에 오랜 시간 걸림

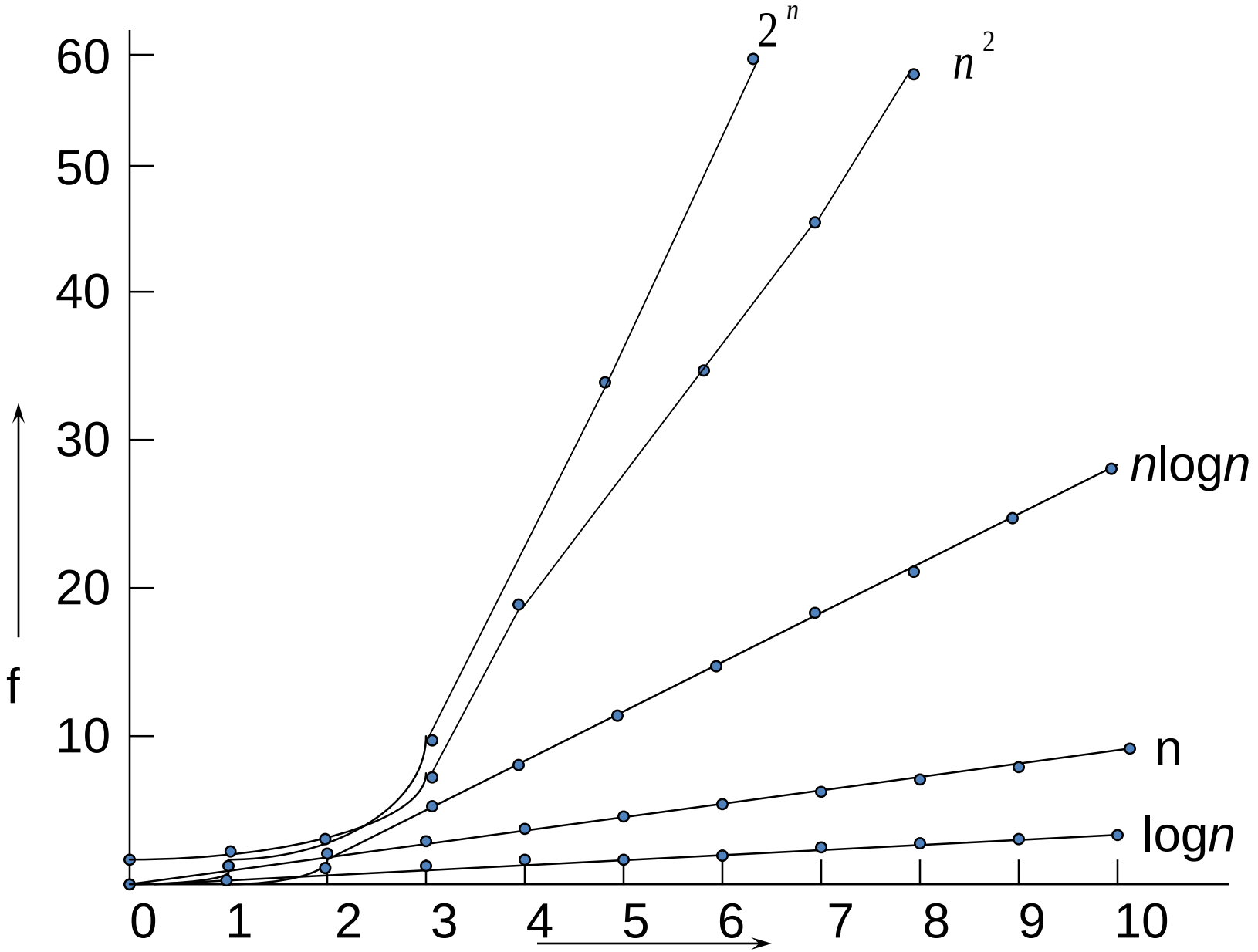


그림 1.3 함수 값의 그래프

1.5 Performance Measurement

- 성능측정
  - 프로그램의 수행에 요구되는 실제적인 메모리와 시간을 얻는 방법,
  - 함수 clock 또는 time사용, 43 페이지

- 최악의 경우, 탐색하는 시간 측정

```
int seqsearch(int list[], int searchnum, int n)
{
        int I;
        list [n] = searchnum ;
        for (I=0; list[I] != searchnum; I++)
                ;
        return ((I<n) ? I: -1) ;
}
```

```
#include <stdio.h>
#include <time.h>
#define MAX_SIZE 1001
#define ITERATIONS 16
int search (int [], int, int)
void main (void)
{  int I, j, position;
  int list[MAX_SIZE];
  int sizelist[] = {0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 400, 600, 800, 1000} ;
  int numtimes[] = {30000, 12000, 6000, 5000, 4000, 4000, 4000, 3000, 3000, 2000, 2000, 1000,
                          500, 500, 500, 200};
  clock_t start, stop;   double duration, total;
  for (I=0; I < MAX_SIZE; I++)  list[I] = I ;
  for (I=0; I < ITERATIONS; I++) {
     start = clock() ;
     for (j=0; j < numtimes[I]; j++)
        position = seqsearch(list, -1, sizelist[I]) ;
     stop = clock() ;
     totoal = ((double)(stop - start)) / CLK_TCK;
     duration = total / numtimes[I] ;
     printf("%5d  %d  %d  %f  %f\n", sizelist[I], numtimes[I], (int)(stop – start), total, duration) ;
     list[sizelist[I]] = sizelist[I]; /*값의 재설정*
   }
}
```