제 1 장 기본 개념

1.1 개요 : 시스템 생명 주기

- 프로그램은 매우 복잡하게 상호 작용하는 부품들로 구성된 시스템이다 !!

- 시스템 생명 주기
  - 요구 사항 (requirements)
  - → 분석 (analysis)
  - → 설계 (design)
  - → 정제 및 코딩 (refinement and coding)
  - → 검증 (verification)

□ 요구 사항

- 프로젝트들의 목적을 정의한 명세들의 집합
  - 프로그래머에게 주어진 데이터 입력과 프로그래머가 생성해내야 하는 결과(출력)에 관한 정보 기술
  - 모든 경우에 대한 입력과 출력의 기술을 정밀하게 작성해야 한다

□ 분석

- 문제들을 실제로 다룰 수 있을 정도의 작은 단위들로 나눈다
- 두 가지 접근 방법
  - 상향식 접근 방법
    : 코딩에 주안점을 둔 비구조적 방법
  - 하향식 접근 방법
    : 최종 결과 프로그램을 다룰 수 있을 정도의 프로그램 단위로 분리 …

□ 설계

- 프로그램이 필요로 하는 데이터 객체와 이들 위에서 수행될 연산들의 관점에서 시스템에 접근
  - 추상 데이터 타입 (abstract data type)
  - 알고리즘의 명세와 알고리즘 설계 기법의 고려
- 사용하는 프로그래밍 언어와는 별개이므로 구현을 위한 결정 사항은 뒤로 연기

□ 정제 및 코딩

- 데이터 객체에 대한 표현 선택
  - 데이터 객체의 표현 방법이 알고리즘의 효율성을 결정한다
- 수행되는 연산에 대한 알고리즘 작성

□ 검증

- 정확성 증명
- 테스트
- 오류 제거

1.2 알고리즘 명세

1.2.1 소 개

정의 : 알고리즘

특정한 일을 수행하는 명령어들의 유한 집합으로 다음의 조건들을 만족해야 한다

- 입력 : 외부에서 제공하는 자료가 0 개 이상
- 출력 : 적어도 한 가지 이상의 결과를 생성
- 명확성 : 명확하고 모호하지 않아야 한다
- 유한성 : 한정된 수의 단계 뒤에는 반드시 종료
- 유효성 : 원칙적으로 종이와 연필만으로 수행할 수 있도록 기본적이어야 한다

☐ 전산학에서는 (엄밀한 의미로) 알고리즘과 프로그램을 구별한다

☒ 프로그램

- 유한성을 만족하지 않는다
- ✓ 운영 체제 (operating system)

□ 알고리즘 명세

- 자연어 사용
  - 명확성을 유지해야
- 그래프 표현법
  - ✓ 흐름도 (flowchart)
- 프로그래밍 언어 사용
  - 작고 단순한 알고리즘의 경우에 유리
  - ✓ C, C++, Java, …
  - ✓ Pascal
  - ✓ Sparc

▶ 알고리즘 예 1 → 페이지 5 ~ 6, 프로그램 1.1 : 선택 정렬 알고리즘

- $n \geq 1$개의 서로 다른 정수를 정렬하는 프로그램의 작성

- 1 단계 :
  - "정렬되지 않은 정수들 중에서 가장 작은 값을 찾아서 정렬된 리스트 다음 자리에 놓는다"

→ 2 단계 : 저장 문제 해결
  - 정수들이 배열, *list*에 저장된다고 가정

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

→ 3 단계 : 알고리즘 완성

– 최소 정수의 탐색, 값의 교환 작업 명시

```
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);
        /* temp = list[i]; list[i] = list[min]; list[min] = temp; */
    }
}
```

→ 4 단계 : 프로그램 완성

– 페이지 7 ~ 8, 프로그램 1.3 : 선택 정렬

▶ 알고리즘 예 2 → 페이지 9, 프로그램 1.6 : 순서 리스트 탐색

- $n \geq 1$개의 서로 다른 정수가 정렬되어 배열 *list*에 저장되어 있을 때 *searchnum*이 *list*에 있는지 검사하는 알고리즘
  - 존재하면 *list*[*i*] = *searchnum*인 인덱스 *i*를 반환하고 존재하지 않으면 -1을 반환

- 1 단계 : 알고리즘의 기술
  - *left*, *right*가 각각 탐색하고자 하는 배열의 왼쪽, 오른쪽 끝 지점을 가리킬 때 (초기값은 *left* = 0, *right* = *n* - 1) *middle* = (*left* + *right*) / 2로 설정

    searchnum < *list*[*middle*] ⇒ *searchnum*이 *list*[*left*]와 *list*[*middle* - 1] 사이에 있으므로 *right* ← *middle* - 1

    searchnum = *list*[*middle*] ⇒ *middle*을 반환

    searchnum > *list*[*middle*] ⇒ *searchnum*이 *list*[*middle* + 1]과 *list*[*right*] 사이에 있으므로 *left* ← *middle* + 1
  - *searchnum*이 찾아지지 못하고 검색할 정수가 있으면 *middle*을 계산하여 탐색을 계속

- 2 단계 : *searchnum*과 *list*[*middle*]의 비교 구체화

```
while (there are more integers to check) {      → 검사할 정수가 남아있는지 결정
    middle = (left + right) / 2;
    if (searchnum < list[middle])
        right = middle – 1;
    else if (searchnum == list[middle])          } searchnum과 list[middle]의 비교
        return middle;
    else
        left = middle + 1;
```

- 3 단계 : 알고리즘 구체화

```
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;
}
```

1.2.2 순환 알고리즘

- 순환 기법(recursive mechanism)
  - 직접 순환
    - 수행이 완료되기 전에 자기 자신을 다시 호출
  - 간접 순환
    - 호출 함수를 다시 호출하게 되어 있는 다른 함수를 호출
  - → 매우 복잡해질 과정들을 간단히 표현할 수 있다

- 순환에 의해 잘 해결되는 문제들
  - 문제 자체가 순환적으로 정의되는 경우
    - ✓ factorial : $n! = n \times (n - 1)!$
  - 알고리즘이 다루는 자료 구조가 순환적으로 정의되어 있는 경우
    - ✓ list, binary tree, ...

▶ Iterative binary search algorithm

→ 페이지 9, 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;
}
```

▶ Recursive binary search algorithm

→ 페이지 11, 프로그램 1.7 : 이진 탐색에 대한 순환 구현

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

1.3 데이터 추상화

□ 데이터 타입 & 추상 데이터 타입

- 자료구조와 알고리즘은 서로 어느 한쪽이 없이는 이해를 할 수 없는 하나의 단위

데이터 타입 (data type)

- 데이터 타입은 객체들(objects)과 이들 객체들에 대해 동작하는 연산들(operations)의 집합이다

✓ *int*

– { 0, +1, -1, +2, -2, …, INT_MAX, INT_MIN }의 객체로 구성

– 최소 산술 연산자 : +, -, *, /, %

✓ 객체의 표현 방법의 이해 ?

– 알고리즘 기술시에 표현방법을 이용하면 유용할 수도 있지만 위험하다

– 객체의 표현 방법이 변경되면 이를 이용하는 루틴도 변경해야 한다

추상 데이터 타입(abstract data type, ADT)

- 추상 데이터 타입은 객체의 명세와 이들 객체에 대한 연산의 명세가 객체의 표현과 연산의 구현으로부터 분리된 방식으로 구성된 데이터 타입이다

✓ Natural Number

▶ Natural Number 추상 데이터 타입

→ 페이지 17 ~ 18, 구조 1.1 : Natural Number 추상 데이터 타입

struct *Natural_Number*

objects : '0에서 시작해서 컴퓨터 상의 최대 정수값(*INT_MAX*)까지 순서화된 정수의 부분 범위이다.

functions :

*Nat_No*의 모든 원소 *x*, *y*, 그리고 *Boolean*의 원소 *TRUE*, *FALSE*에 대해, 여기서, +, –, <, 그리고 ==는 일반적인 정수 연산이다.

Nat_No Zero()	::=	0
Boolean Is_Zero()	::=	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*

end *NaturalNumber*

1.4 성능 분석

- 성능 평가의 단계
  - 사전 예측 - 성능 분석 (performance analysis)
    - 컴퓨터와 관계없는 시간과 공간의 추산
  - 이후 검사 - 성능 측정 (performance measurement)
    - 컴퓨터에 의존적인 실행 시간

- 기본적인 판단 기준
  - 프로그램이 원래의 명세와 부합하는가?
  - 정확하게 작동하는가?
  - 프로그램을 어떻게 사용하고, 또 어떻게 수행하는지에 관한 문서화가 프로그램 내에 되어 있는가?
  - 프로그램에서 논리적인 단위를 생성하기 위해 함수를 효과적으로 사용하는가?
  - 프로그램의 코드가 읽기 쉬운가?

□ 추가된 판단 기준

- 프로그램이 주기억 장치와 보조기억장치를 효율적으로 사용하는가?
- 작업에 대한 프로그램의 실행 시간은 받아들일 만한가?

공간 복잡도 (space complexity)

- 프로그램을 실행시켜 완료하는데 필요한 공간의 양

시간 복잡도 (time complexity)

- 프로그램을 실행시켜 완료하는데 필요한 컴퓨터 시간의 양

1.4.1 공간 복잡도

- 프로그램이 필요로 하는 공간
  - 고정 부분, *c*
    - 프로그램의 입출력 특성(횟수나 크기)에 관계 없는 공간 요구
    - 명령어 공간, 단순 변수, 고정 크기 구조화 변수, 상수 공간 등
  - 가변 부분, $S_p(I)$
    - 문제의 특성 인스턴스(*I*)에 의존하는 크기를 가진 구조화 변수들이 필요로 하는 공간
      : 입출력의 횟수, 크기, 값 …
    - 순환 호출을 할 경우에 요구되는 추가 공간 포함

⇒ 프로그램 *P*의 총 공간 요구, $S(P)$

- $S(P) = c + SP(I)$

▶ 예 1 → 페이지 21, 프로그램 1.9 : 단순 산술 함수

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

- 3개의 단순 변수를 입력으로 받아 하나의 단순 값을 출력

→ 고정 공간 요구만을 가짐

→ $S_{abc}(I) = 0$

▶ 예 2 → 페이지 21, 프로그램 1.10 : 리스트의 수치값을 합산하기 위한 반복 함수

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

- 입력으로 배열을 가짐

→ 배열이 함수로 어떻게 전달되는지에 달려 있다

- Pascal의 경우 : call by value 방식으로 배열을 전달
  : $S_{sum}(I) = S_{sum}(n) = n$
- C 언어의 경우 : 배열의 첫번째 요소의 주소 전달 → 배열은 복사되지 않음
  : $S_{sum}(I) = S_{sum}(n) = 0$

▶ 예 3 → 페이지 22, 프로그램 1.11 : 리스트의 수치값을 합산하기 위한 순환 함수

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

- 순환 함수 → 매개 변수, 지역 변수, 순환 호출 시에 복귀 주소를 저장해야 한다

→ 한번 호출될 때의 필요 공간

– 2개의 매개변수 저장공간 + 복귀 주소 공간 = (4byte(*list*) + 4byte(*n*) + 4byte(복귀 주소)) = 12byte

→ 호출 수 × 호출 당 필요 공간 = $n \times$ 12byte

1.4.2 시간 복잡도

- 프로그램 P에 소요되는 시간, $T(P)$
  - $T(P)$ = 컴파일 시간 + 실행 시간(T_P)
  - $T_P(n)$의 측정, n : 인스턴스 특성

- T_P의 결정 방법
  - 물리적 실행 시간 측정
  - 연산의 총 횟수 측정
    - 프로그램을 어떻게 상이한 단계로 분할할 것인가?

프로그램 단계

- 프로그램 단계는 인스턴스 특성에 독립적인 실행 시간을 가지는 구문적으로나 의미적으로 합당한 프로그램의 세그먼트이다
- → 프로그램 세그먼트에 의해 표현되는 계산양은 다른 세그먼트에 의해 표현되는 계산양과 다르다
- → 한 단계로 간주되는 각 명령문을 실행하는데 필요한 시간이 인스턴스 특성에 독립적이어야 한다

□ 프로그램의 단계 수를 결정하는 두 가지 방법

- 프로그램에 count라는 새로운 변수를 생성
- 각 문장의 총 단계 수를 나열하는 테이블을 구성

□ 프로그램의 시간 복잡도

- 프로그램의 기능을 수행하기 위해 프로그램이 취한 단계수로 표현
- 인스턴스 특성 중 중요한 특성만을 선택
- → 입력 수의 증가에 따라 시간이 얼마나 증가하는가
- → 특정한 입력의 크기가 증가함에 따라 연산 시간이 얼마나 증가하는가
- 문제에 대한 어떤 특성이 사용될 것인가를 정확하게 알아야

▶ 페이지 26, 프로그램 1.15 : 행렬의 덧셈
페이지 27, 프로그램 1.16 : count 문이 첨가된 행렬의 덧셈

```
void add (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++)
        for (j = 0; j < cols; j++)
            c[i][j] = a[i][j] + b[i][j];
}
```

```
void add (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++) {
        count++;
        for (j = 0; j < cols; j++) {
            count++;
            c[i][j] = a[i][j] + b[i][j];
            count++;
        }
        count++;
    }
    count++;
}
```

▶ 페이지 28, 그림 1.4 : 행렬 덧셈에 대한 단계수 테이블

	s/e	빈도수	총 단계수
void *add* (int *a*[][MAX_SIZE], …)	0	0	0
{	0	0	0
int *i*, *j*;	0	0	0
for (*i* = 0; *i* < *rows*; *i*++)	1	$rows+1$	$rows+1$
for (*j* = 0; *j* < *cols*; *j*++)	1	$rows \cdot (cols + 1)$	$rows \cdot cols + rows$
c[*i*][*j*] = *a*[*i*][*j*] + *b*[*i*][*j*];	1	$rows \cdot cols$	$rows \cdot cols$
}	0	0	0
	$2rows \cdot cols + 2rows + 1$		

- 단계수를 유일하게 결정할 때 발생하는 매개 변수 선택의 부적절함을 세 가지 경우의 단계수를 정의함으로써 해결
  - 최상 단계수
    - 주어진 매개 변수에 대해 실행될 수 있는 단계수가 최소인 경우
  - 최악 단계수
    - 주어진 매개 변수에 대해 실행될 수 있는 단계수가 최대인 경우
  - 평균 단계수
    - 주어진 매개 변수를 가지는 인스턴스에 대해 실행되는 평균 단계수

1.4.3 점근 표기법

- 정확한 단계 수의 측정
  - 단계 개념 자체가 부정확하므로 낭비 초래
  - ✓ $(3n + 3)$ Vs. $(100n + 10)$

- 균형 분기점 (break-even point)
  - ✓ $(c_1n^2 + c_2n)$ Vs. (c_3n)
    - 복잡도가 c_3n인 프로그램이 복잡도가 $c_1n^2 + c_2n$인 프로그램보다 빠를 수 있는 조건이 되는 n이 존재

- 점근 표기법들 (asymptotic notations)
  - big oh : $O(n)$
  - omega : $\Omega(n)$
  - theta : $\Theta(n)$

Big “oh”

- upper bound
- $f(n) = O(g(n))$
  - 모든 $n, n \geq n_0$에 대해 $f(n) \leq cg(n)$인 조건을 만족하는 두 양의 상수 c와 n_0가 존재
- → $f(n) = a_m n^m + \ldots + a_1 n + a_0$이면 $f(n) = O(n^m)$

✓ 예제

- $3n + 2 = O(n)$
- $1000n^2 + 100n - 5 = O(n^2)$
- $6 \times 2^n + n^2 = O(2^n)$
- $10n^2 + 4n - 5 = O(n^4)$

✓ $O(1)$ $O(\log n)$ $O(n)$ $O(n\log n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$

Omega

- lower bound
- $f(n) = \Omega(g(n))$
  - 모든 n, $n \geq n_0$에 대해 $f(n) \geq cg(n)$인 조건을 만족하는 두 양의 상수 c와 n_0가 존재

→ $f(n) = a_m n^m + \ldots + a_1 n + a_0$, $a_m > 0$이면 $f(n) = \Omega(n^m)$

✓ 예제

- $3n + 2 = \Omega(n)$
- $10n^2 + 4n + 2 = \Omega(n^2) = \Omega(n) = \Omega(1)$
- $3n + 2 = \Omega(n)$
- $6 \times 2^n + n^2 = \Omega(2^n) = \Omega(n^{100}) = \Omega(n^{50.2}) = \Omega(n^2) = \Omega(n) = \Omega(1)$

Theta

- $f(n) = \Theta(g(n))$
  - 모든 n, $n \geq n_0$에 대해 $c_1 g(n) \leq f(n) \leq c_2 g(n)$인 조건을 만족하는 두 양의 상수 c_1, c_2와 n_0가 존재

→ $f(n) = a_m n^m + \ldots + a_1 n + a_0$, $a_m > 0$이면 $f(n) = \Theta(n^m)$

✓ 예제

- $3n + 2 = \Theta(n) \neq \Theta(1)$

▶ 페이지 35, 그림 1.5 : 행렬 덧셈의 시간 복잡도

	점근적 복잡도
void *add* (int *a*[][MAX_SIZE], …)	0
{	0
int *i*, *j*;	0
for (*i* = 0; *i* < *rows*; *i*++)	$\Theta(rows)$
for (*j* = 0; *j* < *cols*; *j*++)	$\Theta(rows \cdot cols)$
c[*i*][*j*] = *a*[*i*][*j*] + *b*[*i*][*j*];	$\Theta(rows \cdot cols)$
}	0
	$\Theta(rows \cdot cols)$

✍ 이진 탐색의 시간 복잡도

```
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;
}
```

- 인스턴스 특성 = 리스트 원소의 수 n
- while 루프
  - 매 반복에 $\Theta(1)$의 시간
  - 루프의 반복 횟수

    : 매 반복마다 list의 세그먼트가 반으로 줄어듦

    : $1 \rightarrow n / 2 \rightarrow n / 4 \rightarrow n / 8 \rightarrow \dots \rightarrow 1$

    : $1 \rightarrow n / (2^1) \rightarrow n / (2^2) \rightarrow n / (2^3) \rightarrow \dots \rightarrow n / (2^{i-1}) = 1$ ← i는 반복 횟수

    : $2^{i-1} = n \rightarrow i - 1 = \log_2 n \rightarrow i = \log_2 n + 1$

▶ 페이지 36, 그림 1.6 : 매직 스퀘어 (magic square)

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*

- n이 홀수일 때 매직 스퀘어 규칙 (by Coexter)
  – 첫번째 행의 중앙에 1을 넣는다
  → 왼쪽 대각선 방향으로 올라가면서 빈 자리에 1씩 큰 수를 넣는다
  → 만약 정방형 밖으로 벗어나면 정방형의 반대편 자리에서 계속한다
  : 상단을 벗어나면 같은 열의 최하단으로, 왼쪽으로 벗어나면 같은 행의 제일 오른쪽으로 이동한다
  → 이동하려는 자리에 숫자가 이미 채워져 있으면 바로 밑으로 가서 계속한다

▶ 페이지 38, 프로그램 1.22 : 매직 스퀘어 프로그램

```
for (i = 0; i < size; i++)
        for (j = 0; j < size; j++)
                square[i][j] = 0;
square[0][(size - 1) / 2] = 1;          /* 첫번째 행의 중앙에 1을 넣는다 */
i = 0; j = (size - 1) / 2;
for (count = 2; count <= size * size; count++) {
        row = (i – 1 < 0) ? (size – 1) : (i – 1);                /* 위로 */
        column = (j – 1 < 0) ? (size – 1) : (j – 1);             /* 왼쪽으로 */
        if (square[row][column]) i = (++i) % size;               /* 아래로 */
        else { i = row; j = (j – 1 < 0) ? (size – 1) : —j ; }
        square[i][j] = count;
}
/* 출력 */
printf("Magic Square of size %d : \n\n", size);
for (i = 0; i < size; i++) {
        for (j = 0; j < size; j++) printf("%5d", square[i][j]);
        printf("\n");
}
```

- 복잡도 계산
  - 첫번째 for 문의 복잡도 = $\Theta(n^2)$ 두번째 for 문의 복잡도 = $\Theta(n^2)$
    세번째 for 문의 복잡도 = $\Theta(n^2)$
  - → 프로그램의 복잡도 = $\Theta(n^2)$

1.4.4 실용적인 복잡도

▶ **페이지 40, 그림 1.7 : 함수 값**

		인스턴스 특성 n					
시간	이름	1	2	4	8	16	32
1	상수	1	1	1	1	1	1
$\log n$	로그형	0	1	2	3	4	5
n	선형	1	2	4	8	16	32
$n\log n$	로그 선형	0	2	8	24	64	160
n^2	평방형	1	4	16	64	256	1024
n^3	입방형	1	8	64	512	4096	32768
2^n	지수형	2	4	16	256	65536	4294967296
$n!$	계승형	1	2	24	40326	20922789888000	26313×10^{33}

1.5 성능 측정

☐ 성능 측정

- 프로그램이 공간 및 시간 요구량을 구하는 것
- 특정 컴퓨터, 컴파일러, 사용되는 옵션에 따라 다르다

☐ 프로그램의 계산 시간을 측정하는데 초점을 둠

- 시간 함수 (clocking function) 필요
  - ✓ *time()* :
    - : 페이지 44, 프로그램 1.23 참조

☐ 테스트 데이터의 생성

- 프로그램의 최악의 성능을 초래하는 데이터 집합을 생성하는 것이 항상 쉽지는 않다
- 최악의 성능을 측정하기 위한 또 다른 접근법을 사용
  - – 관심 있는 인스턴스 특성 값들에 대해 적당히 큰 규모의 무작위 테스트 데이터를 생성
  - → 이 테스트 데이터 각각에 대해 실행 시간을 구한다
- 실험을 위해 생성되어야 할 데이터를 결정하기 위해 사용되는 알고리즘을 분석해야