8장. 알고리즘과 효율

❑ 학습목표

- 알고리즘의 정의를 이해한다.
- 알고리즘의 정확성 증명방법을 이해한다.
- 빅 오 기호에 의한 알고리즘 효율분석 기법을 이해한다.
- 선형, 제곱, 지수 알고리즘 차이가 의미하는 바를 이해한다.

❑ 알고리즘

- 하나의 문제를 해결할 수 있는 여러 알고리즘
- 효율성(문제해결에 걸리는 시간)이 선택의 기준
- 빅 오 기호를 써서 알고리즘의 시간적인 효율을 분석할 수 있다

알고리즘

- 건포도 케이크 만들기
  - 케이크: 출력
  - 재료: 입력
  - 그릇, 오븐, 요리사: 하드웨어

알고리즘

❑ 알고리즘

- 추상적, 개략적으로 기술한 조리법
- 알고리즘을 구체적으로 표현한 것이 프로그램
- Mohammed Al Khowarizmi: 십진수의 사칙연산 방법을 단계적으로 기술

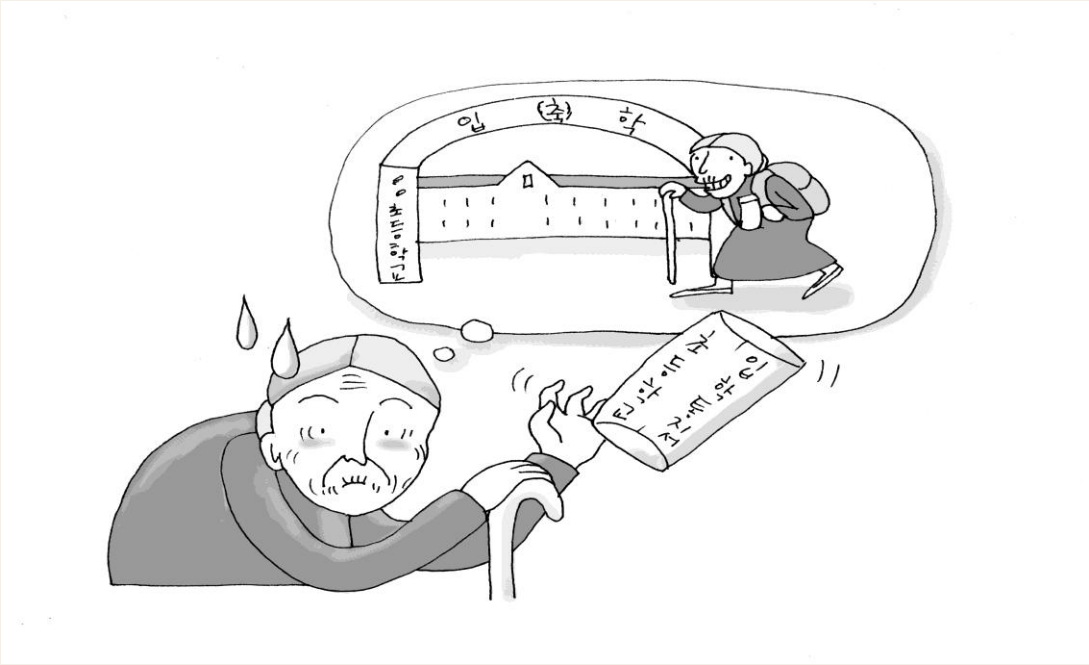
❑ 상세성

- Level of Detail cf. Level of Abstraction
- “분말설탕을 넣고 젓는다” -> “설탕을 반 숟갈 넣고 젓는다. 다시 설탕을 반 숟갈 넣고 젓는다.”-> “설탕 2,365 알갱이를 넣는다. 왼팔을 135도 각도로 해서 초당 30cm 속도로 젓는다.”
- 기본 동작을 언급하되 너무 추상적이거나 너무 구체적이어서는 안된다.

알고리즘의 정확성

❑ 부정확성의 결과

- 2002년 경기도 고교 추첨
- 1960년대 매리너 우주선의 실종
- 1981년 캐나다 퀘벡 주 지방선거 소수당이 이긴 것으로 공표
- 1990년 덴마크의 107세 된 할머니에게 초등학교 입학 통지서
- 경우에 따라서 소프트웨어 오류는 생사의 문제나 재앙의 문제와 직결된다.



프로그램 오류

- 정확성(Correctness)
  - 허용된 입력(Legal Input)에 대해서 제대로 동작해야 함
  - ‘정확히 우리가 기대하는 것’을 수행해야 함
- 문법 오류(Syntax Error)
  - 명령문 끝에 세미콜론
  - 프로그래머와 컴퓨터 간에 정확한 의사소통
  - 모호성(Ambiguity)을 배제하기 위한 약속
- 의미상의 오류(Semantic Error)
  - 프로그래머와 컴파일러 사이의 오해
  - A**B는 A의 B승? AB? if (A = 5)? if (A = = 5)?

프로그램 오류

- 논리적인 오류(Logical Error, Algorithmic Error)
  - 가장 치명적인 오류

- 다음 문장 중에 소득이라는 문자가 들어간 문장은 몇 개인가.

  - 소득이 부진했던 지난해 경제성장률이 외환위기 이후 최저인 3.1%에 불과한데, 국민소득이 10.1%가 늘어나게 된 이유는 국민소득의 개념 및 계산방법의 차이에 있다.
  - "소득이라는 문자가 나오고 이어서 문장의 끝을 의미하는 마침표가 나오면 한번 나온 것이니 파일 끝까지 읽으면서 카운트를 더해간다"
  - 이 논리에 의하면 위 예에는 소득이라는 단어가 들어간 문장이 세 개.

알고리즘의 정확성

❑ 샘플 테스트

- 부정확성을 증명(오류가 있음을 증명)
- 정확성을 증명할 수는 없음 (오류가 없음을 증명할 수는 없음)
- 정확성 = 가능한 모든 입력에 대해 프로그램이 제대로 작동

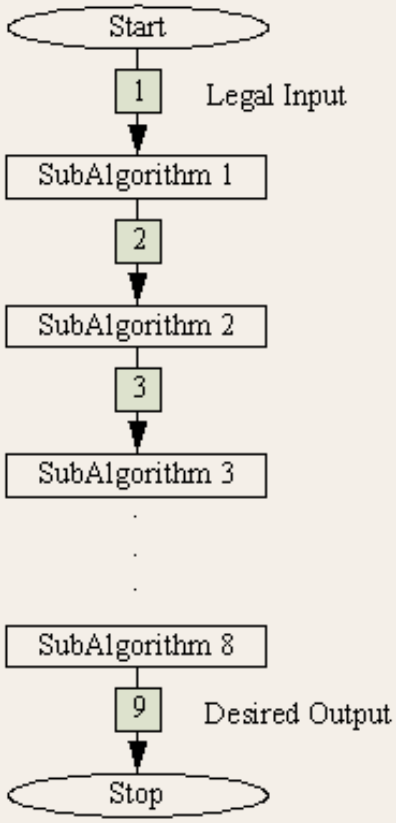
❑ 부정확한(Incorrect) 알고리즘의 결과

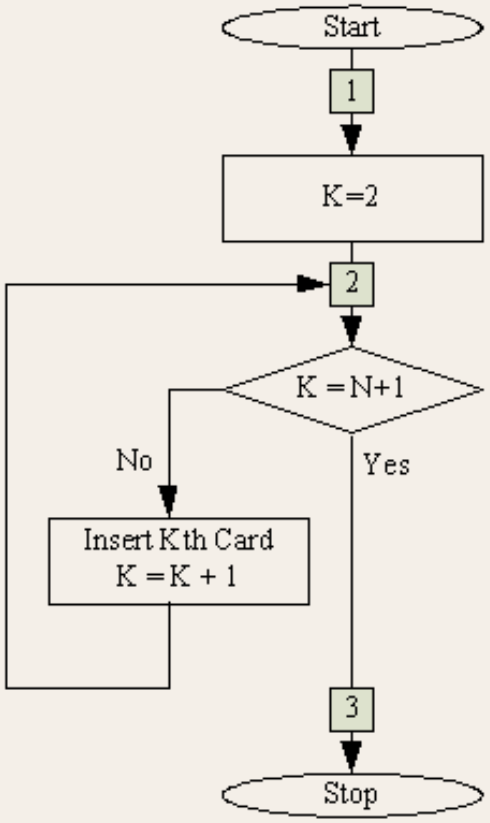
- 어보트(ABORT: Abnormal Termination, ABEND: Abnormal End)
- 무한 루프(Infinite Loop)
- 정상적으로 끝났지만 잘못된 결과를 출력

정확성 증명

❑ 분할정복 전략

- 부분 알고리즘(Sub-Algorithm)으로 분할
- 부분 알고리즘의 전후에 단언(斷言, 잘라 말함, Assertion)이 옳음을 증명
- 단언은 “이 상태에 이르면 이런 사실이 성립한다”의 형태로 제시
- 단언 = 인베리언트(Invariant, 불변의 사실).

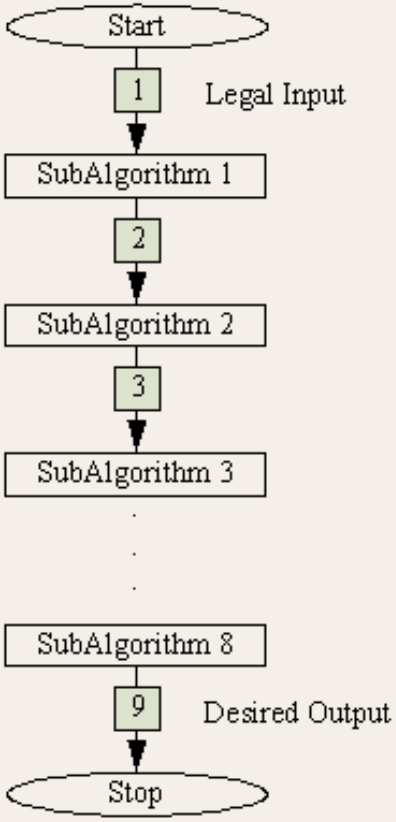


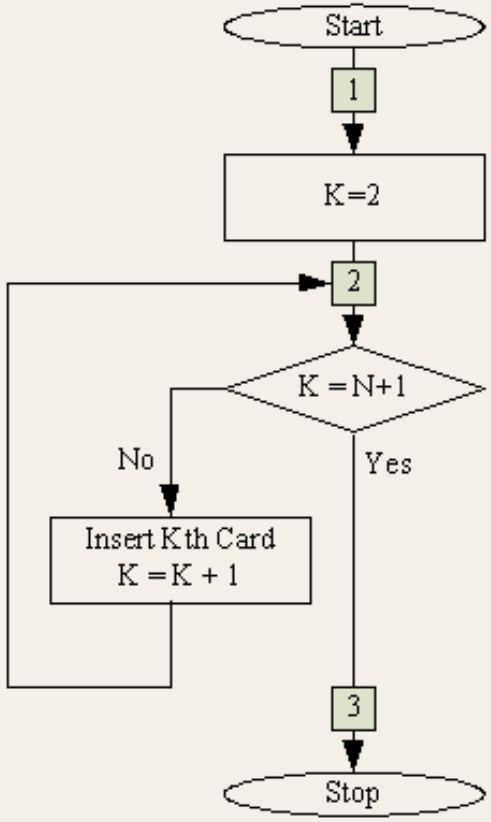


정확성 증명

단언

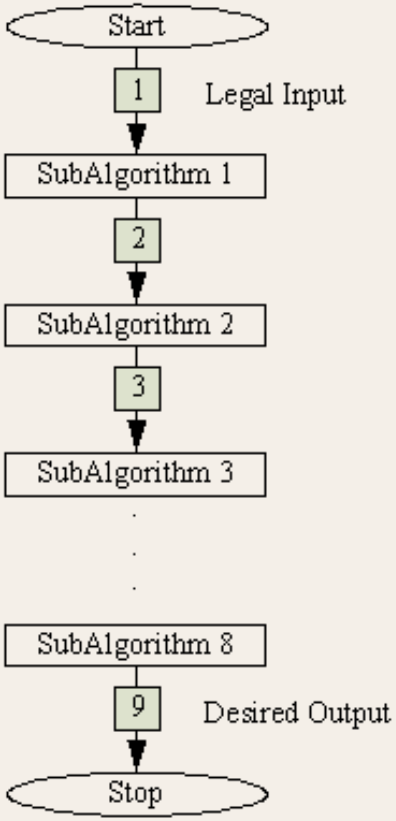
- 1번 단언= “허용된 입력이 들어온다”. 9번 단언 = “원하는 출력이 나온다”
- 1번 단언이 사실인 상태에서 서브 알고리즘 1을 거치면 반드시 2번 단언도 사실임을 증명. 삼단 논법과도 유사
- 단계별로 단언이 사실이라는 증명과 함께, 해당 단계가 반드시 종료함을 보여야 한다.

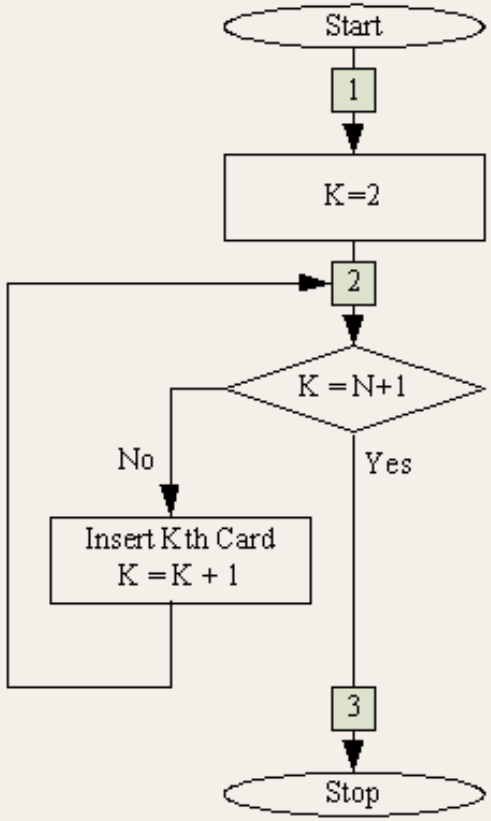




정확성 증명

- 1번 단언: "입력이 1부터 10사이의 숫자를 가진 카드다"
- 2번 단언: "K번째 카드 즉 현재 집어낸 카드의 왼쪽에 있는 모든 카드는 정렬된 상태이다" = 루프 인베리언트(Loop Invariant)
- 3번 단언: "카드 N개가 모두 정렬된 상태다"





알고리즘의 효율

❑ 공간적 효율성과 시간적 효율성

- 공간적 효율성은 얼마나 많은 메모리 공간을 요하는가를 말한다.
- 시간적 효율성은 얼마나 많은 시간을 요하는가를 말한다.
- 효율성을 뒤집어 표현하면 복잡도(Complexity)가 된다. 복잡도가 높을수록 효율성은 저하된다.

❑ 시간적 복잡도 분석

- 하드웨어 환경에 따라 처리시간이 달라진다.
  - 부동소수 처리 프로세서 존재유무, 나눗셈 가속기능 유무
  - 입출력 장비의 성능, 공유여부
- 소프트웨어 환경에 따라 처리시간이 달라진다.
  - 프로그램 언어의 종류
  - 운영체제, 컴파일러의 종류
- 이러한 환경적 차이로 인해 분석이 어렵다.

효율분석

점근적 복잡도(Asymptotic Complexity)

- 실행환경과 무관하게 개략적으로 분석
- 입력 데이터의 수 = 데이터 크기(Size) = N
- 실행에 걸리는 시간을 N의 함수로 표시
- 단, N이 무한대로 갈 때의 효율을 표시함으로써 환경적 변수에 의한 영향이 무시됨

코드 8-1: 연결 리스트 데이터 N 개를 출력하는 함수

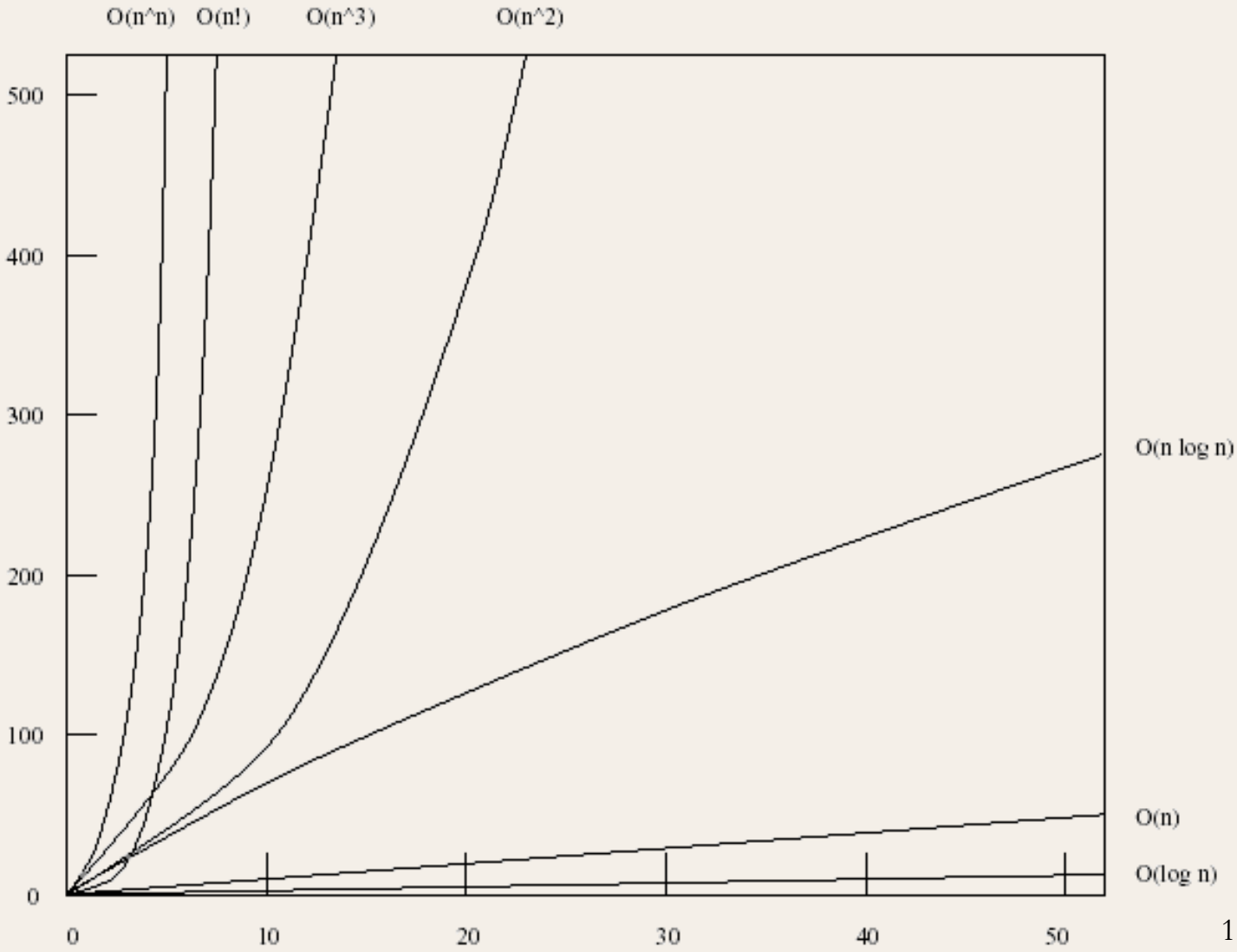
```
void Display(Nptr Head)
{ Nptr Temp = Head;              헤드 포인터를 템프로 복사
  while (Temp != NULL)           템프가 널이 아닐 때까지
  {  printf("%d", Temp->Data);   템프가 가리키는 노드의 데이터를 출력
     Temp = Temp->Next;          템프를 다음 노드로 이동
  }}
```

복잡도

- 할당 한번 시간을 a, 비교 한번 시간을 b, 출력 한번 시간을 c라고 가정
- 정확한 총 실행시간은 a(N+1) + bN + cN = (a+b+c)N + a
- 단순히 “실행시간이 N에 비례” 하는 알고리즘이라고 말함.

입력크기에 따른 함수값 비교

- 입력크기에 따른 함수값 비교



함수값의 크기비교

- $\log N < N < N\log N < N^2 < N^3 < \ldots < 2^N$

- 1000 MIPS 컴퓨터의 연산시간

		$1.3N^3$	$10N^2$	$47N\lg N$	$48N$
연산시간	N = 1,000	1.3 sec	10 msec	0.4 msec	0.048 msec
	N = 10,000	22 min	1 sec	6 msec	0.48 msec
	N = 100,000	15 day	1.7 min	78 msec	4.8 msec
	N = 1,000,000	41 year	2.8 hours	0.94 sec	48 msec
	N = 10,000,000	41,000 year	1.7 week	11 sec	.48 sec

미시에서 거시로 이동

❑ 미시에서 거시로 이동

초	1	10	10^2	10^3	10^4	10^5	10^6	10^7	10^8	10^9	10^{10}	..	10^{17}
시간	1 초	10 초	1.7 분	17 분	2.8 시간	1.1 일	1.6 주	3.8 달	3.1 년	31 년	310 년	..	우주 나이

초속(m/sec)	해당속도	예
10^{-10}	3cm/10년	대륙의 표류
10^{-8}	30cm/년	머리카락 생장
10^{-6}	8.6cm/일	빙하 이동
10^{-4}	37cm/시간	장(腸) 운동
10^{-2}	62cm/분	개미
1	3.5km/시	인간 걸음
10^2	352km/시	비행기 프로펠러
10^4	592km/분	우주 왕복선
10^6	992km/초	지구 공전
10^8	99200km/초	광속의 1/3

최악의 경우 효율

- **데이터 N 개 짜리 정렬되지 않은 배열에서 어떤 레코드를 찾기**
  - 처음부터 원하는 키를 가진 레코드가 나올 때까지 순차적으로 읽음.
  - 효율은 키 값의 비교 횟수를 기준으로 평가됨.

- **세 가지 경우**
  - 최선의 경우: 운이 좋으면 첫 번째 데이터가 찾고자 하는 것. 비교 1번
  - 최악의 경우: 배열 끝에 찾고자 하는 것, 비교 N 번.
  - 평균적 경우: 대략 배열의 반 정도만에 찾음. N/2 번 비교

- **알고리즘의 실행시간**
  - 데이터 개수 N의 함수, 데이터의 분포 특성의 함수
  - 최선의 경우(Best Case)가 나오기를 기대하기는 어렵다.
  - 무작위로 분포하기 때문에 평균적인 경우를 정의하기 어렵다.
  - 효율 분석은 항상 최악의 경우(Worst Case)를 기준으로 말함.
  - 최악의 데이터가 들어오더라도 이 시간이면 실행한다고 말하는 것이 안전
  - 실제 효율이 최소한 그보다는 좋음.

빅 오 기호

조건

- 시간적 복잡도를 데이터 크기 N의 함수로 표시하되 계수를 무시한다. 단, N이 무한대로 갈 때를 기준으로 평가한다. 입력 데이터가 최악일 때 알고리즘이 보이는 효율을 기준으로 한다. 이러한 제반 조건을 전제로 효율을 분석하기 위해 사용되는 수학적 도구가 빅 오(Big Oh) 기호다.

수학적 정의

- 임의의 상수 N_0와 c가 있어서 $N \geq N_0$인 N에 대해서 $c \cdot f(N) \geq g(N)$이 성립하면 $g(N) = O(f(N))$ 이라 한다.

예시

- g(N) = 2N + 5라면 이 알고리즘은 O(N)이다.
- 알고리즘의 효율을 나타내는 O( )의 괄호 안에 들어가는 함수 f(N) = N이다.
- N이 충분히 커서 1000보다 크다면 $40 \cdot f(N) \geq g(N)$이 항상 성립한다. N이 1000이라면 벌써 $40 \cdot f(N) = 40 \cdot N = 40000$이고, $g(N) = 2 \cdot 1000 + 5 = 2005$에 불과하기 때문이다. 이 경우의 N_0는 1000으로 c는 40에 해당한다. 이러한 값들은 임의로 할당할 수 있다.

빅 오 기호

- $210N^2 + 90N + 200 = O(N^2)$

  since $210N^2 + 90N + 200 \leq 9000N^2$ for $N \geq 8000$

- $10N^3 + 40N^2 + 90N + 200 = O(N^3)$

  since $10N^3 + 40N^2 + 90N + 200 \leq 1000N^3$ for $N \geq 10000$

- $5N + 15 = O(N)$

  since $5N + 15 \leq 1000N$ for $N \geq 200$

- $5N + 15 = O(N^2)$

  since $5N + 15 \leq 100N^2$ for $N \geq 2000$

빅 오 기호

❑ 빅 오 기호

- 주어진 함수의 가장 높은 차수의 항만 끄집어내되 계수를 1로 하면 됨

❑ 느슨한 정의

- $5N + 12 = O(N) = O(N^2) = O(N^3) = O(N^4) = O(2^N)$
- 실행시간의 상한(Upper Bound)을 표시
- “길어야 이 정도 시간이면 된다”라는 의미
- “길어야 N 시간이면 된다”가 사실이라면 당연히 “길어야 N^2 시간이면 된다”라거나 “길어야 N^3 시간이면 된다”도 사실

❑ 바싹 다가선 정의 (Tight Upper Bound)

- 알고리즘의 특성을 표현하는 데는 바싹 다가선 상한을 사용
- $5N + 12$는 $O(N)$이 가장 바짝 다가선 표현이라 할 수 있

제곱시간 알고리즘

❑ 코드 8-2: 루프 알고리즘 I

```
1 for  i = 1 to N
2   for  j = 2 to (N-1)
3     do
      {  Read A[i, j];
        Write A[i, j];
      }
```

❑ 효율

- 2 ((N-1)-2+1) N = O(N^2)
- 실행시간이 데이터 개수의 제곱에 비례하는 알고리즘
- 제곱시간(Quadratic Time) 알고리즘

선형시간 알고리즘

- 코드 8-3: 루프 알고리즘 II

```
#define MAX 9000000
 for i = 1000 to N-20000
   for  j = 2 to MAX
      do
     {    Read A[i, j];
        Write A[i, j];
      }
```

- 효율
  - (N-20999)·(MAX - 1)·2 = O(N)
  - 선형시간(Linear Time) 알고리즘

상수시간 알고리즘

- 코드 8-4: 단일 명령 집합

```
x = 20;
printf("%d", x);
```

- 효율
  - 두 번 실행
  - $2 = 2N^0 = O(N^0) = O(1)$
  - 데이터 수 N에 무관하게 상수 번 수행되는 알고리즘
  - 상수시간 알고리즘(Constant Time Algorithm)

if 문의 효율

- 코드 8-5: if 문의 효율

```
if (x = = 1)
{  for i = 1 to N
     for  j = 2 to (N-1)
       do
       {  Read A[i, j];
         Write A[i, j];
       }
}
else
{ x = 20;
  printf("%d", x);
}
```

- 효율
  - $O(N^2)$: 최악의 경우에 대해서 평가

로그시간, 지수시간

지수시간 알고리즘 (Exponential Time Algorithm)

- $O(a^N)$
- 사실상 컴퓨터로 실행하기 불가능할 정도의 많은 시간을 요하는 알고리즘

로그 시간 알고리즘(Logarithmic Time Algorithm)

- O(logN)으로서 O(N)보다 매우 빠른 알고리즘
- 로그함수의 밑수(Base)는 아무렇게나 표시
- $\log_b N = \log N / \log b = (1/\log b) \cdot \log N$. 빅 오 기호에서 계수는 무시
- $O(\log_b N) = O(\log N)$
- $\log_2 N$을 lgN으로 표시

복잡도의 상한, 하한

❑ 빅 오메가 Ω(N) (Big Omega)

- 빅 오 기호의 반대 개념
- 알고리즘 수행시간의 하한(Lower Bound)
- “최소한 이만한 시간은 걸린다”
- $N^2 + 100 = \Omega(N^2)$, $N + 1 = \Omega(N^{0.9})$
- 빅 오메가 분석은 주로 문제 자체에 내재하는 속성에 의해 구해짐
- 모든 정렬 알고리즘은 Ω(N). N개의 데이터를 정렬하는데 N개 모두를 읽지 않고 정렬을 완료할 수는 없기 때문.

❑ 최선의 알고리즘

- 어떤 알고리즘이 $O(N^2)$임을 증명. “길어야 이만한 시간이면 된다”
- 그 문제는 Ω(N)임을 증명. “최소한 이만한 시간은 걸린다”
- 우리가 모르는, 효율이 N에서 N^2 사이인 더 나은 알고리즘 존재가능
- 알고리즘이 개발됨에 따라 사이는 점차 좁혀짐

❑ 빅 쎄타 Θ(N) (Big Theta)

- Ω(N)인 문제에 O(N)인 알고리즘이 존재
- 상한과 하한이 마주 침으로써 더 이상 나은 알고리즘은 존재하지 않음.

리스트 함수의 효율

	배열로 구현	연결 리스트로 구현
1) Find List Length	O(1) / O(N)	O(1) / O(N)
2) Insert an Item	O(1) / O(N)	O(1)
3) Insert an Item at Ith Position	O(N)	O(N)

- 1) 배열내 레코드 수를 추적하는 변수가 별도로 있으면 O(1)
- 2) 정렬 안된 배열과 정렬된 배열에 따라 달라짐
- 3) 사용자가 위치를 직접 지정하여 삽입

선형탐색

- 코드 8-6: 선형탐색 I

```
int SequentialSearch(recordType A[ ], int SearchKey, int
  N)
{  for (int i = 0; i < N; i++)
     if (A[i].Key = = SearchKey) 찾는 키가  일치하면
       return A[i];                해당 요소를 리턴
  return(-1);                      찾는 레코드가 없으면 -1
  을 리턴
}
```

- 효율
  - 최악의 경우 배열 끝까지 비교
  - O(N)

선형탐색: 센티넬 사용

코드 8-7: 선형탐색 II

```
int SequentialSearch(recordType A[ ], int SearchKey, int N)
{  A[N].Key = K;
   for (int i = 0; A[i].Key != SearchKey; i++)
   if (i < N)                    배열 내부에 찾는 레코드가 있음
     return A[i];          해당 요소 구조체를 리턴
   else                  그렇지 않으면
     return(-1)            찾는 레코드가 없음을 -1로 표시하여 리턴
}
```

효율

- 동일함. O(N)
- 경우에 따라서는 계수를 줄이려는 노력

이진탐색

❑ 비교회수

- 비교 한번에 데이터 크기가 64, 32, 16, 8, 4, ..., 1
- 총 비교회수는 대략 lgN 번

❑ 지수함수 알고리즘

- 10N + 10, 이진탐색 1000lgN + 1000

N	선형탐색	이진탐색
10	110	4,322
100	1,010	7,644
1,000	10,010	10,966
10,000	100,010	14,288
100,000	1,000,010	17,610
1,000,000	10,000,010	20,932

분할상각 복잡도

❑ 아모타이제이션

- 연속된 작업의 전체적인 효율
- 모든 작업이 서로 연관되어 있다고 전제
- A가 느려지면 B가 빠를 수도 있고 역으로 A가 빨라지면 B가 느려질 수 있음.

❑ 일반적 분석방법

- A, B가 완전히 독립된 작업으로서 그 중에 복잡도가 큰 작업만을 기준으로 전체적인 복잡도를 평가

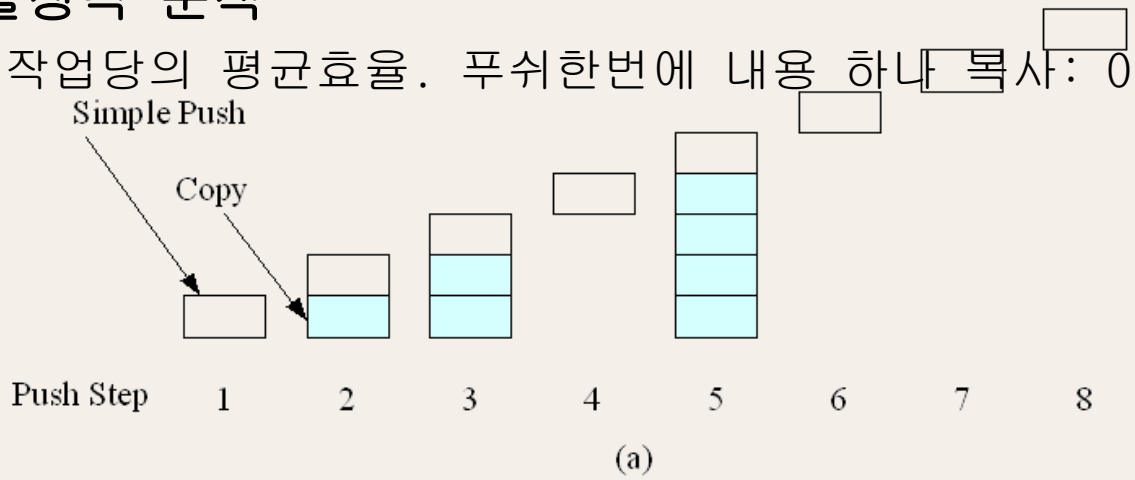
분할상각 복잡도

- 코드 8-8: 동적 배열

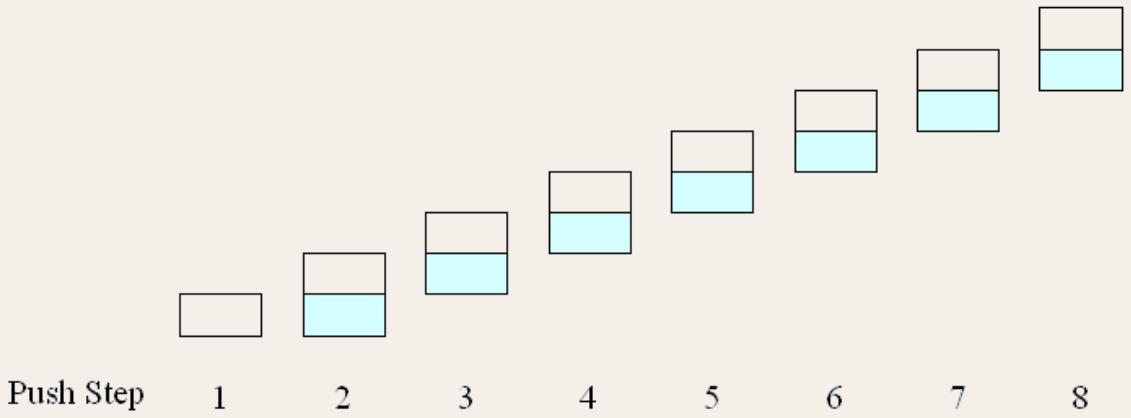
```
push(Item)
{ if (StackIsFull)
  {   new DoubleArray[2 * StackSize];
     Copy Old Stack Data into DoubleArray;
     Push Item into DoubleArray;
     Increase StackSize;
  }
}
```

분할상각 복잡도

- **일반적 분석**
  - 최악의 경우 푸쉬 한번에 새로운 배열 만들고 모든 내용 복사: O(N)
- **분할상각 분석**
  - 작업당의 평균효율. 푸쉬한번에 내용 하나 복사: O(1)



(a)



(b)