알고리즘

김 용 석

2020년 가을

강원대학교 컴퓨터공학과

Chapter 1

Algorithms: Efficiency, Analysis, and Order

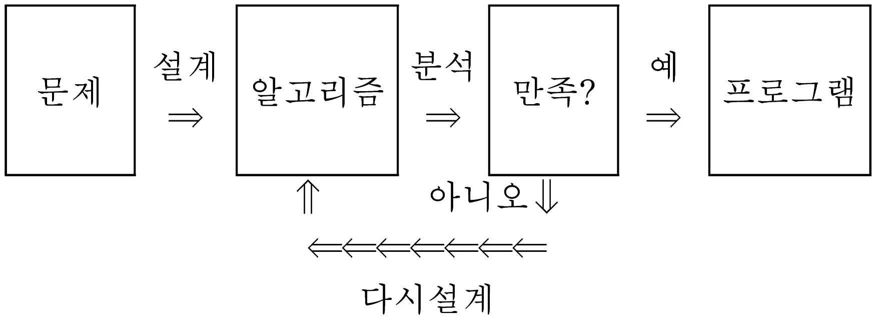
알고리즘들 간에 효율성을 비교평가하기 위해 알고리즘의 복잡도를 분석

⇨ Order로 표현

소프트웨어 개발 과정

요구분석 ⇒ 설계 ⇒ 구현 ⇒ 시험 ⇒ 안정화

프로그램의 설계과정



이 과목은

- 프로그래밍 언어가 아니라
  문제를 해결하는 방법에 대한 것
- 프로그램을 전체가 아니라
  성능에 중요한 영향을 미치는 모듈에 대한 것

Algorithm

- 알고리즘이란
  - 문제에 대한 답을 찾기 위한 계산 절차
  - 단계별로 주의 깊게 설계된 계산과정
  - 입력을 받아서 출력을 내는 일련의 계산절차
- 예시

- Problem: 전화번호부에서 “홍길동”의 전화번호 찾기
- Algorithm1: 첫 쪽부터 차례대로 검사
- Algorithm2: “ㅎ”이 있을 만한 곳에서 앞뒤로 검사
  <== 전화번호부는 가나다순으로 되어있는 점을 활용
- Algorithm1: Sequential Search
- Algorithm2: Modified Binary Search
- Analysis: 어느 알고리즘에 우수한가?
- Correctness: 무순이라면 어느 알고리즘이 맞나?

문제의 표기

● 문제 표기에 필요한 사항
- **Problem**: 답을 찾기 위해 제공된 질문
- **Parameter**: 문제에서 특정한 값이 지정되지 않은 변수
- **Instance** (**Input**): Parameter에 특정 값 지정
- **Solution** (**Output**): Instance에 대한 문제의 답

● 보기
- **Problem**: n개의 수를 가진 리스트 S에 x라는 수가 있는지를 결정하시오. 답은 S에 x가 있으면 "예", 없으면 "아니오"
- **Parameter**: S, n, x
- **Input**: S = [10, 7, 11, 5, 13, 8], n=6, x=5 ⇐ Instance
- **Output**: "예"

알고리즘의 표기

● 자연어 (한국어, 영어, ...)

● 프로그래밍 언어: C, C++, Java, Python, Pascal, ...

● Pseudo-code:
- 표현하기 쉬우면서도, 계산 과정을 실제 프로그램에 가깝게 표현 할 수 있는 언어
- 일반적으로 Pseudo-code를 많이 사용
- 교재는 C++와 유사한 형식의 Pseudo-code 사용

알고리즘 예: Sequential Search

● Problem: 크기가 n인 배열 S에 x가 있는가?

● Parameter: 양수 n, 배열 S[1..n], 찾는 항목 x

● Output: x의 위치, 없으면 0

● 자연어 알고리즘: x와 일치하는 항목을 찾을 때까지 S에 있는 모든 항목을 처음부터 차례대로 비교한다. 만일 x와 일치하는 항목을 찾으면 그 위치를 출력한다. 만약 S의 모든 항목을 검사하고도 찾지 못하면 0을 출력한다.

● Pseudo-code 알고리즘: (알고리즘 1.1)

```
void seqsearch(
   int n, const keytype S[], keytype x,        // input
   index& location)                            // output
{
     location = 1;
     while (location <= n && S[location] != x)
          location++;
     if (location > n)
          location = 0;
}
```

● 관찰사항: x를 찾기 위해 몇 개나 검사해야 하나?

⇒ 최선의 경우 1번, 최악의 경우 n번

● 좀 더 빠른 알고리즘은?

그러나 S가 오름차순으로 정렬되어 있다면?

Binary Search Algorithm

```
void binsearch(                    // 알고리즘 1.5
   int n, const keytype S[], keytype x,    // input
   index& location)                        // output
{
      index low, mid, high;

      low = 1; high = n; location = 0;
      while (low <= high && location == 0) {
            mid = ⌊(low + high) / 2⌋;
            if (x == S[mid])
                  location = mid;
            else if (x < S[mid])
                  high = mid - 1;
            else
                  low = mid + 1;
      }
}     // divide-and-conquer
```

- 관찰사항: x를 찾기 위해 몇 개나 검사해야 하나?
  ⇒ 최선의 경우 1번, 최악의 경우 $\log_2 n + 1$ 번
- 검사회수 비교 (표 1.1)

n	seqsearch	binsearch
128	128	8
1,024	1,024	11
1,048,576	1,048,576	21
4,294,967,296	4,294,967,296	33

n-th Fibonacci Term

- Definition

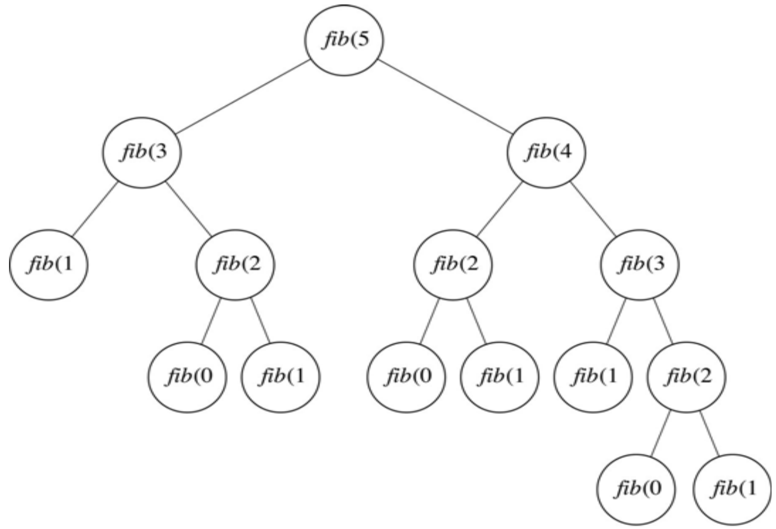
- $f_0 = 0,\ f_1 = 1,$

- $f_n = f_{n-1} + f_{n-2}$ for $n \geqq 2$

- `int fib(int n)            // 알고리즘 1.6`

```
{
      if (n <= 1)
            return n;
      else
            return fib(n-1) + fib(n-2);
}     // divide-and-conquer
```

- fib의 Recursion Tree



● 관찰사항

- fib 에서는 동일한 항목을 중복하여 계산

⇒ 속도 개선을 위해서 필요한 항목을 미리 계산하고 그 결과를 활용

● *int* **fib2**(*int* n) // 알고리즘 1.7

```
{
      index i; int f[0..n];

      f[0] = 0;
      if (n > 0) {
            f[1] = 1;
            for (i=2; i <= n; i++)
                  f[i] = f[i-1] + f[i-2];
      }
      return f[n];
}     // dynamic programming
```

● Number of Computed Terms for T(n)

- **fib2**: T(n) = n+1
- **fib**: Since T(k) > T(k-1),

$$T(n) > T(n-1) + T(n-2) > 2 \times T(n-2)$$
$$> 2 \times 2 \times T(n-4)$$
$$> \ldots\ldots > 2^{n/2} \times T(0) = 2^{n/2}$$

⇨ $T(n) \in \Omega(2^{n/2})$

● 실행 시간 비교: Table 1.2

- Computation speed : assume 1 ns / term

n	Alg. 1.7 (fib2)	Alg. 1.6 (fib)
60	61 ns	1 s
120	121 ns	36 years
200	201 ns	4×10^{13} years

Performance Evaluation

- 측정(사후) : performance measurement
- 예측(사전) : performance (complexity) analysis

Complexity Analysis

- Time Complexity : 계산 시간
  - 입력의 크기에 따른 단위연산의 실행 회수
  - 부가적인 오버헤드는 무시할 수 있다고 가정

o 단위연산: comparison, assignment, addition 등

o 입력의 크기: 배열의 크기, 리스트의 길이, 행렬의 행과 열의 크기, 트리의 node 수와 edge 수 등

- Space (Memory) Complexity : 메모리 용량
- 분석 방법의 종류
  - Every-Case Complexity: $T(n)$
  - Worst-Case Complexity: $T_{worst}(n)$ 또는 $W(n)$
  - Best-Case Complexity: $T_{best}(n)$ 또는 $B(n)$
  - Average-Case Complexity: $T_{average}(n)$ 또는 $A(n)$

Time Complexity 예

- 배열의 모든 수 더하기 (Alg. 1.2)

```
number sum(int n, const number S[]) {
    index i;   number result;
    result = 0;
    for (i=1; i <= n; i++)
        result = result + S[i];
    return result;
}
```

- Time Complexity

- 단위 연산: result = result + S[i]

  $T(n) = n$

(참고)

- *단위연산은 전체 실행시간에 영향을 많이 미치는 부분으로 선택*
- *단순히 정수를 이용한 반복회수 처리는 단위연산에서 제외 (전체 시간에서 중요하지 않을 때)*
- *i++ 는 number의 종류와 무관*
- *result = result +S[i]는 number의 종류에 따라 다름*
- *차수가 중요하고 곱하는 상수는 덜 중요함*

Time Complexity 예

● Exchange Sort (non-decreasing order) (Alg. 1.3)

```
void exchangesort(int n, keytype S[]) {
    index i, j;
    for (i=1; i < n; i++)
        for (j=i+1; j <= n; j++)
            if (S[j] < S[i])
                exchange S[i] and S[j];
}
```

● 단위연산이 if 문의 비교일 때

$T(n) = (n-1) + (n-2) + \dots + 1 = n(n-1)/2$

● 단위연산이 exchange 연산일 때

Best-case: $B(n) = 0$

Worst-case: $W(n) = n(n-1)/2$

Every-case: $T(n) =$???

Average-case (확률 반영): $A(n) =$???

⇨ $A(n) \le n(n-1)/2$ ==> $A(n) \in O(n^2)$

● 관찰사항

- 목적에 따라 적절한 것 사용
- average-case 는 분석이 어려움
- 일반적으로 worst-case와 average-case는 같은 카테고리에 속함

Correctness Analysis

● 정확성 분석

- 알고리즘이 예상한 대로 실행되는지의 증명

● 정확한 알고리즘이란

- 어떤 입력에도 맞는 답을 출력하고 종료

- 본 과목에서는 Time Complexity Analysis에 중점
- Greedy 알고리즘은 정확성 분석이 필수

몇 가지 수학 기초

● 지수와 로그

- $a^0 = 1$
- $n^a \times n^b = n^{a+b}$
- $(n^a)^b = n^{ab} = n^{ba} = (n^b)^a$
- $a^k = n$ 이면 $k = \log_a n$
- $\log_a b^k = k \log_a b$
- $\log_a uv = \log_a u + \log_a v$

  $u = a^x$,$v = a^y$ 이면 $uv = a^{x+y}$

  $\log_a uv = \log_a a^{x+y} = x + y = \log_a u + \log_a v$
- $2^{10} = 1024 \approx 1000 = 10^3$

  2^{32} 는 십진수로 대략 얼마 ???

● 등차수열의 합

(1) $1 + 2 + \dots + n = S$

순서를 거꾸로 하면

(2) $n + \dots + 2 + 1 = S$

(1)과 (2)의 양변을 각각 더하면 $(n+1) \times n = 2S$

따라서 $S = n(n+1)/2$

● 등비수열의 합

(1) $1 + r^1 + r^2 + r^3 \dots + r^{n-1} = S$

양변에 r을 곱하면

(2) $r^1 + r^2 + r^3 \dots + r^{n-1} + r^n = rS$

(1)에서 (2)의 양변을 각각 빼면 $1-r^n = (1-r)S$

따라서 $S = (1-r^n)/(1-r)$

Order

● 알고리즘의 Complexity를 간단하게 표시하는 수단

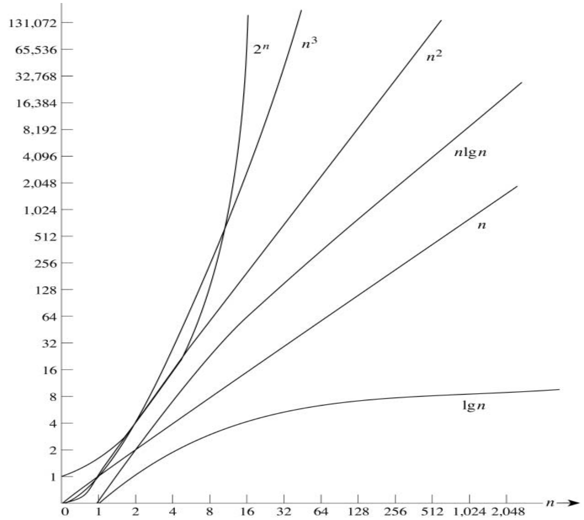
- Time complexity가 $n(n-1)/2$ 이면 $\Rightarrow \Theta(n^2)$

● 예: $\Theta(n^2)$

- 2차 함수에 해당하는 모든 Complexity 함수의 집합
- n^2, $9n^2+1000$, $2n^2 + 3n \log n$, $2n^2 + n^{1/2}$, ...

● 제일 높은 order가 전체 소요 시간을 좌우

⇨ 낮은 차수는 무시해도 무방함 (그림 1.3)



Algorithm의 Complexity 표기

● 실행 시간의 상/하한선 표시
- 예를 들어 실행시간이 60이라면 그냥 60으로 표시
- 정확히는 알지 못하지만 50~100인 것은 분명할 때 하한선이 50, 상한선이 100 이라고 표시

● Asymptotic Bound
- input size n 인 problem instance에 대해
- time complexity가 5n 이라면 $\Theta(5n)$ 으로 표시; 더 간단하게는 $\Theta(n)$ 으로 표시
- time complexity가 정확히는 모르지만 2n 이상 $3n^2$ 이하인 것이 확실하다면 $\Omega(2n)$, $O(3n^2)$로 표시; 더 간단하게는 $\Omega(n)$, $O(n^2)$로 표시

● Asymptotic Bound 표기: Big O, Ω, 및 Θ
- 알고리즘의 정확한 Order를 알 수 있으면 Θ로 표기
- 그렇지 않으면 상한선 O와 하한선 Ω로 표기

Asymptotic Bound

● 정확하게는 **집합으로 정의**함

● Upper Bound **O(f(n))** : 최고차항의 차수가 f(n)보다 <u>크지 않은</u> 모든 함수들의 집합

예: $O(n^2) = \{ n^2, 5n^2 + 20n, 5n + 100, n \log n, \ldots \}$

즉, $5n^2 + 20n \in O(n^2)$, $5n + 100 \in O(n^2)$, …

● Lower Bound **Ω(f(n))** : 최고차항의 차수가 f(n)보다 <u>작지 않은</u> 모든 함수들의 집합

예: $\Omega(n^2) = \{ n^2, 5n^3+20, n^2\log n, \ldots \}$

즉, $n^2 \in \Omega(n^2)$, $5n^3+20 \in \Omega(n^2)$, $n^2\log n \in \Omega(n^2)$, …

● Order **Θ(f(n))** : 최고차항의 차수가 f(n)과 <u>일치하는</u> 모든 함수들의 집합

⇨ $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

예: $\Theta(n^2) = \{ n^2, 5n^2 + 20n, n^2/3 - 5n + 100, \ldots \}$

즉, $n^2 \in \Theta(n^2)$, $5n^2 + 20n \in \Theta(n^2)$, …

● 상수 부분은 무시함 (가장 간단한 표기가 좋음)
- $O(3n^2) = O(n^2+2n+100) = O(n^2)$
- $\Omega(2n) = \Omega(10n+2) = \Omega(n)$
- $\Theta(2n) = \Theta(10n+2) = \Theta(n)$

⇦ 같은 알고리즘도 프로그래밍 언어, 하드웨어 성능 등에 따라 실행속도에 차이가 있으므로 단순히 몇 배의 의미는 무시함

Asymptotic Bound Examples

- Example

ex1) x = x+y;

⇨ $T(n) \in \Theta(1)$, $T(n) \in O(1)$, $T(n) \in \Omega(1)$

ex2) for (i=0; i < n; i++)

x = x + i;

⇨ $T(n) \in \Theta(n)$, $T(n) \in O(n)$, $T(n) \in \Omega(n)$

ex3) for (i=0; i < n; i++)

for (j=0; j < n; j++)

x = x + y;

⇨ $T(n) \in \Theta(n^2)$, $T(n) \in O(n^2)$, $T(n) \in \Omega(n^2)$

- ExchangeSort 알고리즘에서 exchange 회수

$T(n) \leq n(n-1)/2$

⇨ $T(n) \in O(n^2)$, $T(n) \notin \Theta(n^2)$, $T(n) \notin \Omega(n^2)$

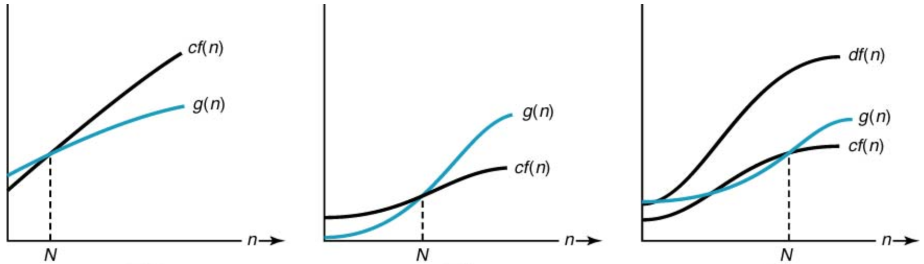
- fib 알고리즘에서 계산 회수

$T(n) > 2^{n/2}$

⇨ $T(n) \in \Omega(2^{n/2})$, $T(n) \notin \Theta(2^{n/2})$, $T(n) \notin O(2^{n/2})$

Big O, Ω, 및 Θ의 정확한 정의

- 그림 1.4



(a) $g(n) \in O(f(n))$ (b) $g(n) \in \Omega(f(n))$ (c) $g(n) \in \Theta(f(n))$

- Example 1.7: $T(n) = 5n^2$

  $T(n) \in \mathbf{O}(n^2)$ since $5n^2 \leq 5 \times n^2$ for $n \geq 0$

  and we can take c = 5 and N = 0

  추가: $T(n) \in \mathbf{\Omega}(n^2)$ since $5n^2 \geq 1 \times n^2$ for $n \geq 0$

  따라서 $T(n) \in \mathbf{\Theta}(n^2)$

- 추가: $5n^2 \in \mathbf{O}(10n^2)$ since $5n^2 \leq 1 \times 10n^2$ for $n \geq 0$
- 추가: $T(n) \leq n^2 + 10n$ 이면

  $T(n) \in \mathbf{O}(n^2)$

  since for $n \geq 1$, $T(n) \leq n^2+10n \leq 11 \times n^2$
- Example 1.11:

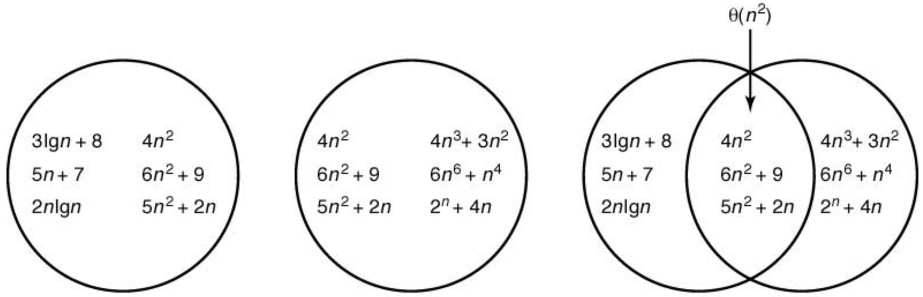
  $n \in \mathbf{O}(n^2)$ since $n \leq 1 \times n^2$ for $n \geq 0$

- More Examples of Big O
  - $10n^2 + 4n + 2 \in O(n^2)$
  - $6*2^n + 10n \log n + 2 \in O(2^n)$
  - $3n + 2 \notin O(1)$
  - $10n^2 + 4n + 2 \notin O(n)$
- More Examples of Ω
  - $10n^2 + 4n + 2 \in \Omega(n^2)$
  - $6*2^n + 10n \log n + 2 \in \Omega(2^n)$
  - $100n + 8 \notin \Omega(n^2)$
  - $3n + 2 \in \Omega(1)$
  - $10n^2 + 4n + 2 \in \Omega(n)$

Order of a Function (Theta)

- $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

- Set O, Ω, and Θ (그림 1.6)



(a) $O(n^2)$ (b) $\Omega(n^2)$ (c) $\theta(n^2) = O(n^2) \cap \Omega(n^2)$

- 표기의 편의상 가장 단순한 형태를 사용
  (예) $4n^2$, $6n^2+9$, $5n^2+2n$, ... ⇨ $\Theta(n^2)$

- (참고) 실용적인 관점에서
  - P와 Q의 시간복잡도가 각각 $\Theta(n)$ 와 $\Theta(n^2)$ 이라면 n 이 충분히 크면 P가 Q보다 빠르다고 한다.
  - 지수함수의 시간복잡도인 프로그램은 n 이 작을 때만 사용 가능하다 (보통 40이하)
- 지수함수의 예
  - 200번째 Fibonacci number: 4 x 10^{13} years
  - 0.1mm 두께의 종이를 100번 반으로 접으면 두께는?
  - 64개의 링을 가진 Hanoi Tower ?

Example of Worst/Best Case

● Exchange Sort (non-decreasing order) (Alg. 1.3)

```
void exchangesort(int n, keytype S[]) {
  index i, j;
  for (i=1; i < n; i++)
    for (j=i+1; j <= n; j++)
      if (S[j] < S[i])
        exchange S[i] and S[j];
}
```

● 단위연산 exchange 회수

- input 이 n=4, S={3,5,6,8} 일 때 0회 <-- best
- input 이 n=4, S={2,8,3,7} 일 때 2회
- input 이 n=4, S={9,6,3,2} 일 때 6회 <-- worst

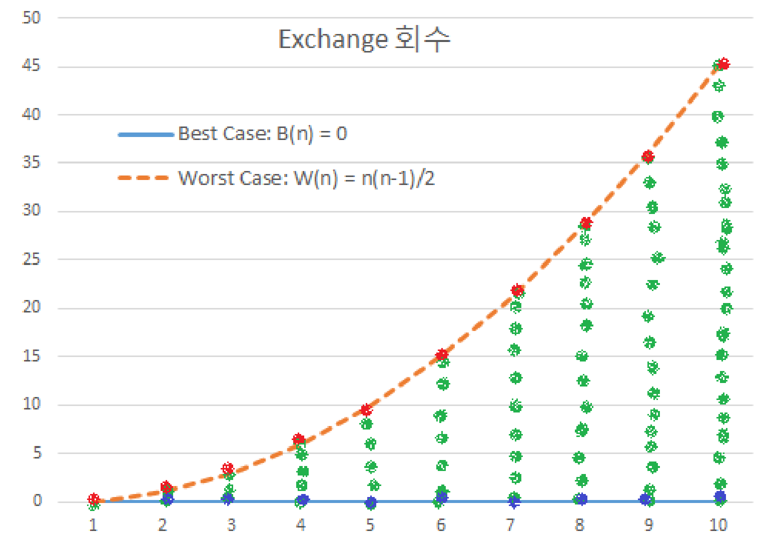
...

- input 이 n=5, S={3,5,6,8,9} 일 때 0회 <-- best
- input 이 n=5, S={2,8,3,7,9} 일 때 2회
- input 이 n=5, S={9,8,6,3,2} 일 때 10회 <-- worst

...

● Exchange 회수

n	Best-case	Worst-case
1	0	0
2	0	1
3	0	3
4	0	6
5	0	10
…	…	…
n	0	n(n-1)/2



$W(n) \in \Theta(n^2)$: 빨간 점들 대상

$B(n) = 0$: 파란 점들 대상

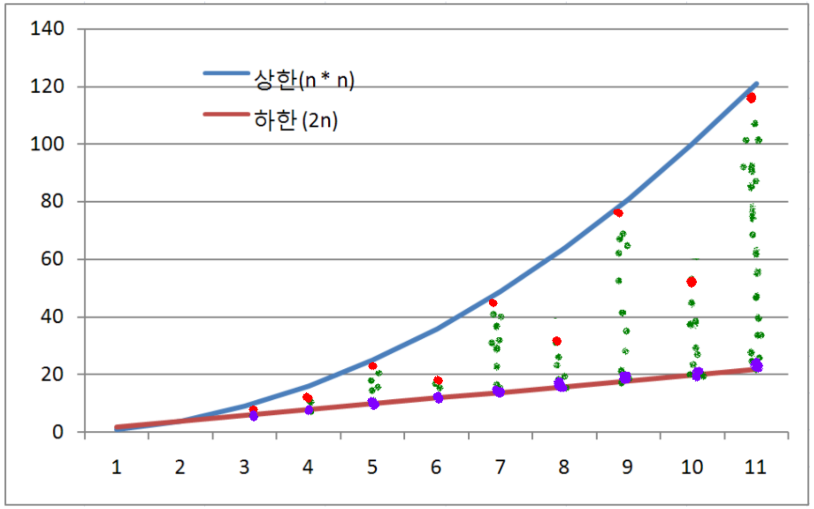
$T(n) \in O(n^2)$: 모든 점들 대상

Average Case:

만약 0과 n(n-1)/2 사이에 고루 분포한다면

$A(n) = n(n-1)/4 \in \Theta(n^2)$, 즉 $A(n) \in \Theta(n^2)$

T(n), W(n), B(n), A(n)



- W(n)은 빨간 점들만, B(n)은 파란 점들만, T(n)은 모든 점들이 대상임
- $T(n) < n^2$ ⇨ $T(n) \in O(n^2)$, $T(n) \in \Theta(n^2)$??

  $T(n) \geq 2n$ ⇨ $T(n) \in \boldsymbol{\Omega}(n)$, $T(n) \in \Theta(n)$??

  $W(n) < n^2$ ⇨ $W(n) \in O(n^2)$, $W(n) \in \Theta(n^2)$??

  $B(n) = 2n$ ⇨ $B(n) \in \Theta(n)$

  $A(n) =$??

만약 $n^2/2 \leq W(n) < n^2$ 이면: $W(n) \in \Theta(n^2)$

간단한 분석과 정확한 분석

● Exchange Sort 에서 단위연산 exchange 에 대하여

```
for (i=1; i < n; i++)
    for (j=i+1; j <= n; j++)
        if (S[j] < S[i])
            exchange S[i] and S[j];
```

● Worst-case 의 간단한 분석

- 이중 for loop 은 각각 n 회 이하이므로

  $W(n) \leq n \cdot n$,

  따라서 $W(n) \in O(n^2)$ --- (1)

⇨ *자세하게 분석하지 않고도 얻을 수 있음*

● Worst-case 의 정확한 분석

- $W(n) = (n-1) + (n-2) + \ldots + 1 = n(n-1)/2$

  따라서 $W(n) \in \Theta(n^2)$ --- (2)

(참고) *복잡한 알고리즘은 정확한 분석이 매우 어려움*

● Average-case 의 간단한 분석

- (1) 의 결과로부터 $A(n) \in O(n^2)$
- (2) 의 결과로부터 $A(n) \in O(n^2)$

● Average-case 의 정확한 분석

- 확률을 적용하여 처리: $A(n) \in \Theta($???$)$

● T(n) 에 대하여

- (1) 의 결과로부터 $T(n) \in O(n^2)$
- (2) 의 결과로부터 $T(n) \in O(n^2)$