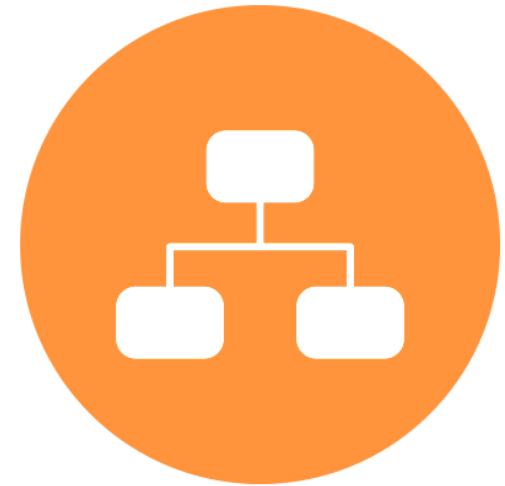


Data Structure

1st Study: Basic Concepts

- Course Introduction
- Performance Analysis and Measurement



C++ Korea 옥찬호 (utilForever@gmail.com)

Course Introduction

Course Overview

- 교과서
 - Horowitz, Sahni, Mehta, “Fundamentals of Data Structures in C++”
- 평가 방법
 - 과제 (매 주 출제) : 30%
 - 프로그래밍 프로젝트 : 30%
 - 중간 시험 : 20%
 - 기말 시험 : 20%
- 보충 교과서
 - Goodrich, Tamassia, “Data Structures and Algorithms in C++”
 - Cormen, Leiserson, “Introduction to Algorithms, 3rd Edition”
 - Heineman, Pollice, “Algorithms in a Nutshell”

Prerequisites

- C++ 프로그래밍
 - 명령문, 연산자, 데이터 선언, 주석문, 입출력
 - 함수, 매개변수 전달, 함수 오버로딩, 인라인 함수, 동적 메모리 할당
 - 객체 지향 설계, 데이터 추상화와 캡슐화
 - 클래스, 객체, 상속, 다형성, 템플릿, 예외 처리
 - STL (Standard Library)

Schedules

- 07/25 : 기본 개념 (과목 소개, 성능 분석과 측정)
- 08/01 : 배열 (다항식, 희소 행렬, KMP 알고리즘)
- 08/08 : 스택과 큐 (미로 문제, 수식의 계산)
- 08/15 : 연결 리스트 (원형 리스트, 가용 공간 리스트,
연결 스택과 큐, 다항식, 동치 관계, 희소 행렬)
- 08/22 : 트리 1 (이진 트리, 이진 트리 순회, 스레드 이진 트리)
- 08/29 : 트리 2 (힙, 이원 탐색 트리, 선택 트리, 포레스트)

Schedules

- 09/05 : 그래프 1 (DFS, BFS, MST, Kruskal, Prim, Sollin)
- 09/12 : 그래프 2 (Dijkstra, Bellman-Ford, AOV, AOE)
- 09/19 : 정렬 1 (삽입 정렬, 퀵 정렬, 합병 정렬, 힙 정렬)
- 09/26 : 추석 (중간 시험)
- 10/03 : 정렬 2 (키 정렬, 리스트 정렬, 테이블 정렬, 외부 정렬)
- 10/10 : 해싱 (정적 해싱, 동적 해싱, 블룸 필터)
- 10/17 : 우선순위 큐 1 (좌향 트리, 이항 힙, 피보나치 힙)

Schedules

- 10/24 : 우선순위 큐 2 (페어링 힙, SMMH 힙, 구간 힙)
- 10/31 : 효율적 이원 탐색 트리 (AVL, Red-Black, Splay)
- 11/07 : 다원 탐색 트리 (m원 탐색, B, B+)
- 11/14 : 디지털 탐색 구조 (Trie, m원 Trie, 접미 트리)
- 11/21 : 기말 시험 및 종강

Performance Analysis and Measurement

Program Complexity

- 프로그램 복잡도에는 크게 두 종류가 있음
 - 공간 복잡도(Space Complexity) : 프로그램을 수행하는데 필요한 메모리량
 - 시간 복잡도(Time Complexity) : 프로그램을 수행하는데 필요한 시간
- 성능 평가 단계
 - 성능 분석(Performance Analysis) : 사전 예측
 - 성능 측정(Performance Measurement) : 사후 검사

Space Complexity

- 고정 부분 : 프로그램의 입출력 특성과 관계 없음
 - 명령어 공간, 단순 변수, 일정 크기의 변수, 상수들을 위한 공간 등
- 가변 부분 : 문제의 인스턴스를 통해 크기가 결정됨
 - 인스턴스를 통해 크기가 결정되는 변수, 재귀 함수를 위한 공간 등
- $S(P) = c + S_P(n)$
 - $S(P)$: 프로그램 P 에서 필요한 공간
 - c : 상수
 - n : 인스턴스 특성 (예 : 입출력 크기, 숫자 등)

Space Complexity : ABC Function

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

- 함수 ABC에 필요한 공간은?
 - a , b , c , 그리고 반환값이 필요
 - 따라서, ABC에 필요한 공간은 4 워드

Space Complexity : Sum Function

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

- 함수 Sum에 필요한 공간은?
 - n 이 값에 의한 전달로 복사됨 : 1 워드
 - a 는 $a[0]$ 의 주소 : 1 워드
 - 그러므로 함수에 필요한 공간은 n 에 무관함 : $S_P(n) = 0$
 - s, i , 반환값이 필요 \rightarrow Sum에 필요한 공간은 5 워드

Space Complexity : Rsum Function

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

- 함수 Rsum에 필요한 공간은?
 - n 이 입력되었을 때, 재귀 함수의 깊이는 $n + 1$ ($0 \sim n$)
 - 함수를 호출할 때마다 n , a , 반환값, 반환 주소가 필요함
→ 4개의 워드 필요
 - 따라서, Rsum에 필요한 공간은 $4(n + 1)$ 워드

Time Complexity

- $T(P) = c + T_p(n)$
 - $T(P)$: 프로그램 P 를 수행하는데 걸리는 시간
 - c : 컴파일 시간
 - T_p : 실행 시간
 - n : 인스턴스 특성

Program Steps

- 프로그램 단계는 정확하지는 않지만 인스턴스 특성에 독립적인 실행 시간을 갖는 프로그램의 세그먼트(Segment)
- 실행 시간은 인스턴스 특성과 무관함
- 프로그램 명령문의 단계 수는 명령문의 특성에 의존함

Number of Steps

- 주석 : 0 (비실행 명령문)
- 선언문 : 0
 - 변수나 상수를 정의하는 모든 명령문(int, long, short, char, float, double, const, enum, signed, unsigned, static, extern)
 - 사용자 정의 데이터 타입을 정의하는 모든 명령문(class, struct, union, template)
 - 접근을 결정하는 모든 명령문(private, public, protected, friend)
 - 함수의 타입을 결정하는 모든 명령문(void, virtual)
- 산술식 및 지정문
 - 대부분의 산술식은 1, 다만 함수 호출을 포함하는 산술식은 예외 (잠시 후에 예제로 살펴볼 예정)
 - 지정문 `<variable> = <expr>`은 `<variable>`의 크기가 인스턴스 특성의 함수가 아니라면 `<expr>`의 단계 수와 같음
 - 하지만 인스턴스 특성의 함수라면, `<expr>`의 단계 수에 `<variable>`의 크기를 더한 값이 됨
- 함수 호출
 - 호출이 인스턴스 특성에 의존하는 인자를 포함하지 않으면 1
 - 인자를 포함한다면, 단계 수는 인자 크기의 합이 됨
 - 호출되는 함수가 재귀 함수라면 호출되는 함수에서 지역 변수도 고려해야 함
인스턴스에 관련된 지역 변수의 크기는 단계 수에 합쳐짐

Number of Steps

- 반복문

- for (<init-stmt>; <expr1>; <expr2>
 - <init_stmt>, <expr1>, 또는 <expr2>가 인스턴스 특성의 함수가 아니라면 1
 - 인스턴스 특성의 함수라면, for 명령문의 첫 번째 실행은 <init-stmt>와 <expr1>의 단계 수의 합과 같음
그 다음 실행 단계 수는 <expr1>과 <expr2>의 단계 수의 합과 같음
- while <expr> do : <expr>에 할당된 단계 수와 같음
- do ... while <expr> : <expr>에 할당된 단계 수와 같음

- switch 문

- switch (<expr>) {
 - case cond1: <statement1>
 - case cond2: <statement2>
 - ...
 - default: <statement>
- }
- switch(<expr>)의 비용은 <expr>에 할당된 비용과 같음
- 각 조건의 비용은 자기의 비용 + 앞에서 나온 모든 조건의 비용

Number of Steps

- if-else 문
 - if (<expr>) <statement1>;
else <statement2>;
 - <expr>, <statement1>, <statement2>에 따라 각 단계 수가 할당됨
 - 만약 else문이 없다면 거기에 해당되는 비용도 없음
- 메모리 관리 명령문
 - new object, delete, sizeof()를 포함
 - 각 명령문은 1
 - 묵시적으로 new와 delete가 호출되는 경우 함수 호출과 같은 방법으로 계산됨
- 함수 명령문 : 0 (비용이 이미 호출문에 할당됨)
- 분기 명령문
 - continue, break, goto, return, return <expr>을 포함
 - return <expr>을 제외하면 모두 1
 - return <expr>에서도 <expr>의 단계 수가 인스턴스 특성의 함수가 아니라면 1
인스턴스 특성의 함수라면 <expr>의 비용과 같음

Program Steps : Sum Function

```
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 }
```

s/e : 실행당 단계 수 (Step per Execution)			
행 번호	s/e	빈도	단계 수
1	0	0	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			$2n + 3$

Program Steps : Rsum Function

```

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 }

```

		s/e : 실행당 단계 수 (Step per Execution)			
행 번호	s/e	빈도		단계 수	
		$n = 0$	$n > 0$	$n = 0$	$n > 0$
1	0	1	1	0	0
2(a)	1	1	1	1	1
2(b)	1	1	0	1	0
3	$1 + t_{Rsum}(n - 1)$	0	1	0	$1 + t_{Rsum}(n - 1)$
4	0	1	1	0	0
총 단계 수				2	$2 + t_{Rsum}(n - 1)$

Program Steps : Rsum Function

- $$\begin{aligned} t_{\text{rsum}}(n) &= 2 + t_{\text{rsum}}(n - 1) \\ &= 2 + 2 + t_{\text{rsum}}(n - 2) \\ &= 2 * 2 + t_{\text{rsum}}(n - 2) \\ &= \dots \\ &= 2n + t_{\text{rsum}}(0) \\ &= 2n + 2 \end{aligned}$$

Program Steps : Add Function

```
void Add(int** a, int** b, int** c, int m, int n)
1 {
2     for (int i = 0; i < m; i++)
3         for (int j = 0; j < n; j++)
4             c[i][j] = a[i][j] + b[i][j];
5 }
```

s/e : 실행당 단계 수 (Step per Execution)			
행 번호	s/e	빈도	단계 수
1	0	0	0
2	1	$m + 1$	$m + 1$
3	1	$m(n + 1)$	$mn + m$
4	1	mn	mn
5	0	1	0
총 단계 수			$2mn + 2m + 1$

Three Kinds of Step Counts

- 최상의 경우(Best Case) : 주어진 매개변수에 대해 실행될 수 있는 단계 수가 최소인 경우
- 최악의 경우(Worst Case) : 주어진 매개변수에 대해 실행될 수 있는 단계 수가 최대인 경우
- 평균(Average Case) : 주어진 매개변수에 대해 인스턴스가 실행되는 평균 단계 수

Asymptotic Notation (O , Ω , Θ)

- 프로그램의 정확한 단계 수를 결정하는 작업은 매우 어려움
 - 단계 수를 정확하게 결정하는데 드는 노력은
개념 자체가 부정확하기 때문에 낭비를 초래할 수 있음
- 점근 표기법은 어떤 함수의 증가 양상을 다른 함수와
비교로 표현하는 수론과 해석학의 방법
 - 알고리즘의 복잡도를 단순화할 때 사용
 - 대문자 O 표기법, 소문자 o 표기법
 - 대문자 Ω 표기법, 소문자 ω 표기법
 - 대문자 Θ 표기법

Definition of Big "oh" (O)

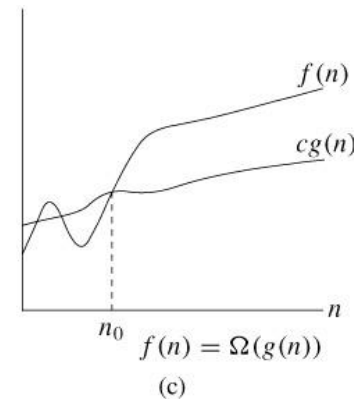
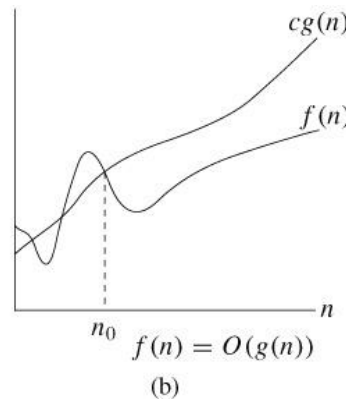
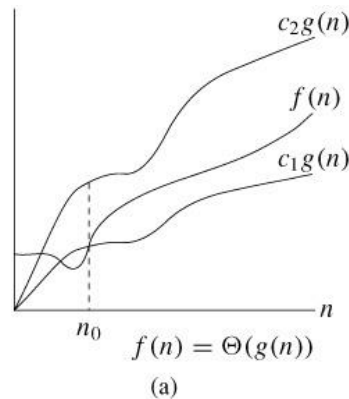
- 모든 n , $n \geq n_0$ 에 대해 $f(n) \leq cg(n)$ 인 조건을 만족시키는 두 양의 상수 c 와 n_0 가 존재하기만 하면 $f(n) = O(g(n))$
 - $n \geq 2$ 일 때, $3n + 2 \leq 4n$ 이므로 $3n + 2 = O(n)$
 - $n \geq 5$ 일 때, $10n^2 + 4n \leq 11n^2$ 이므로 $10n^2 + 4n = O(n^2)$
 - $n \geq 2$ 일 때, $10n^2 + 4n \leq 10n^4$ 이므로 $10n^2 + 4n = O(n^4)$
- 모든 n 에 대해 $g(n)$ 값은 $f(n)$ 의 상한값 (Upper Bound)
 - 의미상으로 유익하기 위해서는 $g(n)$ 이 가능한 작아야 함
 - $3n + 2 = O(n)$ 이라고 하지만 $3n + 2 = O(n^2)$ 이라고 하지는 않음
 - 물론 후자가 틀린 것은 아님

Definition of Omega (Ω)

- 모든 n , $n \geq n_0$ 에 대해 $f(n) \geq cg(n)$ 인 조건을 만족시키는 두 양의 상수 c 와 n_0 가 존재하기만 하면 $f(n) = \Omega(g(n))$
 - $n \geq 1$ 일 때, $3n + 2 \geq 3n$ 이므로 $3n + 2 = \Omega(n)$
 - $n \geq 1$ 일 때, $10n^2 + 4n \geq n^2$ 이므로 $10n^2 + 4n = \Omega(n^2)$
 - $n \geq 1$ 일 때, $10n^2 + 4n \geq 1$ 이므로 $10n^2 + 4n = \Omega(1)$
- 모든 n 에 대해 $g(n)$ 값은 $f(n)$ 의 하한값 (Lower Bound)
 - 의미상으로 유익하기 위해서는 $g(n)$ 이 가능한 커야 함
 - $3n + 2 = \Omega(n)$ 이라고 하지만 $3n + 2 = \Omega(1)$ 이라고 하지는 않음
 - 물론 후자가 틀린 것은 아님

Definition of Theta (Θ)

- 모든 $n, n \geq n_0$ 에 대해 $c_1g(n) \leq f(n) \leq c_2g(n)$ 인 조건을 만족시키는 세 양의 상수 c_1, c_2 와 n_0 가 존재하기만 하면 $f(n) = \Theta(g(n))$
 - $n \geq 2$ 일 때, $3n + 2 \geq 3n$ 이고 $3n + 2 \leq 4n$ 이므로 $3n + 2 = \Theta(n)$
- 모든 n 에 대해 $g(n)$ 값은 $f(n)$ 의 상한값이자 하한값



Asymptotic Notation : Sum Function

```
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 }
```

s/e : 실행당 단계 수 (Step per Execution)			
행 번호	s/e	빈도	단계 수
1	0	0	$\Theta(0)$
2	1	1	$\Theta(1)$
3	1	$n + 1$	$\Theta(n)$
4	1	n	$\Theta(n)$
5	1	1	$\Theta(1)$
6	0	1	$\Theta(0)$
$t_{Sum}(n) = \Theta(\max_{1 \leq i \leq 6} \{g_i(n)\}) = \Theta(n)$			

Asymptotic Notation : Rsum Function

```

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 }
    
```

		s/e : 실행당 단계 수 (Step per Execution)			
행 번호	s/e	빈도		단계 수	
		$n = 0$	$n > 0$	$n = 0$	$n > 0$
1	0	1	1	0	$\Theta(0)$
2(a)	1	1	1	1	$\Theta(1)$
2(b)	1	1	0	1	$\Theta(0)$
3	$1 + t_{Rsum}(n - 1)$	0	1	0	$\Theta(1 + t_{Rsum}(n - 1))$
4	0	1	1	0	$\Theta(0)$
		$t_{Rsum}(n) =$		2	$\Theta(1 + t_{Rsum}(n - 1))$

Asymptotic Notation : Add Function

```
void Add(int** a, int** b, int** c, int m, int n)
1 {
2     for (int i = 0; i < m; i++)
3         for (int j = 0; j < n; j++)
4             c[i][j] = a[i][j] + b[i][j];
5 }
```

s/e : 실행당 단계 수 (Step per Execution)			
행 번호	s/e	빈도	단계 수
1	0	0	$\Theta(0)$
2	1	$\Theta(m)$	$\Theta(m)$
3	1	$\Theta(mn)$	$\Theta(mn)$
4	1	$\Theta(mn)$	$\Theta(mn)$
5	0	1	$\Theta(0)$
총 단계 수			$\Theta(mn)$

Asymptotic Notation : Binary Search

```
int BinarySearch(int* a, const int x, const int n)
{ // Search the sorted array a[0], ... , a[n-1] for x.
    int left = 0, right = n - 1;
    while (left <= right)
    { // There are more elements
        int middle = (left + right) / 2;
        if (x < a[middle]) right = middle - 1;
        else if (x > a[middle]) left = middle + 1;
        else return middle;
    } // End of while
    return -1; // Not found
}
```

2	5	9	11	15	18
---	---	---	----	----	----

값이 정렬된 배열이 있을 때, Best Case는 찾는 값이 middle에 바로 있는 경우(9를 검색)

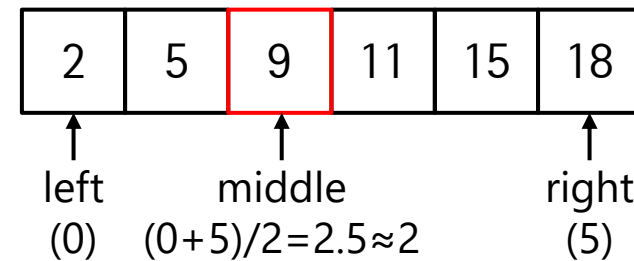
2	5	9	11	15	18
↑		↑			↑
left		middle			right
(0)		$(0+5)/2=2.5\approx 2$			(5)

$x = 9$, $a[\text{middle}] = a[2] = 9$ 이므로
9가 있는 위치 2를 반환
이 때, 시간 복잡도는 $O(1)$!

Asymptotic Notation : Binary Search

```
int BinarySearch(int* a, const int x, const int n)
{ // Search the sorted array a[0], ... , a[n-1] for x.
    int left = 0, right = n - 1;
    while (left <= right)
    { // There are more elements
        int middle = (left + right) / 2;
        if (x < a[middle]) right = middle - 1;
        else if (x > a[middle]) left = middle + 1;
        else return middle;
    } // End of while
    return -1; // Not found
}
```

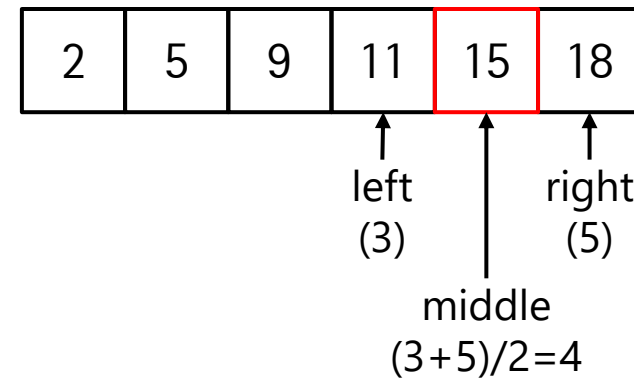
Worst Case는 배열에 없는 값을 검색할 때
(24를 검색)



$x = 24$, $a[\text{middle}] = a[2] = 9$ 이므로
left를 3으로 바꾼 뒤 다시 수행

Asymptotic Notation : Binary Search

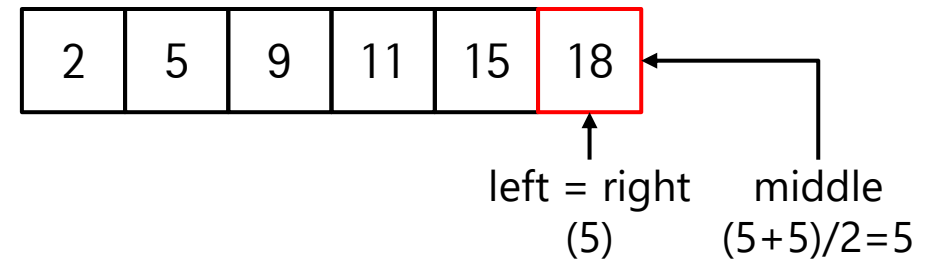
```
int BinarySearch(int* a, const int x, const int n)
{ // Search the sorted array a[0], ... , a[n-1] for x.
  int left = 0, right = n - 1;
  while (left <= right)
  { // There are more elements
    int middle = (left + right) / 2;
    if (x < a[middle]) right = middle - 1;
    else if (x > a[middle]) left = middle + 1;
    else return middle;
  } // End of while
  return -1; // Not found
}
```



x = 24, a[middle] = a[4] = 15이므로
left를 5로 바꾼 뒤 다시 수행

Asymptotic Notation : Binary Search

```
int BinarySearch(int* a, const int x, const int n)
{ // Search the sorted array a[0], ... , a[n-1] for x.
  int left = 0, right = n - 1;
  while (left <= right)
  { // There are more elements
    int middle = (left + right) / 2;
    if (x < a[middle]) right = middle - 1;
    else if (x > a[middle]) left = middle + 1;
    else return middle;
  } // End of while
  return -1; // Not found
}
```



$x = 24$, $a[\text{middle}] = a[5] = 18$ 이므로
left를 6로 바꾼 뒤 다시 수행

하지만, while (left <= right) 문에서
left = 6, right = 5이므로 while (false),
while 문을 빠져나오게 되어 -1을 반환

이 때, 시간 복잡도는?

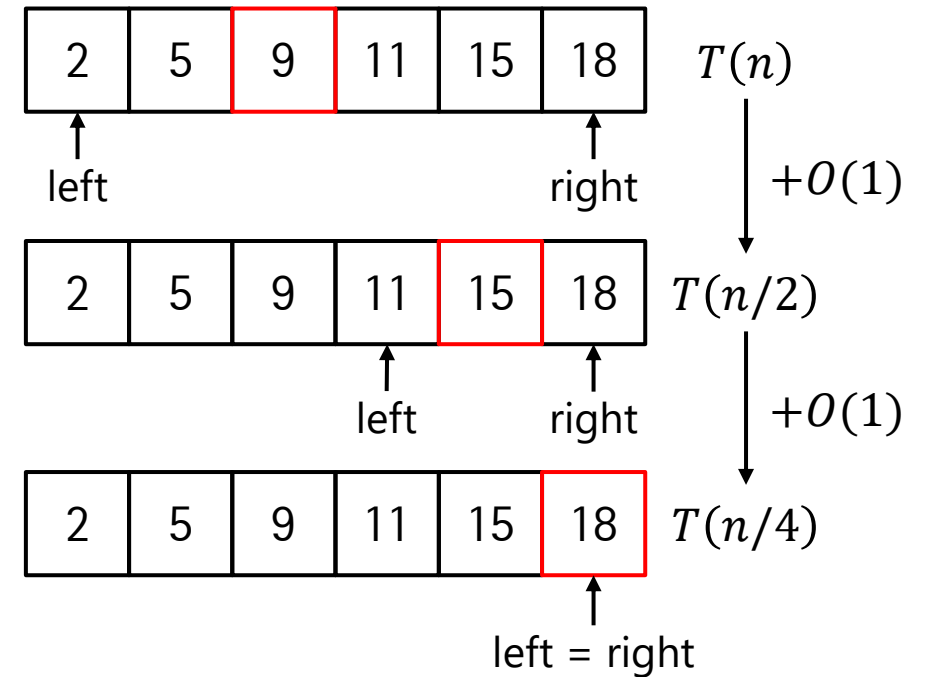
Asymptotic Notation : Binary Search

```
int BinarySearch(int* a, const int x, const int n)
{ // Search the sorted array a[0], ... , a[n-1] for x.
  int left = 0, right = n - 1;
  while (left <= right)
  { // There are more elements
    int middle = (left + right) / 2;
    if (x < a[middle]) right = middle - 1;
    else if (x > a[middle]) left = middle + 1;
    else return middle;
  } // End of while
  return -1; // Not found
}
```

시간 복잡도 계산의 핵심은 “while 문이 얼마나 반복되었는가?”

□ 부분이 얼마나 많이 수행되었느냐가 관건

□ 부분은 상수 시간에 수행되므로 $O(1)$



따라서, 시간 복잡도는

$$T(n) = T\left(\frac{n}{2}\right) + O(1)!$$

Asymptotic Notation : Binary Search

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

$$= T\left(\frac{n}{2^2}\right) + 2O(1)$$

= ...

$$= T\left(\frac{n}{2^k}\right) + kO(1)$$

$n = 2^k, T(1) = O(1)$ 라면

$$\therefore T(n) = T(1) + kO(1)$$

$$= (k + 1)O(1)$$

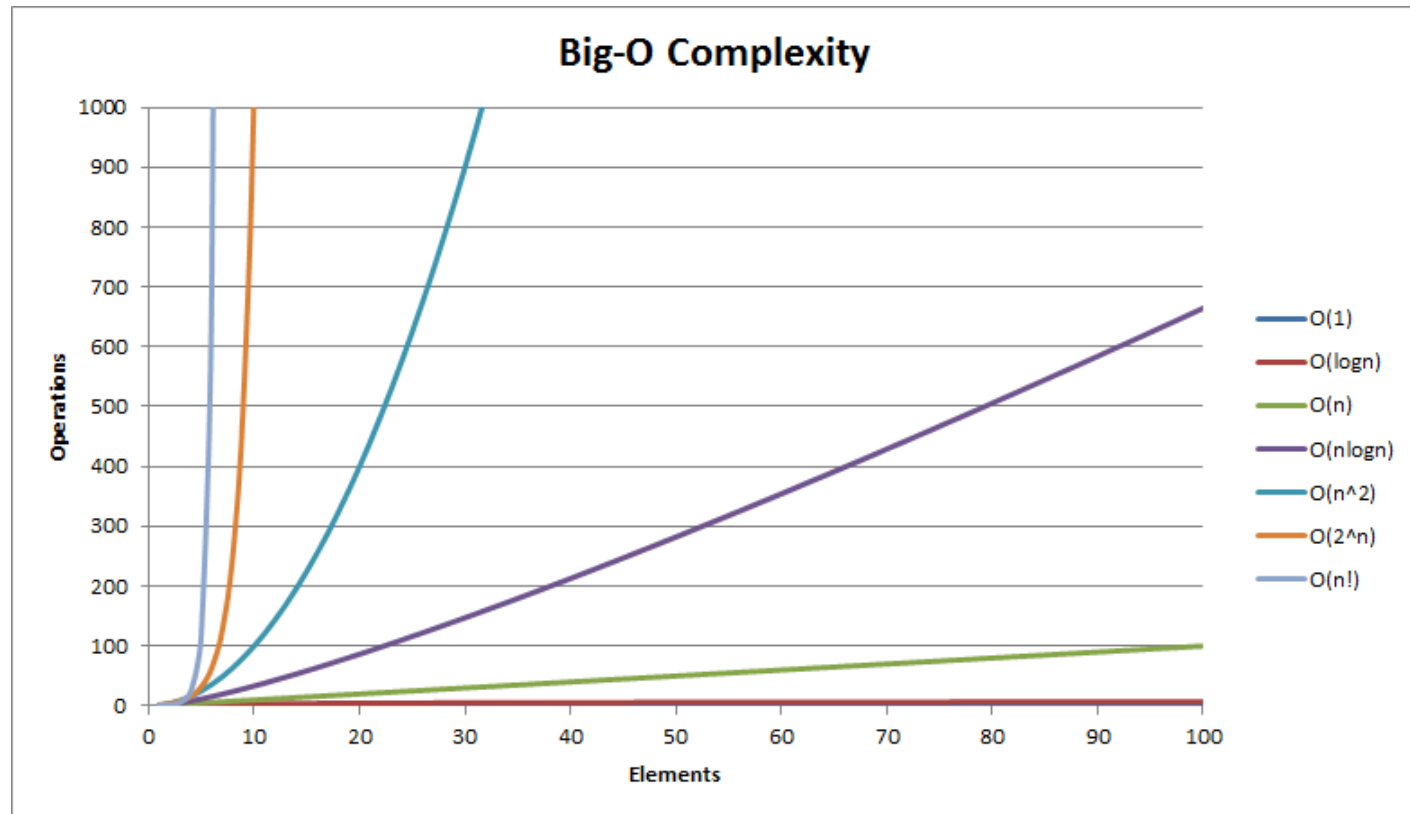
$n = 2^k$ 이므로 $k = \log_2 n$

$$\therefore T(n) = (\log_2 n + 1)O(1)$$

$$= O(\log_2 n) = O(\log n)$$

Orders of Common Functions

Name	Complexity class	Running time ($T(n)$)
constant time		$O(1)$
inverse Ackermann time		$O(\alpha(n))$
iterated logarithmic time		$O(\log^* n)$
log-logarithmic		$O(\log \log n)$
logarithmic time	DLOGTIME	$O(\log n)$
polylogarithmic time		$\text{poly}(\log n)$
fractional power		$O(n^c)$ where $0 < c < 1$
linear time		$O(n)$
"n log star n" time		$O(n \log^* n)$
linearithmic time		$O(n \log n)$
quadratic time		$O(n^2)$
cubic time		$O(n^3)$
polynomial time	P	$2^{O(\log n)} = \text{poly}(n)$
quasi-polynomial time	QP	$2^{\text{poly}(\log n)}$
sub-exponential time (first definition)	SUBEXP	$O(2^{n^\epsilon})$ for all $\epsilon > 0$
sub-exponential time (second definition)		$2^{o(n)}$
exponential time (with linear exponent)	E	$2^{O(n)}$
exponential time	EXPTIME	$2^{\text{poly}(n)}$
factorial time		$O(n!)$
double exponential time	2-EXPTIME	$2^{2^{\text{poly}(n)}}$



https://en.wikipedia.org/wiki/Big_O_notation
<http://bigocheatsheet.com/>

Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Stack	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Singly-Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Doubly-Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Skip List	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log(n))$
Hash Table	-	$O(1)$	$O(1)$	$O(1)$	-	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Binary Search Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Cartesian Tree	-	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	-	$O(n)$	$O(n)$	$O(n)$	$O(n)$
B-Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Red-Black Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Splay Tree	-	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	-	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
AVL Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$	$O(\log(n))$
Mergesort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$
Timsort	$O(n)$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$
Heapsort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(1)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell Sort	$O(n)$	$O((n \log(n))^2)$	$O((n \log(n))^2)$	$O(1)$
Bucket Sort	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(n)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$

Graph & Heap Operations

Node / Edge Management	Storage	Add Vertex	Add Edge	Remove Vertex	Remove Edge	Query
Adjacency list	$O(V + E)$	$O(1)$	$O(1)$	$O(V + E)$	$O(E)$	$O(V)$
Incidence list	$O(V + E)$	$O(1)$	$O(1)$	$O(E)$	$O(E)$	$O(E)$
Adjacency matrix	$O(V ^2)$	$O(V ^2)$	$O(1)$	$O(V ^2)$	$O(1)$	$O(1)$
Incidence matrix	$O(V \cdot E)$	$O(V \cdot E)$	$O(V \cdot E)$	$O(V \cdot E)$	$O(V \cdot E)$	$O(E)$

Type	Time Complexity						
	Heapify	Find Max	Extract Max	Increase Key	Insert	Delete	Merge
Linked List (sorted)	-	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(m+n)$
Linked List (unsorted)	-	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Binary Heap	$O(n)$	$O(1)$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(m+n)$
Binomial Heap	-	$O(1)$	$O(\log(n))$	$O(\log(n))$	$O(1)$	$O(\log(n))$	$O(\log(n))$
Fibonacci Heap	-	$O(1)$	$O(\log(n))$	$O(1)$	$O(1)$	$O(\log(n))$	$O(1)$