게임 자료구조와 알고리즘

-CHAPTER6-

SOULSEEK

목차

1. 헤더파일과 구현파일
2. 자료구조와 알고리즘의 이해
3. 재귀**(Recursion)**

헤더파일과 구현파일(.H, .CPP)

1. 헤더파일과 구현파일(.H, .CPP)

- 헤더파일(.h)는 선언부분, 구현파일(.cpp)는 구현부분
- 헤더와 구현은 각각 서로 파일명을 동일하게 해준다.
- 파일 묶음(.h, .cpp)단위로 특정 목적을 가지는 함수의 집합으로 만들어서 여러 곳에서 재사용 하기위해서 필요하다.
- 여러 개의 헤더파일을 사용해야 할 경우 불러오는 순서에 주의하자.

1. 헤더파일과 구현파일(.H, .CPP)

```
#pragma once
#include <iostream>
#include <cmath>
using namespace std;

struct Point
{
	int x, y;
};

double Distance(Point p1, Point p2)
{
	double distance;
	distance = sqrt(pow(p1.x - p2.x, 2) + pow(p1.y - p2.y, 2));

	return distance;
}

void main()
{
	Point a = { 0, 0 };
	Point b = { 3, 4 };

	double dist_a_b = Distance(a, b);

	cout << "(" << a.x << ", " << a.y << ") 와 ";
	cout << "(" << b.x << ", " << b.y << ") 의 거리 = " << dist_a_b << endl;
}
```

1. 헤더파일과 구현파일(.H, .CPP)

Distance.h

```
#include <iostream>
#include <cmath>
using namespace std;

struct Point
{
	int x, y;
};

double Distance(Point p1, Point p2);
```

Distance.cpp

```
#include "Distance.h"

void main()
{
Point a = { 0, 0 };
Point b = { 3, 4 };

double dist_a_b = Distance(a, b);

cout << "(" << a.x << ", " << a.y << ") 와 ";
cout << "(" << b.x << ", " << b.y << ") 의 거리 = “;
cout << dist_a_b << endl;

return;
}

double Distance(Point p1, Point p2)
{
double distance;
distance = sqrt(pow(p1.x - p2.x, 2) + pow(p1.y - p2.y, 2));

return distance;
}
```

1. 헤더파일과 구현파일(.H, .CPP)

Distance.h

```
#include <iostream>
#include <cmath>
using namespace std;

struct Point
{
	int x, y;
};

double Distance(Point p1, Point p2);
```

Diatance.cpp

```
#include "Distance.h"

double Distance(Point p1, Point p2)
{
	double distance;
	distance = sqrt(pow(p1.x - p2.x, 2) + pow(p1.y - p2.y, 2));

	return distance;
}
```

Main.cpp

```
#include "Distance.h"

void main()
{
	Point a = { 0, 0 };
	Point b = { 3, 4 };

	double dist_a_b = Distance(a, b);

	cout << "(" << a.x << ", " << a.y << ") 와 ";
	cout << "(" << b.x << ", " << b.y << ") 의 거리 = " << dist_a_b << endl;

	return;
}
```

1. 헤더파일과 구현파일(.H, .CPP)

Point.h

```
struct Point
{
	int x, y;
};
```

Diatance.h

```
#include <cmath>
#include "Point.h"

double Distance(Point p1, Point p2);
```

Main.cpp

```
#include <iostream>
#include "Point.h"
#include "Distance.h”

using namespace std;

void main()
{
	Point a = { 0, 0 };
	Point b = { 3, 4 };

	double dist_a_b = Distance(a, b);

	cout << "(" << a.x << ", " << a.y << ") 와 ";
	cout << "(" << b.x << ", " << b.y << ") 의 거리 = " << dist_a_b << endl;

	return;
}
```

Distance.cpp

```
#include "Point.h"
#include "Distance.h“

double Distance(Point p1, Point p2)
{
	double distance;
	distance = sqrt(pow(p1.x - p2.x, 2) + pow(p1.y - p2.y, 2));

	return distance;
}
```

1. 헤더파일과 구현파일(.H, .CPP)

#ifndef ~ #endif 와 #pragma once

- #pragma once
  - 한번만 컴파일 하고 그 뒤로부터는 동일한 파일의 경우, 읽기조차 하지 않는다. 그래서 파일 해석 단계의 속도가 빠르다. 하지만 컴파일러 지시자로 특정 컴파일러에서만 동작하는 지시자이며 Visual C++ 5.0 이상에서만 동작한다.

- #ifndef
  - 헤더파일을 여러 번 include 하게 되면 define 여부를 계속 체크해야 한다. 컴파일 단계 중에서 파일 해석 단계의 속도가 느리다. 하지만 전처리기 지시자라서 모든 컴파일러에서도 동작 할 수 있다.

1. 헤더파일과 구현파일(.H, .CPP)

```
#ifndef POINT_H
#define POINT_H

struct Point
{
    int x, y;
};

#endif
```

```
#pragma once

struct Point
{
    int x, y;
};
```

```
#ifndef DISTANCE_H
#define DISTANCE_H

#include <cmath>
#include "Point.h"

double Distance(Point p1, Point p2);

#endif
```

```
#pragma once
#include <cmath>
#include "Point.h"

double Distance(Point p1, Point p2);
```

자료구조와 알고리즘의 이해

2. 자료구조와 알고리즘의 이해

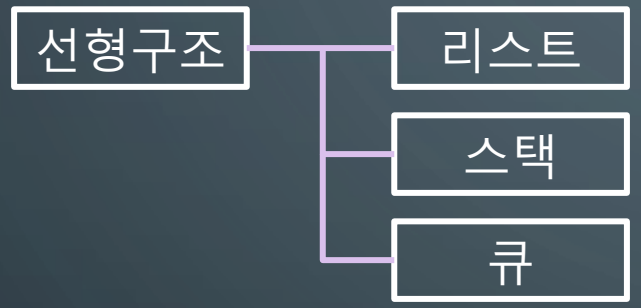
C 언어와 관련된 내용 중 최소한 알아야 할 내용(자료구조를 배우기 위해)

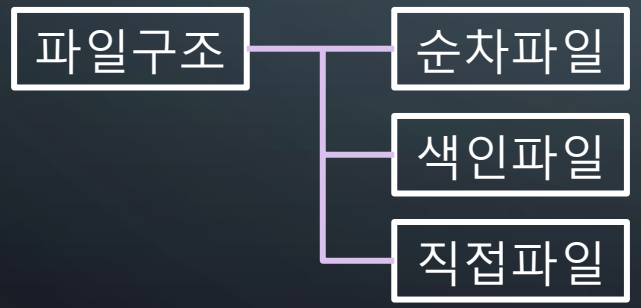
- **구조체를 정의할 줄 안다.**
- 메모리의 동적 할당과 관련하여 이해한다.
- 포인터와 관련하여 이해와 활용에 부담이 없다.
- **헤더파일을 정의하고 활용할 줄 안다.**
- 각종 매크로를 이해하고 #ifndef ~ #endif의 의미를 안다.
- **다수의 소스파일, 헤더파일로 구성된 프로그램 작성이 가능하다.**
- **재귀함수에 어느 정도 익숙하다.**

2. 자료구조와 알고리즘의 이해

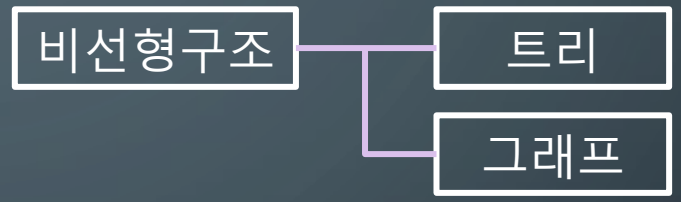
프로그램이란 **데이터를 표현**하고, 그렇게 표현된 **데이터를 처리** 하는 것이다.

자료구조





알고리즘



2. 자료구조와 알고리즘의 이해

```
int main(void)
{
    //배열 선언
    int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    //배열에 저장된 값의 합
    for(int i = 0; i < 10; i++)
    {
        sum += arr[i];
    }
}
```

자료구조

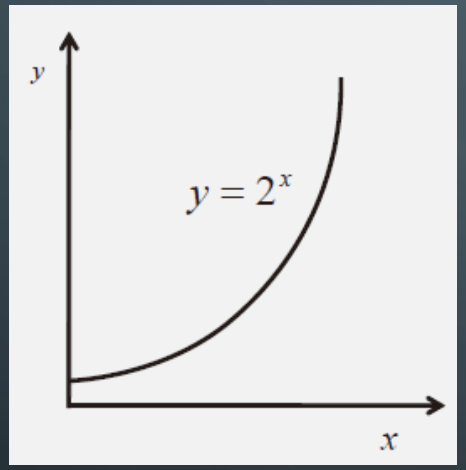
알고리즘

- 자료구조는 알고리즘을 구현하려고 하는 것에 의해 영향을 받는 의존성을 가지고 있다. 그렇기 때문에 알고리즘에 대한 이해도 필요하다.
- 자료구조의 내용을 직접 그려보자.

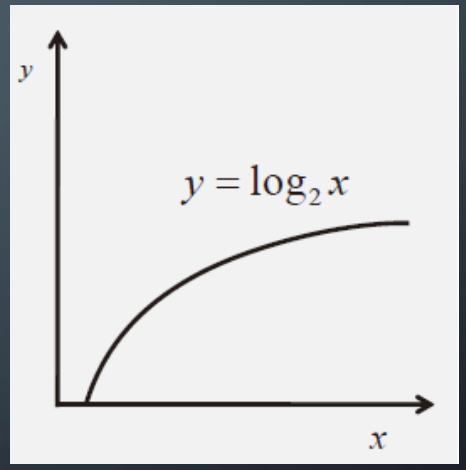
2. 자료구조와 알고리즘의 이해

자료구조와 알고리즘의 성능분석 방법

- 선형 자료구조에서는 성능분석이 쉽가 그렇기 때문에 이런 분석 방법에는 비선형 자료구조에는 적극 활용하게 될 것이다.
- 지수식과 로그식에 대한 지식이 필요하다.



지수식



로그식

X축은 데이터의 수를 나타내고 Y는 시간을 말한다, 지수식이 통용되는 성능보다 로그식이 통용되는 성능을 목적으로 하자

2. 자료구조와 알고리즘의 이해

성능을 평가하는 두 가지 요소

- 시간 복잡도(time complexity) : 얼마나 빠른가? – CPU에 얼마나 영향을 주는가?
- 공간 복잡도(space complexity) : 얼마나 메모리를 적게 쓰는가?
- 시간 복잡도를 더 중요시 한다.

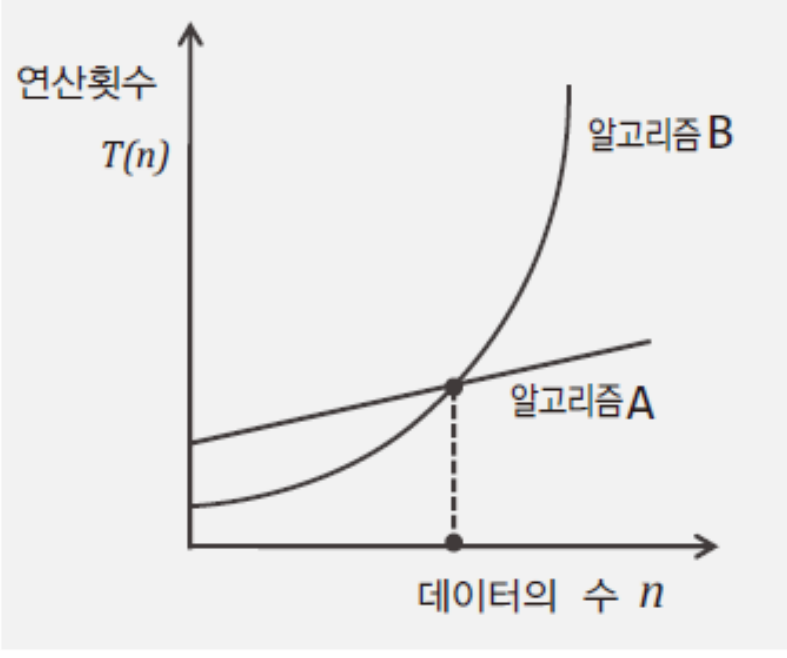
시간 복잡도의 평가 방법

- **중심이 되는 특정 연산의 횟수**를 세어서 평가를 한다.
  - CPU가 얼마나 일을 많이 하느냐.
- 데이터의 수에 대한 연산 횟수의 함수 T(n)을 구한다.
  - 함수로 만들고 그래프로 표현해서 속도의 예측이 가능해진다.

2. 자료구조와 알고리즘의 이해

알고리즘의 수행 속도 비교 기준

- 데이터의 수가 적은 경우의 수행 속도는 큰 의미가 없다.
- 데이터의 수에 따른 수행 속도의 변화 정도를 기준으로 한다.



데이터수의 적을 때와 많을 때를 선택적 구분을 해서는 안된다. 우리가 게임을 만들 때 데이터를 처리할 때 얼마나 늘어날지 알 수 없기 때문에 점점 **늘어 날수록 속도가 빠른 것**을 선택해야 한다.

2. 자료구조와 알고리즘의 이해

```
int LSearch(int arr[], int len, int target)
{
	for (int i = 0; i < len; i++)
	{
		if (arr[i] == target)
			return i;
	}

	return -1;
}
```

주 연산자인 “==“에 의해 “<, ++”의 연산 횟수를 결정지어지게 된다.

최상의 경우

- 배열의 맨 앞에서 대상을 찾을 경우
- 만족스러운 상황이므로 성능평가의 주 관심이 아니다.

$T(n) = 1$

최악의 경우

- 배열의 끝에서 찾거나 대상이 저장되지 않은 경우
- 만족스럽지 못한 상황이므로 성능평가의 주 관심이 아니다.

$T(n) = n$

2. 자료구조와 알고리즘의 이해

평균적인 경우

- 가장 현실적인 경우에 해당한다.
  - 일반적으로 등장하는 상황에 대한 경우의 수이다.
  - 최상의 경우와 달리 알고리즘 평가에 도움이 된다.
  - 하지만 계산하기가 어렵고 객관적 평가가 쉽지 않다.

- 평균적인 경우의 복잡도 계산이 어려운 이유
  - 평균적인 경우의 연출이 어렵다.
  - 평균적인 경우임을 증명하기 어렵다.
  - 평균적인 경우는 상황에 따라 달라진다.

시간의 복잡도를 계산할 때는 **최악의 경우는 항상 동일**하기 때문에 그 상황을 가지고 복잡도를 계산해서 알고리즘을 평가하는 것이 좋다.

“데이터의 수가 **n**개일 때, **최악의 경우**에 해당하는 연산횟수는(비교연산의 횟수는) **n**이다.”

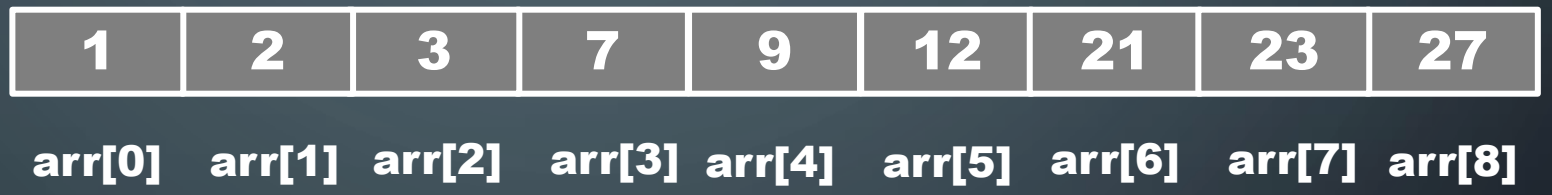
T(n) = n이 순차알고리즘의 시간 복잡도를 알려주는 계산이다.

2. 자료구조와 알고리즘의 이해

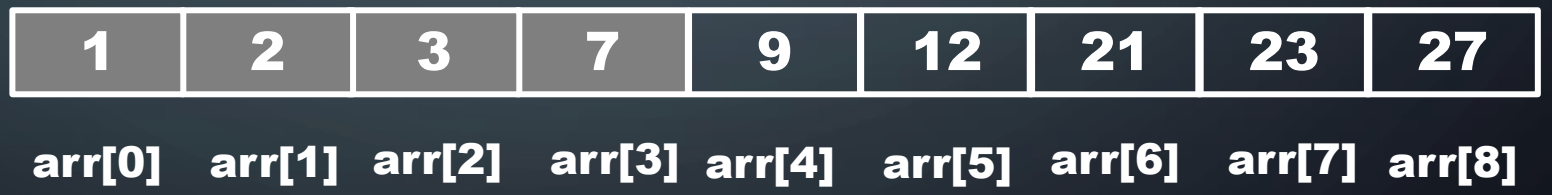
이진 탐색 알고리즘

- 순차 탐색 알고리즘보다 좋은 성능을 발휘한다
- 배열이 정렬이 되어 있어야한다 – 정렬의 기준이 있고 그것에 의해 정렬만 되어있으면 된다.
- 배열 처음과 끝을 합해서 반으로 나누고 가운데 원소 값을 알아와서 정렬한 기준을 근거로 비교해서 어디서 원하는 값을 찾을지 판단하고 계속 그렇게 반으로 나누면서 탐색한다.

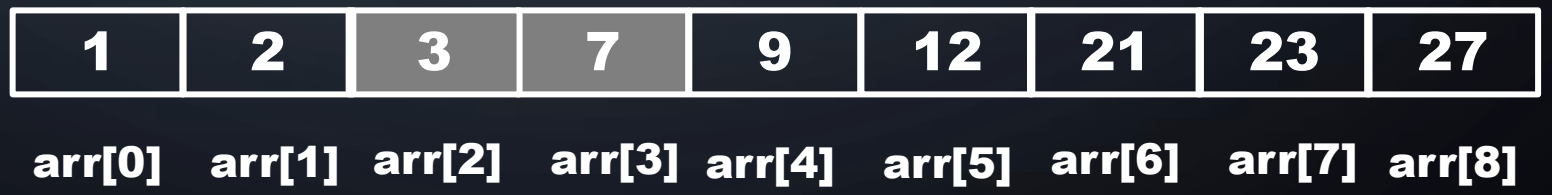
첫번째 탐색



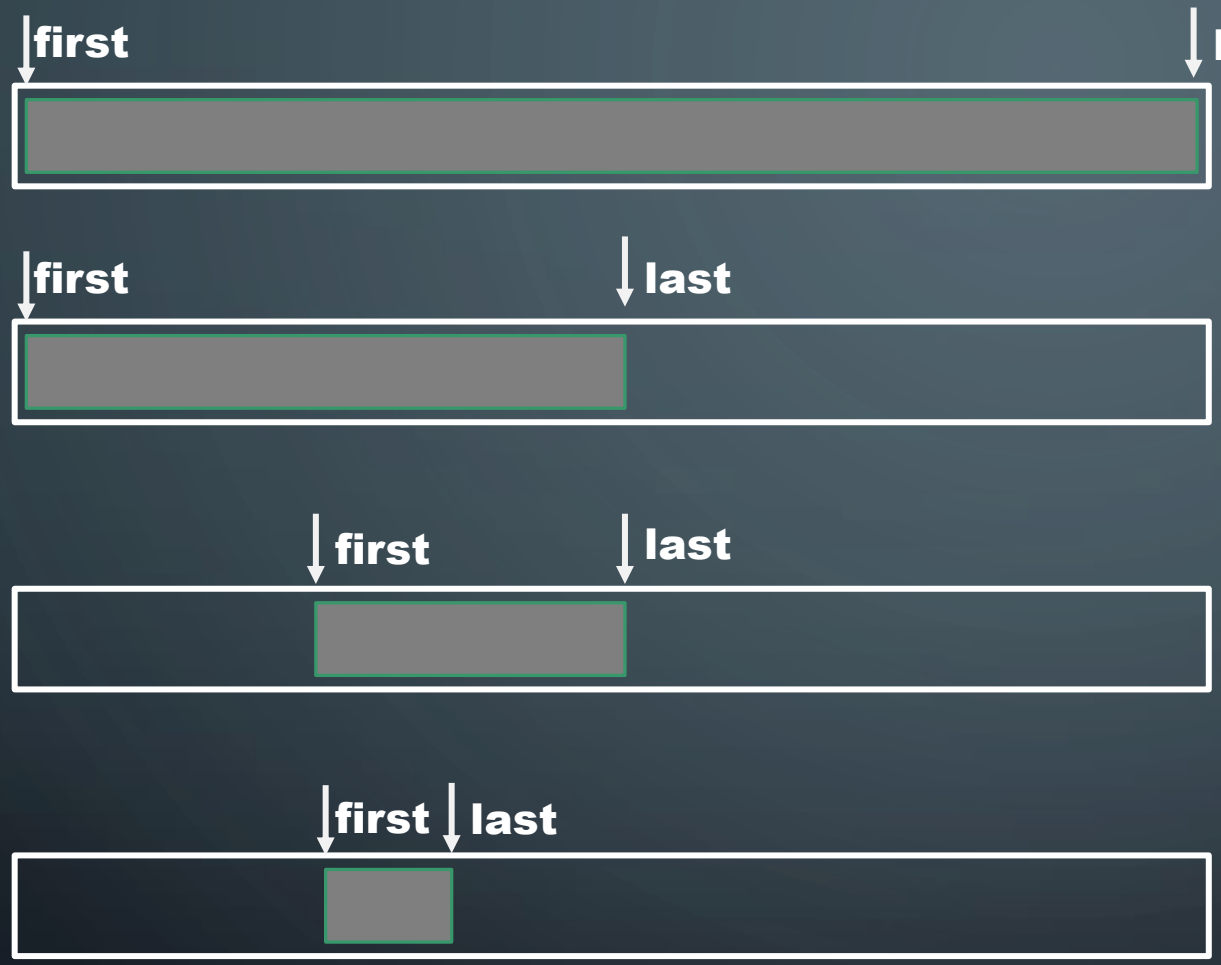
두번째 탐색



세번째 탐색



2. 자료구조와 알고리즘의 이해



- **First와 last가 만났다**는 것은 탐색 대상이 하나 남았다는 것을 뜻한다.
- **first와 last가 역전**될 때까지 탐색 과정을 계속 진행한다.

```
While(first <= last)
//first == last가 아니다.
{
    //이진 탐색 알고리즘의 진행
}
```

2. 자료구조와 알고리즘의 이해

```
int BSearch(int arr[], int len, int target)
{
	int first = 0; // 탐색대상의 시작 인덱스
	int last = len - 1; // 탐색 대상의 마지막 인덱스
	int mid;

	while (first <= last)
	{
		mid = (first + last) / 2; // 1번 과정

		If (target == arr[mid]) // 2번 과정
		{
			return mid;
		}
		else
		{
			if (target < arr[mid]) // 2번 과정
				ast = mid - 1; // 3번 과정
			else
				first = mid + 1; // 3번 과정
		}
	}

	return -1;
}
```

1. 시작 값과 종료 값을 이용해 가운데 값을 알아낸다.
2. 중앙 인덱스와 타켓 값과 같은지 정렬 조건에 의거해 비교해서 어떤 상황인지 판단한다.
3. 중앙 인덱스의 값과 비교했을 때, 어떤 위치에 있는 지 알아낸다.

-1혹은 +1을 추가하지 않으면 first <= mid <= last가 항상 성립이 되어 탐색 대상이 존재하지 않는 경우 fist와 last의 역전 현상이 발행하지 않는다. 즉, 무한루프에서 빠져나오지 못한다.

시간 복잡도 계산을 위한 핵심 연산은?

== 연산자!

따라서 == 연산자의 연산 횟수를 기준으로 시간 복잡도를 결정 할 수 있다.

2. 자료구조와 알고리즘의 이해

최악의 경우일때 – 데이터의 수를 알 수 없으니 n이라고 가정하자.

- 처음에 데이터 수가 **n**개일 때의 탐색과정에서 **1**회의 비교연산 진행
- 데이터 수를 반으로 줄여서 그 수가 **n/2**개일 때의 탐색과정에서 **1**회 비교연산 진행
- 데이터 수를 반으로 줄여서 그 수가 **n/4**개일 때의 탐색과정에서 **1**회 비교연산 진행
- 데이터 수를 반으로 줄여서 그 수가 **n/8**개일 때의 탐색과정에서 **1**회 비교연산 진행

n이 얼마인지 결정되지 않았으니 이 사이에
도대체 몇 번의 비교연산이 진행되는 지 알 수가 없다.

- 언제까지 비교연산이 진행된다는 것은 알 수 있는데 언제까지 인지는 답을 할 수 없다. 몇 번인지 대답을 할 수 있어야 함수를 구할 수 있는데 그러기 위해서는 약간 수학적인 도움이 필요 하다.

2. 자료구조와 알고리즘의 이해

비교 연산의 횟수의 예

- **8**이 **1**이 되기까지 **2**로 나눈 횟수 **3**회**,** 따라서 **비교연산 3회** 진행
- 데이터가 **1**개 남았을 때**,** 이때 마지막으로 **비교연산 1회** 진행

비교 연산의 횟수의 일반화

- **n**이 **1**이 되기까지 **2**로 나눈 횟수 **k**회**,** 따라서 **비교연산 k회** 진행
- 데이터가 **1**개 남았을 때**,** 이때 마지막으로 **비교연산 1회** 진행

비교 연산의 횟수의 1차 결론!

- 최악의 경우에 대한 시간 복잡도 함수 **T(n) = k + 1** – **k**만 구하면 완성이 된다**.**

2. 자료구조와 알고리즘의 이해

$$n * \left(\frac{1}{2}\right)^k = 1$$

n을 몇번 반으로 나눠야 1이 되는가? K가 나눠야 하는 횟수를 말하는 것.

알기 쉬운 공식으로 표현해보자!

$$n * \left(\frac{1}{2}\right)^k = 1 \blacktriangleright \mathbf{n} * 2^{-k} = 1 \blacktriangleright \mathbf{n} = 2^k$$

$$\blacktriangleright \log_2 n = \log_2 2^k \blacktriangleright \log_2 n = k \log_2 2 \blacktriangleright \log_2 n = k$$

T(n) = k + 1 ▶ **T(n) =** $\log_2 n + 1$ ▶ **T(n) =** $log_2\, n$

- 시간 복잡도의 목적은 n의 값에 따른 T(n)의 증가 및 감소의 정도를 판단하는 것이므로 +1은 생략이 가능하다.

2. 자료구조와 알고리즘의 이해

빅 – 오 표기법(Big – Oh Notation)

- T(n)에서 실제로 영향력을 끼치는 부분을 가리킨다.
- T(n)이 다항식으로 표현이 된 경우, **최고차항의 차수가 빅-오**가 된다.

T(n) = $n^2 + 2n + 1$

⬇

T(n) = $n^2 + 2n$

⬇

T(n) = n^2

⬇

O(n^2)

n의 변화에 따른 T(n)의 변화정도를 판단하는 것이 목적이니 +1은 무시할 수 있다.)

n	n^2	$2n$	$T(n)$	n^2의 비율
10	100	20	120	83.33%
100	10,000	200	10,200	98.04%
1,000	1,000,000	2,000	1,002,000	99.80%
10,000	100,000,000	20,000	100,020,000	99.98%
100,000	10,000,000,000	200,000	10,000,200,000	99.99%

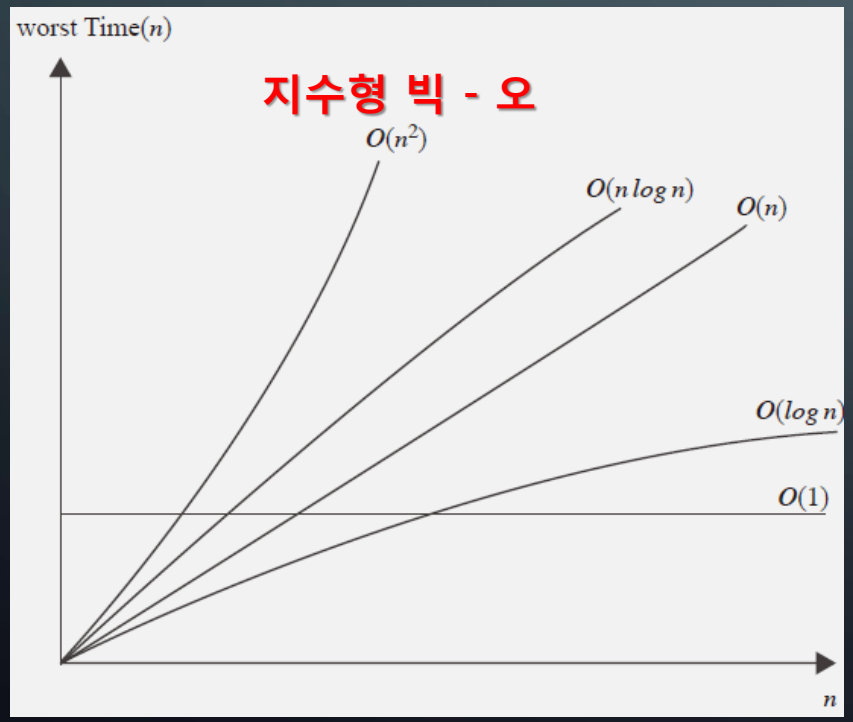
2n도 근사치 식의 구성에서 제외시킬 수 있음을 보인 결과!
n이 증가함에 따라 2n + 1이 차지하는 비율은 미미해진다.

2. 자료구조와 알고리즘의 이해

빅 – 오 결정의 예

- **T(n) = n^2 + 2n + 9** ▶ **O(n^2)**
- **T(n) = n^4 + n^3 + n^2 + 1** ▶ **O(n^4)**
- **T(n) = 5n^3 + 3n^2 + 2n^2 + 1** ▶ **O(n^3)**

- **T(n) = $a_m n^m$ + $a_{m-1} n^{m-1}$ + … + $a_1 n^1$ + a_0** ▶ **O(n^m)**



로그형 빅 - 오

상수형 빅 - 오

빅 – 오 복잡도 순서

O(1) < O(log n) < O(n)

< O(n log n) < O(n^2)

< O(n^3) < O(2^n)

2. 자료구조와 알고리즘의 이해

순차 탐색의 최악의 경우 시간 복잡도

- **T(n) = n ▶ O(n)**

이진 탐색의 최악의 경우 시간 복잡도

- **T(n) =** $\log_2 n$ **+ 1 ▶ O(**$\log\ n$**)**

n	순차 탐색 연산횟수	이진 탐색 연산횟수
500	500	9
5,000	5,000	13
50,000	50,000	16

- **BSWorstOpCount** 코드를 확인하면 이 결과값을 비교할 수 있다.

2. 자료구조와 알고리즘의 이해

학습과제

- 아래 나오는 식들의 빅 – 오를 구하시오 (비교가 힘들 땐 비교표를 활용해 보자.)
  - $3n + 2$
  - $7n^3 + 3n^2 + 2$
  - $2^n + n^2$
  - $n + \log n$
  - $n + n\log n$
  - $2^n + n^3$

재귀(RECURSION)

3. 재귀(RECURSION)

재귀함수의 이해

```
void Recursive()
{
    cout << “Recursive call!” << endl;
    Recursive();
}
```

원 본

```
Void Recursive()
{
    cout << “Recursive call!” << endl;
    Recursive();
}
```

```
Void Recursive()
{
    cout << “Recursive call!” << endl;
    Recursive();
}
```

```
Void Recursive()
{
    cout << “Recursive call!” << endl;
    Recursive();
}
```

```
Void Recursive()
{
    cout << “Recursive call!” << endl;
    Recursive();
}
```

3. 재귀(RECURSION)

팩토리얼

n! = n * (n - 1) * (n - 2) * (n - 3) * .... * 2 * 1 ▶ (n – 1)!

n! = n * (n - 1)!

```
#include <iostream>
using namespace std;

int Factorial(int n)
{
	if (n == 0)
		return 1;
	else
		return n * Factorial(n - 1);
}

int main()
{
	cout << "1! = " << Factorial(1) << endl;
	cout << "2! = " << Factorial(2) << endl;
	cout << "3! = " << Factorial(3) << endl;
}
```

3. 재귀(RECURSION)

피보나치 수열

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55……

수열의 **n**번째 값 **=** 수열의 **n – 1**번째 값 **+** 수열의 **n – 2**번째 값

Fibo(n) =

0	**▶ n = 1**
1	**▶ n = 2**
Fibo(n – 1) + Fibo(n – 2)	**▶ ohterwise**

```
int Fibo(int n)
{
    if (n == 1)
        return 0;
    else if (n == 2)
        return 1;
    else
        return Fibo(n - 1) + Fibo(n - 2);
}
```

3. 재귀(RECURSION)

학습과제

1. 앞의 피보나치 수열의 재귀를 완성하고 원하는 수를 입력 받아서 처리하게 만들자.
2. 이진탐색구조를 재귀를 이용하게 바꿔보자.
3. 하노이타워 게임을 자동으로 돌아가게 구현해보자.