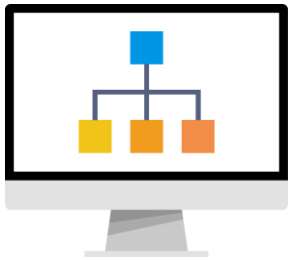
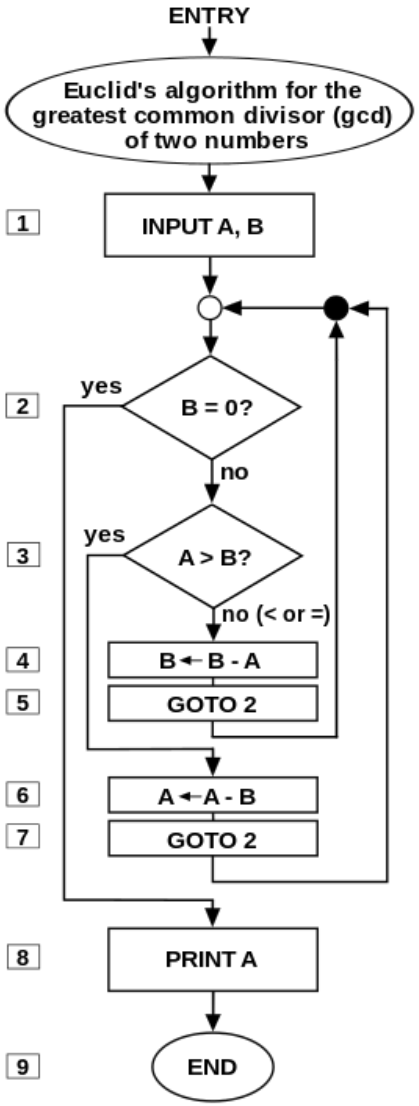


Data Structure & Algorithm
자료구조 및 알고리즘

1. 알고리즘의 복잡도

알고리즘 (Algorithm)

- 어떠한 문제를 해결하기 위한 여러 동작들의 집합
  - 예제) 유클리드 호제법
    - 최대 공약수를 구하는 알고리즘
  - 빠른 알고리즘과 느린 알고리즘



알고리즘의 평가

- 1부터 100,000,000까지 합한 결과를 구하시오.

- 알고리즘 1)
  - 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + … + 100,000,000
    - 덧셈 99,999,999회

- 알고리즘 2)
  - 등차수열의 합 = n × (첫 항 + 마지막 항) / 2
    - 덧셈 1회
    - 곱셈 1회
    - 나눗셈 1회

알고리즘의 평가

- 1부터 **N**까지 합한 결과를 구하시오.

- 알고리즘 1)
  - 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + … + **N**
    - 덧셈 **N-1**회

- 알고리즘 2)
  - 등차수열의 합 = **N** × (1 + **N**) / 2
    - 덧셈 1회
    - 곱셈 1회
    - 나눗셈 1회

알고리즘의 평가

- 1부터 **N**까지 합한 결과를 구하시오.

- 알고리즘 1)
  - 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + … + **N**
    - 덧셈 **N-1**회

O(N)

- 알고리즘 2)
  - 등차수열의 합 = **N** × (1 + **N**) / 2
    - 덧셈 1회
    - 곱셈 1회
    - 나눗셈 1회

O(1)

시간 복잡도와 공간 복잡도

- 알고리즘을 평가하는 두 가지 요소
  - 시간 복잡도 (time complexity) ➔ 얼마나 빠른가?
  - 공간 복잡도 (space complexity) ➔ 얼마나 메모리를 적게 쓰는가?

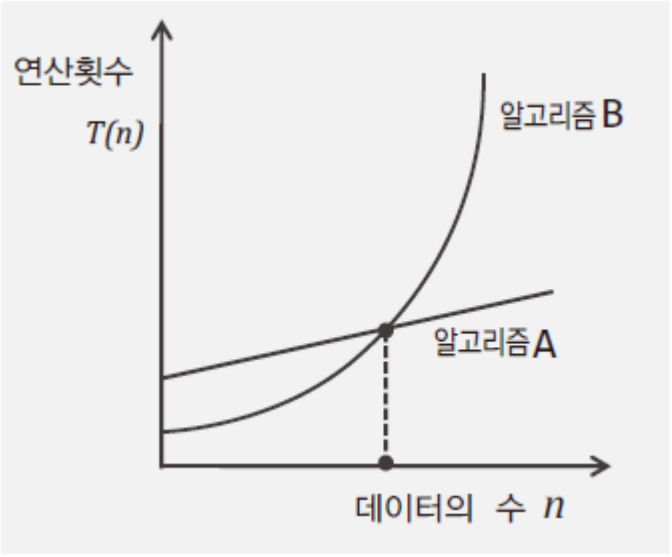
시간 복잡도와 공간 복잡도

- 알고리즘을 평가하는 두 가지 요소
  - 시간 복잡도 (time complexity) ➔ 얼마나 빠른가?
  - 공간 복잡도 (space complexity) ➔ 얼마나 메모리를 적게 쓰는가?

- 시간 복잡도의 평가 방법
  - 중심이 되는 특정 연산의 횟수를 세어서 평가
  - 데이터의 수에 대한 연산횟수의 함수 $T(n)$을 구한다

시간 복잡도와 공간 복잡도

- 알고리즘의 수행 속도 비교 기준
  - 데이터의 수에 따른 수행 속도의 변화 정도를 기준으로 한다.



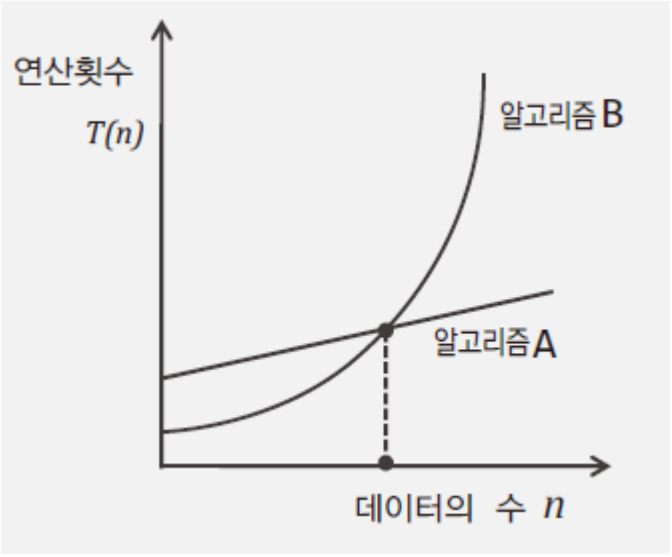
알고리즘 A와 B를 평가해보면?

n이 작을 때는 A가 더 좋다. 그러나 n이 특정 값보다 커지면 B가 더 좋다.

알고리즘 A가 B보다 **점근적으로 더 효율적**이다.

시간 복잡도와 공간 복잡도

- 알고리즘의 수행 속도 비교 기준
  - 데이터의 수에 따른 수행 속도의 변화 정도를 기준으로 한다.



알고리즘 A와 B를 평가해보면?

n이 작을 때는 A가 더 좋다. 그러나 n이 특정 값보다 커지면 B가 더 좋다.

알고리즘 A가 B보다 **점근적으로 더 효율적**이다.

A is "Asymptotically more efficient" than B.

순차 탐색 알고리즘과 시간 복잡도

```
// 순차 탐색(Linear Search) 알고리즘 함수
int LSearch(int ar[], int len, int target)
{
    int i;
    for(i=0; i<len; i++)
    {
        if(ar[i] == target)
            return i;       // 찾은 대상의 인덱스 값 반환
    }
    return -1;       // 찾지 못했음을 의미하는 값 반환
}
```

어떤 연산의 횟수를 세어야 할까?

최상의 경우와 최악의 경우

- 순차 탐색 상황 하나: 운이 좋은 경우
  - 배열의 맨 앞에서 대상을 찾는 경우
  - 만족스러운 상황이므로 성능평가의 주 관심이 아니다!
  - ‘최상의 경우(best case)’라 한다.

최상의 경우와 최악의 경우

- 순차 탐색 상황 하나: 운이 좋은 경우
  - 배열의 맨 앞에서 대상을 찾는 경우
  - 만족스러운 상황이므로 성능평가의 주 관심이 아니다!
  - ‘최상의 경우(best case)’라 한다.

- 순차 탐색 상황 둘: 운이 좋지 않은 경우
  - 배열의 끝에서 찾거나 대상이 저장되지 않은 경우
  - 만족스럽지 못한 상황이므로 성능평가의 주 관심이다!
  - ‘최악의 경우(worst case)’라 한다.

평균적인 경우(Average Case)

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

- 평균적인 경우의 복잡도 계산이 어려운 이유
  - ‘평균적인 경우’의 연출이 어렵다.
  - ‘평균적인 경우’임을 증명하기 어렵다.
  - ‘평균적인 경우’는 상황에 따라 달라진다.

반면 최악의 경우는 늘 동일하다.

순차 탐색 **최악의 경우** 시간 복잡도

"데이터의 수가 *n*개일 때,

최악의 경우에 해당하는

연산횟수는(비교연산의 횟수는) *n* 이다."

T(n) = n 최악의 경우를 대상으로 정의한 함수 *T(n)*

순차 탐색 **최악의 경우** 시간 복잡도

"데이터의 수가 *n*개일 때,

최악의 경우에 해당하는

연산횟수는(비교연산의 횟수는) *n* 이다."

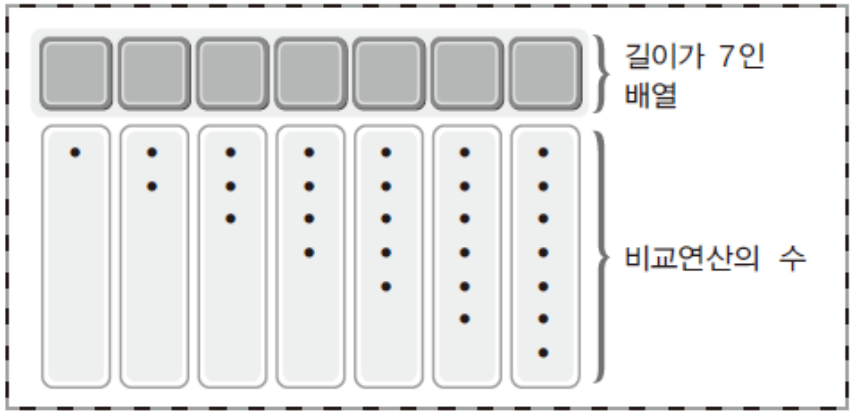
T(n) = n 최악의 경우를 대상으로 정의한 함수 *T(n)*

최선의 경우는? T(n) = 1

순차 탐색 **평균적 경우** 시간 복잡도

- 가정 1. 탐색 대상이 배열에 존재하지 않을 확률 50%
- 가정 2. 배열 내 각 요소에 탐색 대상 존재 확률은 동일

- 탐색 대상이 존재하지 않는 경우(50%), 연산 횟수는 n
- 탐색 대상이 존재하는 경우(50%), 연산 횟수는 약 n/2



$$T(n) = n \times \frac{1}{2} + \frac{n}{2} \times \frac{1}{2} = \frac{3}{4}n$$

가정 1과 2는 평균적인 경우라 할 수 있겠는가?

이진 탐색 알고리즘

- 선형 탐색의 경우 T(n) = n 이었다. 더 빨리 수행할 수 없을까?

- 이진 탐색 (Binary search)
  - 전체 배열을 반으로 나누어 나가면서 원하는 값을 찾아가는 방식
  - 순차 탐색보다 훨씬 좋은 성능을 보이지만, 배열이 정렬되어 있어야 한다는 제약이 있음

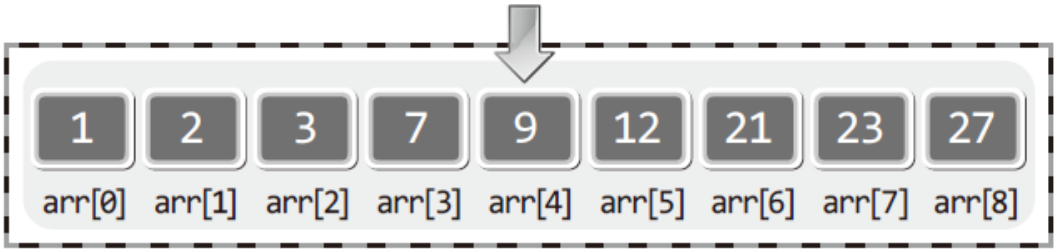
이진 탐색 알고리즘


첫 번째 탐색 대상의 범위
1 2 3 7 9 12 21 23 27
½
두 번째 탐색 대상의 범위
1 2 3 7 9 12 21 23 27
½
세 번째 탐색 대상의 범위
1 2 3 7 9 12 21 23 27

이진 탐색 알고리즘

- 이진 탐색 알고리즘의 첫 번째 탐색 시도:

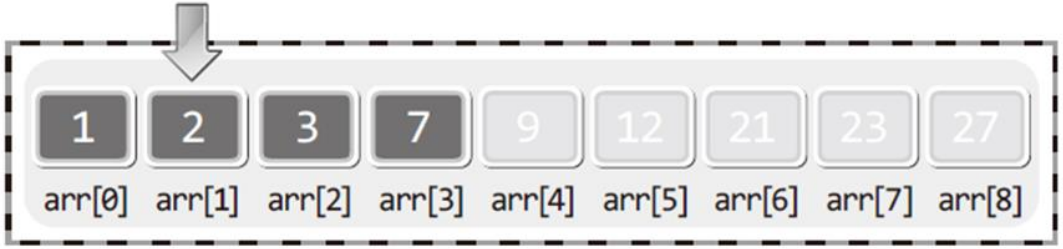
1. 배열 인덱스의 시작과 끝은 각각 0과 8이다.
2. 0과 8을 합하여 그 결과를 2로 나눈다.
3. 위의 결과로 나온 4를 인덱스 값으로 하여 arr[4]에 저장된 값이 원하는 값(3)인지 확인



3이 배열 내 어디에 저장되어 있는지 탐색

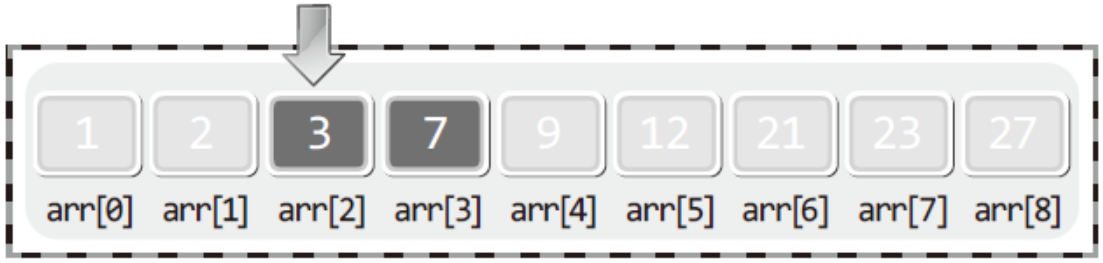
이진 탐색 알고리즘

- 이진 탐색 알고리즘의 두 번째 탐색 시도:
  1. arr[4]에 저장된 값 9와 탐색 대상인 3의 대소를 비교.
  2. 비교 결과 arr[4]>3이므로 탐색 범위를 인덱스 기준 0~3으로 제한
  3. 0과 3을 더하여 그 결과를 2로 나눈다. 이때 나머지는 버린다.
  4. 2로 나눠서 얻은 결과 1을 인덱스 값으로 arr[1]에 저장된 값이 3인지 확인한다.



이진 탐색 알고리즘

- 이진 탐색 알고리즘의 세 번째 탐색 시도:
  1. arr[1]에 저장된 값 2와 탐색 대상인 3의 대소를 비교.
  2. 비교 결과 arr[1]<3이므로 탐색 범위를 인덱스 기준 2~3으로 제한
  3. 2와 3을 더하여 그 결과를 2로 나눈다. 이때 나머지는 버린다.
  4. 2로 나눠서 얻은 결과 2를 인덱스 값으로 arr[2]에 저장된 값이 3인지 확인한다.

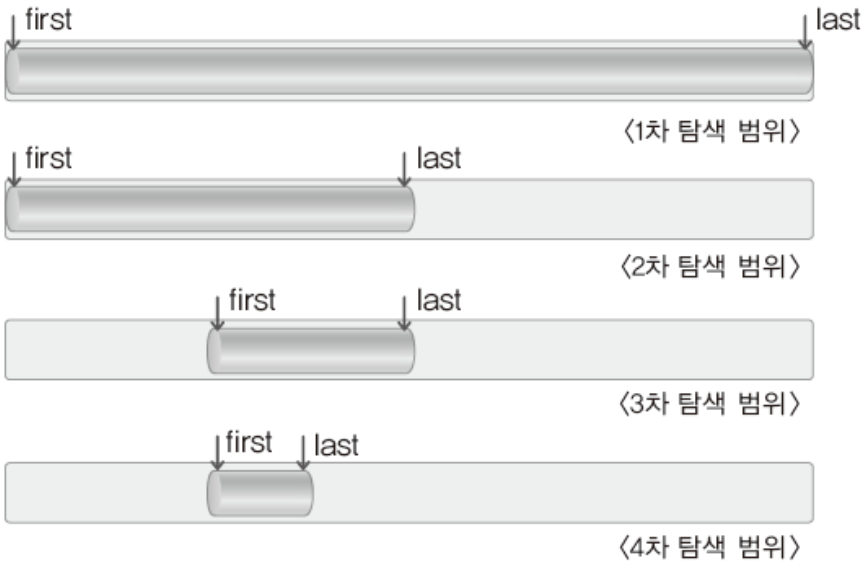


이진 탐색 알고리즘

이진 탐색 알고리즘의 핵심!

이진 탐색의 경우 매 반복마다 탐색 대상을 반으로 줄여 나가기 때문에 순차 탐색보다 비교적 좋은 성능을 보인다.

이진 탐색 알고리즘의 구현



- first와 last가 만났다는 것은 탐색 대상이 하나 남았다는 것을 뜻함
- 따라서 first와 last가 역전될 때까지 탐색 진행

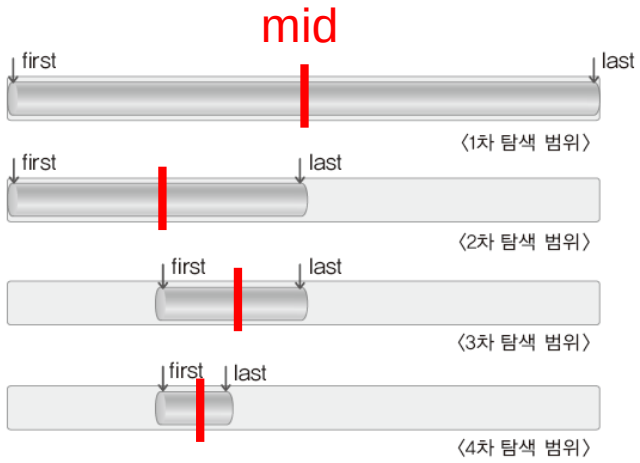
이진 탐색의 기본 골격

```
while(first <= last)        // first <= last 가 반복의 조건임에 주의!
{
      // 이진 탐색 알고리즘의 진행
}
```

이진 탐색 알고리즘의 구현

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

	while(first <= last)
	{
		mid = (first+last) / 2;		// 탐색 대상의 중앙을 찾는다.
		if(target == ar[mid])		// 중앙에 저장된 것이 타겟이라면
		{
			return mid;		// 탐색 완료!
		}
		else	// 타겟이 아니라면 탐색 대상을 반으로 줄인다.
		{
			if(target < ar[mid])
				last = mid-1;	// 왜 -1을 하였을까?
			else
				first = mid+1;	// 왜 +1을 하였을까?
		}
	}
	return -1;		// 찾지 못했을 때 반환되는 값 -1
}
```



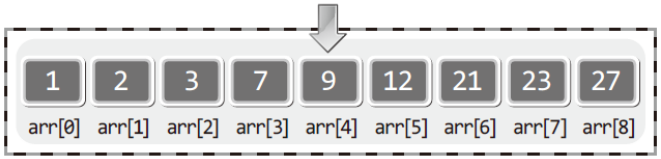
-1 혹은 +1을 추가하지 않으면

first <= mid <=last가 항상 성립이 되어,

탐색 대상이 존재하지 않는 경우 first와 last의 역전 현상이 발생하지 않는다!

이진 탐색 알고리즘에서 값이 없을 때?

- 8(존재하지 않음)을 찾는다고 가정
- (first=0, last=8, mid=4)
  - 8 < arr[4] (=9) 이므로
  - last = mid – 1 (=3)
- (first=0, last=3, mid=1)
  - 8 >= arr[1] (=2) 이므로
  - first = mid + 1 (=2)
- (first=2, last=3, mid=2)
  - 8 >= arr[2] (=3) 이므로
  - first = mid + 1 (=3)
- (first=3, last=3, mid=3)
  - 8 >= arr[3] (=7) 이므로
  - first = mid + 1 (=4)
- (first=**4**, last=**3**) ??




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

	while(first <= last)
	{
		mid = (first+last) / 2;		// 탐색 대상의 중앙을 찾는다.
		if(target == ar[mid])
		{
			return mid;		// 탐색 완료!
		}
		else		// 타겟이 아니라면 탐색 대상을 반으로 줄인다.
		{
			if(target < ar[mid])
				last = mid-1;	// 왜 -1을 하였을까?
			else
				first = mid+1;	// 왜 +1을 하였을까?
		}
	}
	return -1;		// 찾지 못했을 때 반환되는 값 -1
}
```

이진 탐색 알고리즘 구현 팁

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

	while(first <= last)
	{
		mid = (first+last) / 2;		// 탐색 대상의 중앙을 찾는다.
		if(target == ar[mid])		// 중앙에 저장된 것이 타겟이라면
		{
			return mid;			// 탐색 완료!
		}
		else	// 타겟이 아니라면 탐색 대상을 반으로 줄인다.
		{
			if(target < ar[mid])
				last = mid-1;	// 왜 -1을 하였을까?
			else
				first = mid+1;	// 왜 +1을 하였을까?
		}
	}
	return -1;		// 찾지 못했을 때 반환되는 값 -1
}
```

이진 탐색 알고리즘 시간 복잡도

- 최악의 경우에 대해서 시간 복잡도 계산

```
while(first <= last)
{
        mid = (first+last) / 2;
        if(target == ar[mid])
        {
                return mid;
        }
        else        // 타겟이 아니라면
        {
                . . . .
        }
}
```

시간 복잡도 계산을 위한 핵심 연산은?

== 연산자!

따라서 == 연산자의 연산 횟수를 기준으로 시간 복잡도를 결정할 수 있다.

데이터의 수가 n개일 때, 최악의 경우에 발생하는 비교연산의 횟수는?

이진 탐색 알고리즘 시간 복잡도

- 데이터의 수가 n개일 때 비교 연산의 횟수

· 처음에 데이터 수가 n개일 때의 탐색과정에서 1회의 비교연산 진행

· 데이터 수를 반으로 줄여서 그 수가 n/2개일 때의 탐색과정에서 1회 비교연산 진행

· 데이터 수를 반으로 줄여서 그 수가 n/4개일 때의 탐색과정에서 1회 비교연산 진행

· 데이터 수를 반으로 줄여서 그 수가 n/8개일 때의 탐색과정에서 1회 비교연산 진행

. . . . . n이 얼마인지 결정되지 않았으니

이 사이에 도대체 몇 번의 비교연산이 진행되는지 알 수가 없다! . . . . .

· 데이터 수를 반으로 줄여서 그 수가 1개일 때의 탐색과정에서 1회의 비교연산 진행

이러한 표현 방법으로는 객관적 성능 비교 불가능!

이진 탐색 알고리즘 시간 복잡도

비교 연산 횟수의 예

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

비교 연산 횟수의 일반화

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

비교 연산 횟수의 1차 결론

- 최악의 경우에 대한 시간 복잡도 함수 $T(n)=k+1$

이진 탐색 알고리즘 시간 복잡도

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

n을 몇번 반으로 나눠야 1이 되는가?
k가 나눠야 하는 횟수를 말한다.

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

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

이진 탐색 알고리즘 시간 복잡도

- 최악의 경우에 대한 시간 복잡도
  - T(n) = k + 1에서 $T(n) = \log_2 n + 1$

- 여기서 +1을 생략한 최종 시간 복잡도

$$T(n) = \log_2 n$$

시간 복잡도의 목적은 n의 값에 따른 T(n)의 증가 및 감소의 정도를 판단하는 것이므로 +1은 생략 가능!

그렇다면 +1이 아닌 +200 인 경우도 생략이 가능한가? **Big-O**

Big-O 표기법 (notation)

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

1차 근사식

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

2차 근사식

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

T(n)의 Big-O

$$O(n^2)$$

*T(n)*에서 실제로 영향력을 끼치는 부분을 가리켜 빅-오(Big-Oh)라 한다.

- n의 변화에 따른 T(n)의 변화 정도를 판단하는 것이 목적이므로 +1은 무시 가능

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 제외
  - n이 증가하면서 2n+1이 차지하는 비율은 미미해짐

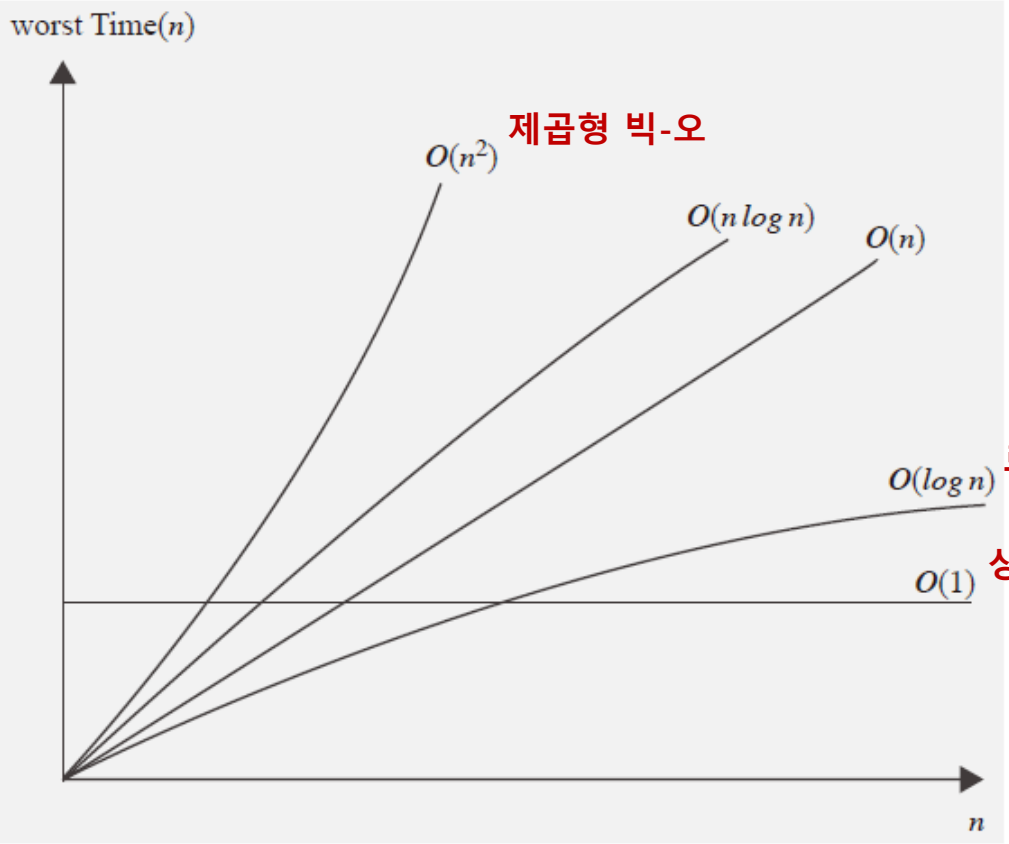
Big-O 구하기

- T(n)이 다항식으로 표현된 경우, 최고차항의 차수가 Big-O

- Big-O 결정의 예
  - $T(n)=n^2+2n+9$ ▶ $O(n^2)$
  - $T(n)=n^4+n^3+n^2+1$ ▶ $O(n^4)$
  - $T(n)=5n^3+3n^2+2n+1$ ▶ $O(n^3)$

- Big-O 결정의 일반화
  - $T(n)=a_m n^m+a_{m-1}n^{m-1}+\cdots+a_1 n^1+a_0$ ▶ $O(n^m)$

대표적인 Big-O



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

순차 탐색과 이진 탐색의 비교

- $T(n) = n$ ▶ $O(n)$

순차 탐색의 최악의 경우 시간 복잡도 **순차 탐색의 빅-오**

- $T(n) = log_2 n + 1$ ▶ $O(log\, n)$

이진 탐색의 최악의 경우 시간 복잡도 **이진 탐색의 빅-오**

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

이진 탐색의 연산횟수는 교재의 예제 BSWorstOpCount.c에서 확인한 결과

최악의 경우에 진행이 되는 연산의 횟수

Big-O에 대한 수학적 접근

- Big-O의 수학적 정의
  - 두 개의 함수 f(n)과 g(n)이 주어졌을 때,
  - 모든 $n \geq K$에 대하여 $f(n) \leq Cg(n)$을 만족하는
  - 두 개의 상수 C와 K가 존재하면,
  - f(n)의 빅-오는 O(g(n))이다.

- Big-O의 실질적 의미
  - 예를 들어 $5n^2+100$의 Big-O는 $O(n^2)$이다.
  - 즉, $5n^2+100$의 증가율은 n^2의 상수배를 넘지 못한다.

Big-O에 대한 수학적 접근

- Big-O 판별의 예

"두 개의 함수 $f(n)$과 $g(n)$이 주어졌을 때"

- 두 개의 함수 $f(n)=5n^2+100$, $g(n)=n^2$이 주어졌을 때

"모든 $n \geq K$에 대하여"

- 모든 $n \geq 12$에 대하여,

n의 값이 점차 증가하여 어느 순간 이후부터
f(n) ≤ Cg(n)을 만족하게 하는 두 개의 상수 C와 K가 존재한다면,

"$f(n) \leq Cg(n)$을 만족하는 두 개의 상수 C와 K가 존재하면"

- $5n^2+100 \leq 3500n^2$을 만족하는 $3500(C)$과 $12(K)$가 존재하니,

"$f(n)$의 빅-오는 $O(g(n))$이다."

- $5n^2+100$의 빅-오는 $O(n^2)$이다.

예제 문제

- 다음 코드의 시간복잡도를 Big-O로 표현하면?

```
int a = 0;
for (i = 0; i < N; i++) {
    for (j = N; j > i; j--) {
        a = a + i + j;
    }
}
```

- 정답: $O(N^2)$

예제 문제

- 다음 코드의 시간복잡도를 Big-O로 표현하면?

```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
    a = a + rand();
}
for (j = 0; j < M; j++) {
    b = b + rand();
}
```

- 정답: $O(N + M)$

예제 문제

- 다음 코드의 시간복잡도를 Big-O로 표현하면?

```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
    for (j = 2; j <= n; j = j * 2) {
        k = k + n / 2;
    }
}
```

- 정답: $O(N \log N)$

요약 내용

- 어떤 알고리즘의 효율성은 시간복잡도와 공간복잡도로 나타낼 수 있다.

- 이분 탐색은 배열이 정렬되어 있을 때 선형탐색 보다 더 효율적으로 원하는 값을 탐색할 수 있다.
  - first <= last의 등호
  - first = mid + 1, last = mid – 1

- 빅-오 표기법으로 알고리즘의 실행시간의 점근적인 상한을 표현할 수 있다.
  - 선형 탐색의 시간복잡도는 $O(N)$
  - 이분 탐색의 시간복잡도는 $O(\log N)$