**00. 자료구조 및 알고리즘의 이해**

자료구조 : 데이터를 저장, 조직, 관리하는 방법

알고리즘의 조건 : 입력, 출력, 명백성, 유한성, 유효성

**순서도&의사코드**

**텍스트, 도표, 스크린샷, 평면도이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**

* **알고리즘 순서도 그리는 연습 해보기**

**최악의 경우와 최선의 경우 분석**

Worst case, Average case, Best case

* 알고리즘의 성능을 나타내는 데에는 최악의 경우에 한해서 나타내는 것이 보통이다.

**알고리즘의 수행 시간**

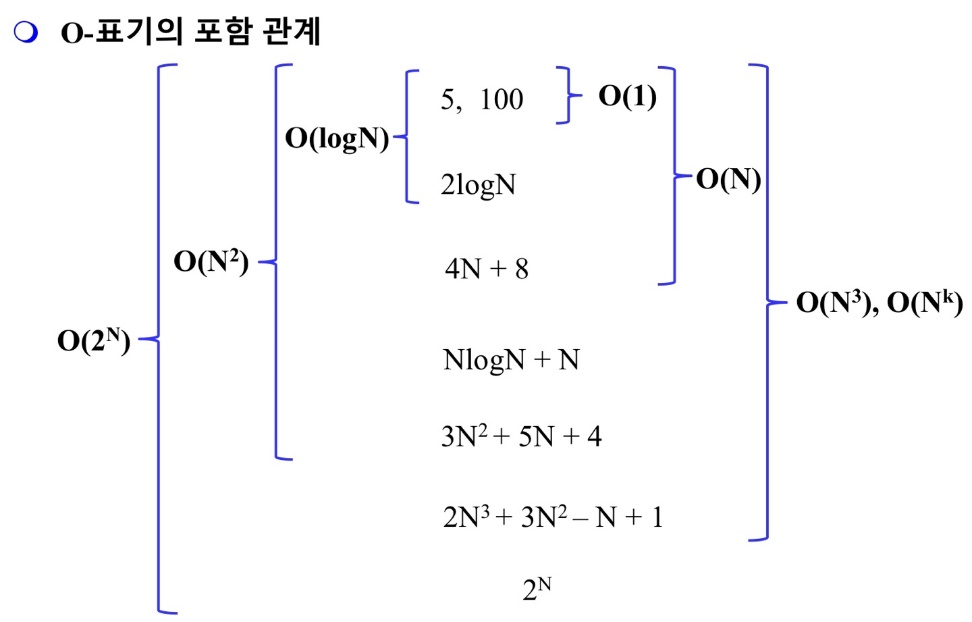
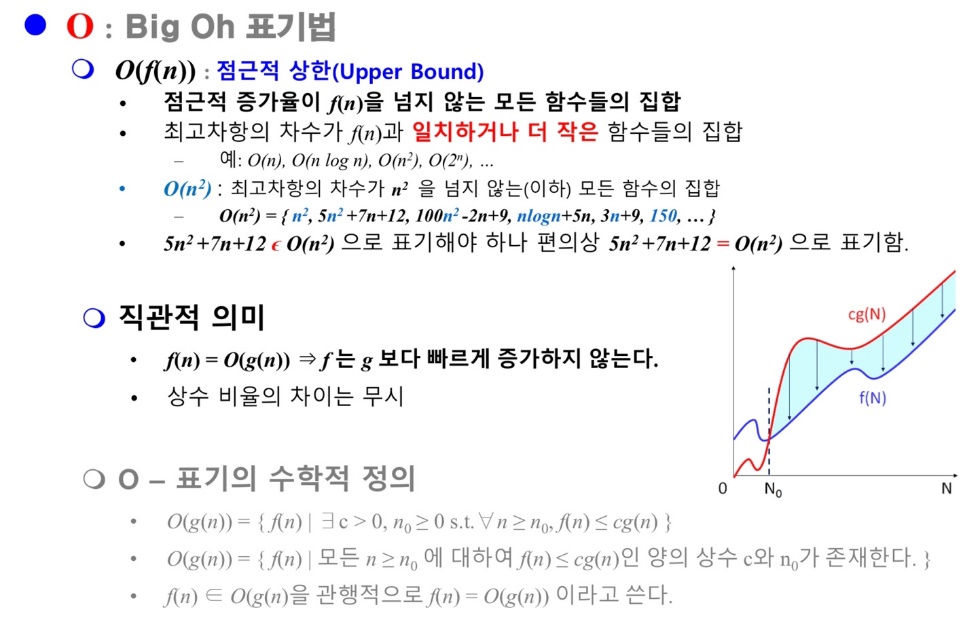
* 자원을 얼마나 효율적으로 사용하는가
* 입력의 크기에 대해 시간이 얼마나 걸리는지 표현함.
  + 정렬 : 정렬하고자 하는 원소의 수
  + 도시간의 최대 거리 : 도시들의 총 수와 도시 간 간선의 총수
  + 팩토리얼 : 팩토리얼을 구하고자 하는 자연수의 크기
* 좌우하는 기준
  + 반복문(for, while)의 반복 횟수
  + 특정 행이 수행되는 횟수
  + 함수의 호출 횟수
  + 정렬의 경우 : 주로 두 수를 비교하는 횟수
* **알고리즘 수행시간 맞춰보기**

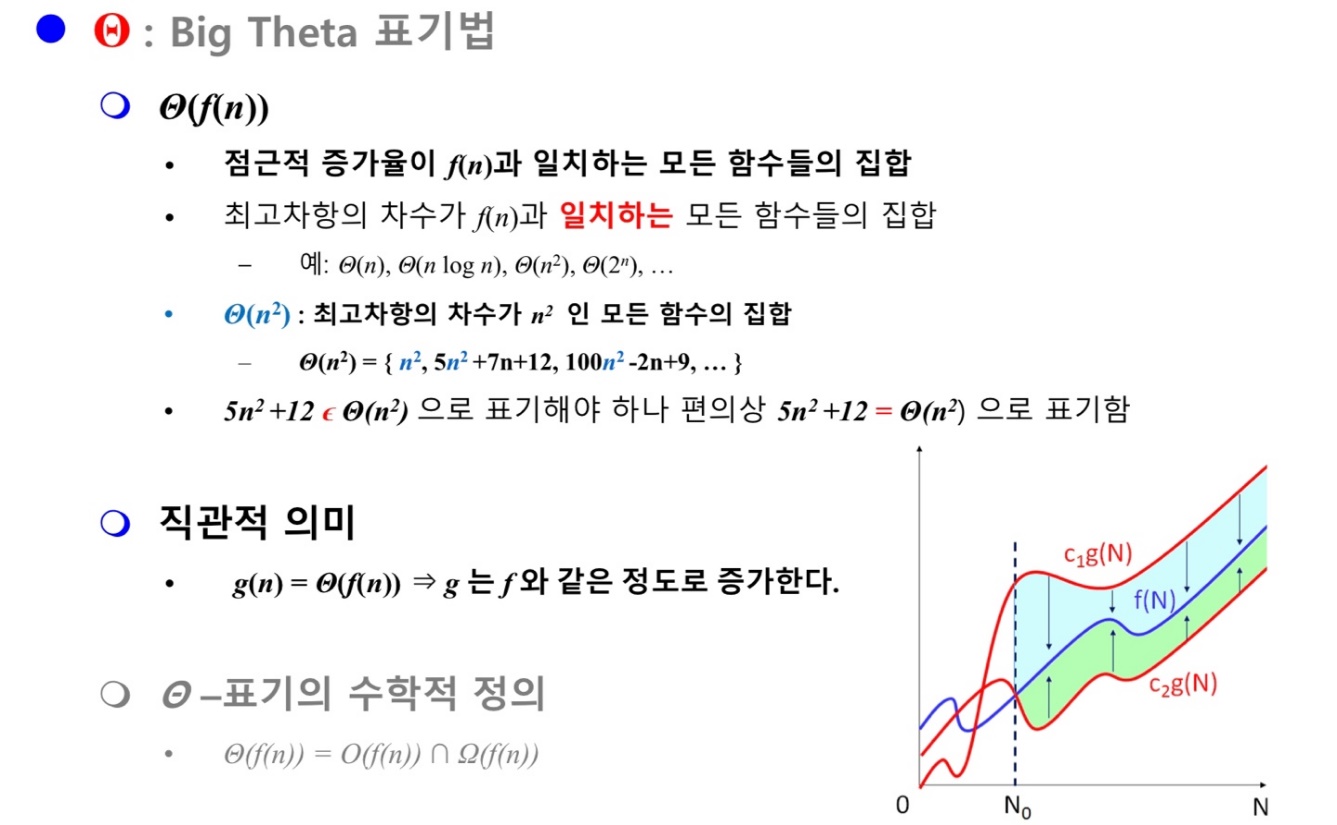
**점근적 분석(Asymptotic Analysis)**

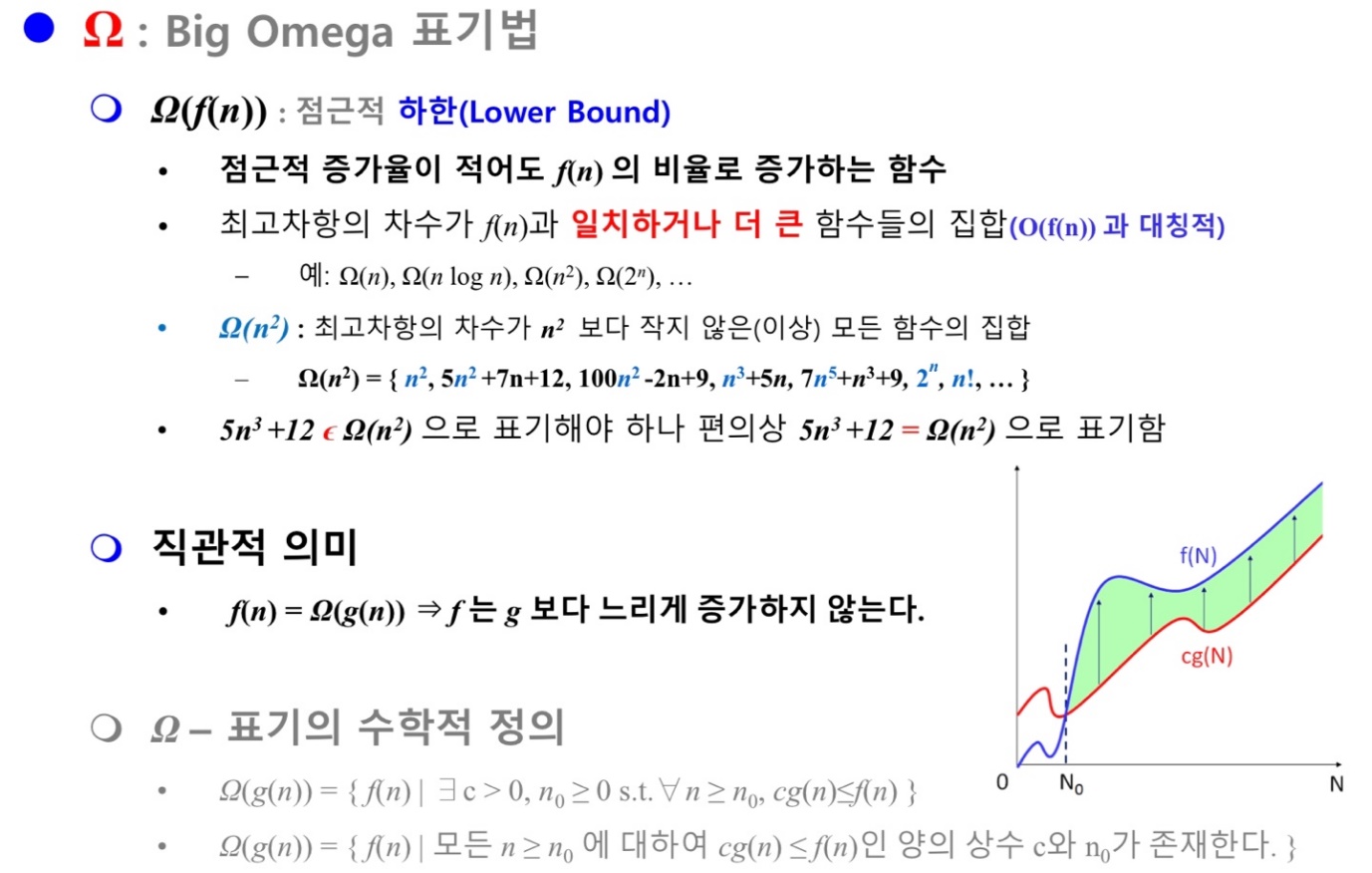
: 알고리즘의 수행시간을 분석할 때는 항상 입력의 크기가 충분히 큰 때에 대해서 분석한다.



* Big oh 표기법 : 최대 f(n)이다.
* Big theta 표기법 : f(n) 이다.
* Big omega 표기법 : 최소 f(n)이다.







**02. 재귀**

**함수 호출시 동작 과정**

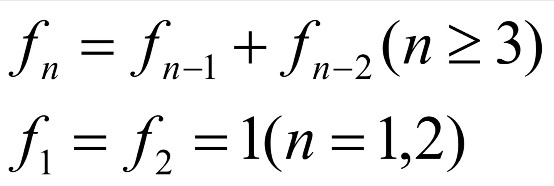
: 스택 구조 - LIFO(Last In First Out)

**재귀 함수(Recursive Fuction)**

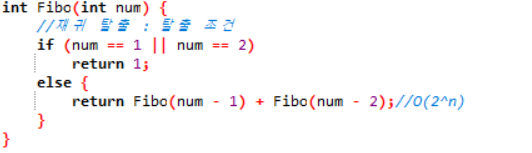
: 자기 자신의 함수를 호출함으로싸, 반복적인 처리를 하는 함수

* 재귀함수 안에서 사용하는 변수는 지역변수
* Call by value
* 반드시 탈출(종료) 조건 명시!!(Stack Overflow 발생 주의)
* **02\_Sum.cpp, 02\_Factorial.cpp**

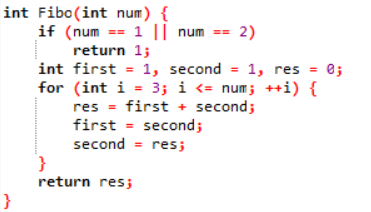
**피보나치 수열**

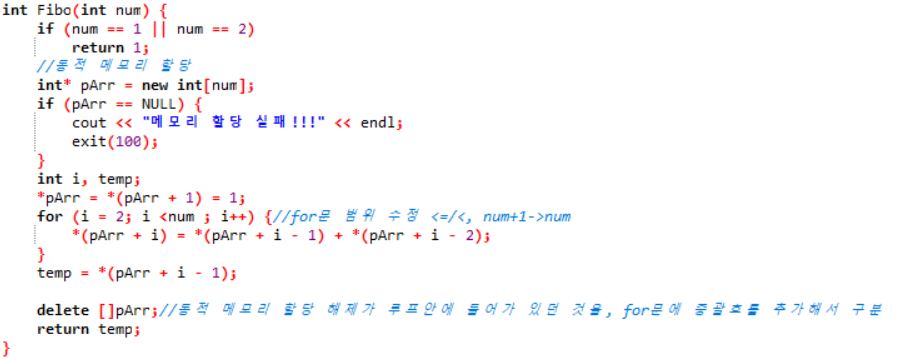


피보나치 수열 : 재귀적 용법 -> O(2^n) -> 문제점 : 중복 호출!! -> 재귀적 알고리즘은 지수함수에 비례하는 시간이 든다.



피보나치 수열 : 동적 프로그래밍 -> Big theta n

 02\_Fibonacci#1.cpp



02\_Fibonacci#2.cpp

**03. 정렬과 탐색**

**정렬(Sort)**

대상 : 레코드/기준 : 정렬 키(sort key) 필드

**알고리즘 성능 비교(점근적 분석)**

: 항상 입력의 크기가 충분히 크다고 가정하고 수행 시간을 분석함.

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

**기초적인 정렬**

**선택 정렬(Selection Sort)**

* 가장 작은 원소를 선택하여 정렬

: 제자리 정렬의 하나로서, 입력 배열(정렬되지 않은 값들) 이외에 다른 추가 메모리를 요구하지 않는 정렬 방법이다. 해당 순서에 원소를 넣을 위치는 이미 정해져 있고, 어떤 원소를 넣을지 선택하는 알고리즘 첫 번째 순서에는 첫 번째 위치에 가장 최솟값을 넣는다. 두 번째 순서에는 두 번째 위치에 남은 값 중에서의 최솟값을 넣는다.

**선택 정렬 수행 과정**

1) 비정렬구역 중에서 최솟값을 찾는다.

2) 그 값을 비정렬구역의 맨앞과 교체한다.

3) 하나의 원소만 남을 때까지 위의 1~3 과정을 반복한다

**메모리 사용 공간**

n개의 원소에 대하여 n개의 메모리 사용

**연산 시간**

(1) Best Case : 자료가 이미 정렬되어 있는 경우, 총 소요시간: n(n-1)/2

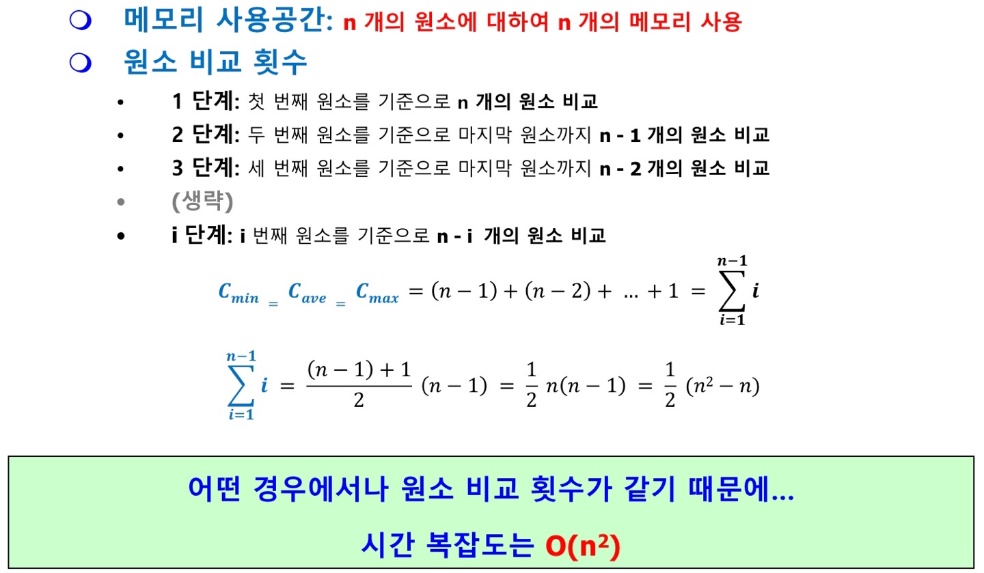
(2) Worst Case : 정렬된 자료에서 최솟값만 맨 뒤에 위치한 경우, 총 소요시간: n(n-1)/2 + (n-1)

데이터 개수 N개 일 때, N\*(N+1)/2번 가량 연산 수행

* 선택정렬의 시간 복잡도 : O(N^2)

텍스트, 폰트, 친필, 스크린샷이(가) 표시된 사진

자동 생성된 설명



수행시간 : (n-1)+…+2+1=O(n^2) <- Worst Case, Average Case

1. for 루프 n-1번 반복
2. 가장 작은 수를 찾기 위한 비교 횟수 : 1,2,…,n-1
3. 교환은 상수시간 작업

**버블 정렬(Bubble sort)**

* 인접합 두 원소의 크기를 비교하고, 작은 원소를 왼쪽으로 옮긴다.

: 거품이 수면 위로 올라오는 듯한 모습을 닮아 붙여진 이름, 1회전을 수행하고 나면 가장 큰 자료가 맨 뒤 로 이동하므로 2회전에서는 맨 끝에 있는 자료는 정렬에서 제외되고. 2회전을 수행하고 나면 끝에서 두 번째 자료까지는 정렬에서 제외된다. 이렇게 정렬을 1회전 수행할 때마다 비정렬구역에서 제외되는 데이 터가 하나씩 늘어난다.

**버블 정렬 수행 과정**

1) 비 정렬구역의 맨 칸 원소와 한칸 앞의 원소를 비교한다.

2) 맨 뒤 칸의 원소가 더 작을 경우 둘의 위치를 변경한다.

3) 비 정렬구역의 맨 앞 경계를 한 칸 뒤로 미루다

4) 위 과정을 반복하다.

**메모리 사용 공간**

n개의 원소에 대하여 n개의 메모리 사용

**연산 시간**

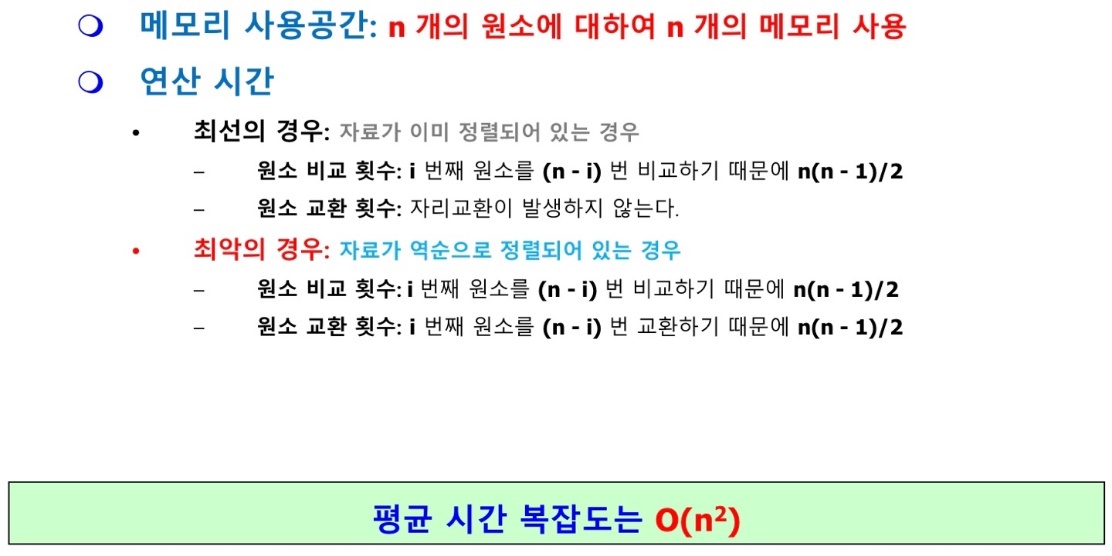
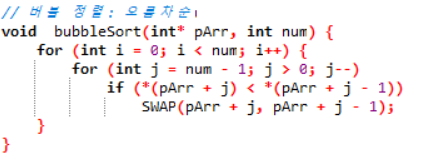
(1) Best Case : 자료가 이미 정렬되어 있는 경우, 총 소요시간: n(n-1)/2

2) Worst Case : 자료가 역순으로 정렬되어 있는 경우, 종 소요시간: n(n-1)

구현이 매우 간단하나, 구조 자체의 비효율성, 그리고 일반적으로 자료의 SWAP이 자료를 이동시키는 것 보다 더 복잡하기 때문에 단수성에도 불구하고 거의 쓰이지 않는다

제일 무식한 정렬

* 버블 정렬의 시간 복잡도 : O(N^2)



수행시간 : (n-1)+…+2+1=O(n^2) <- Worst Case, Average Case

1. for 루프 n-1번 반복
2. 가장 큰 수를 찾기 위한 비교 횟수 : n-1, n-2,…,2,1
3. 교환은 상수시간 작업

**삽입 정렬(Insertion Sort)**

* 정렬되지 않은 원소를 정렬된 영역의 적당한 위치에 넣어주는 정렬

**삽입 정렬 수행 과정**

1) 각 단계에서 비 정렬 구역의 첫 번째 원소를 선택한다.

2) 첫 번째 원소를 정렬 구역의 원소와 비교하여 적당한 자리에 삽입한다.

**메모리 사용 공간**

n개의 원소에 대하여 n개의 메모리 사용

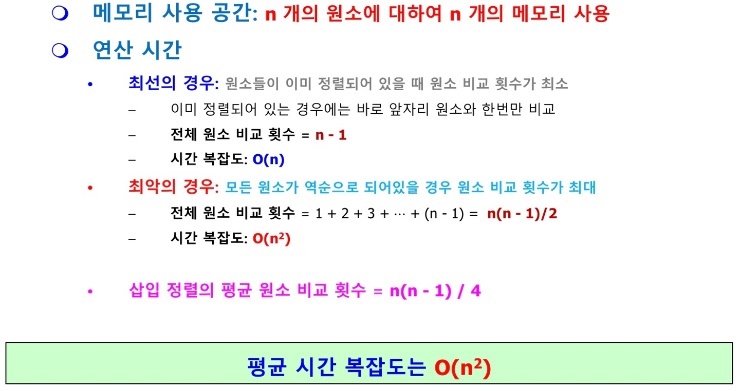
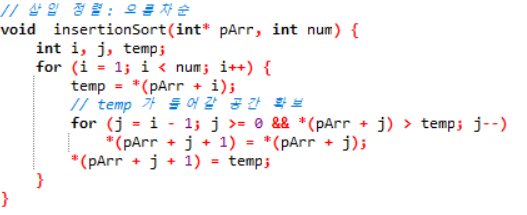
**연산 시간**

1) Best Case : 자료가 이미 정렬되어 있는 경우 : n-1

2) Worst Case : 자료가 역순으로 정렬되어 있는 경우 : n(n-1)/2

삽입 정렬은 기본적으로 ‘정렬이 되어있다고 가정’을 한다는 점에서 특정한 경우에 따라 어떠한 알고리즘보다도 빠른 속도를 자랑함 : 제일 좋은 성능!!

* 삽입 정렬의 시간 복잡도 : O(N^2)



수행시간 :

1. for루프는 n-1번 반복
2. 삽입은 최악의 경우 i-1회 바교

Worst case : 1+2+…+(n-2)+(n-1) = O(n^2)

Average case : 1/2(1+2+…+(n-2)+(n-1))=O(n^2)

Best case : n-1=O(n)

**기초적인 검색 알고리즘**

**검색(Search)**

: 레코드의 집합에서 주어진 키를 지닌 레코드를 찾는 작업 탐색

주어진 키 값 : 목표 키(target key) 또는 검색 키(searc key)

**순차검색(Sequential Search)**

* 선형검색/순서가 없는 리스트일 때 사용/리스트가 작거나, 가끔 한번씩 검색할 경우에만 사용
* 목표치를 찾기 위해 리스트의 처음부터 탐색을 시작해서, 목표치를 찾거나 리스트에 목표치가 없다는 것이 밝혀질 때까지 검색을 계속한다.



**이진검색(Binary Search)**

: 중간 값을 이용하여 찾고자 하는 값인 Key값을 찾아내는 알고리즘

* 이진검색은 배열이 정렬되어 있을 때 효율적인 알고리즘(순차검색은 매우 느리다.)

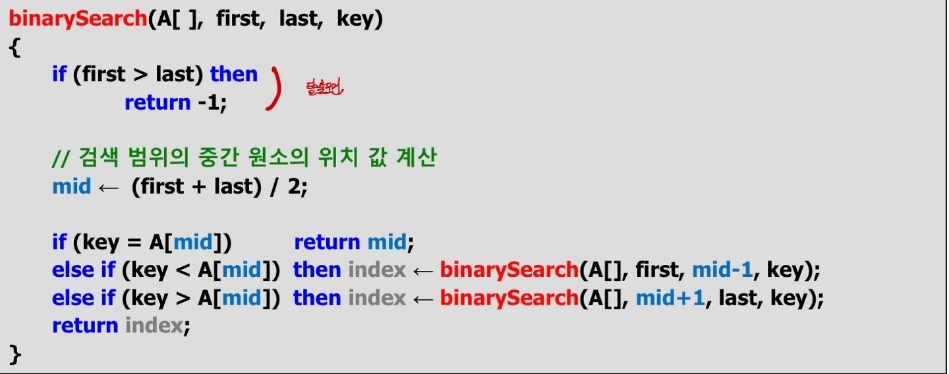
**조건**

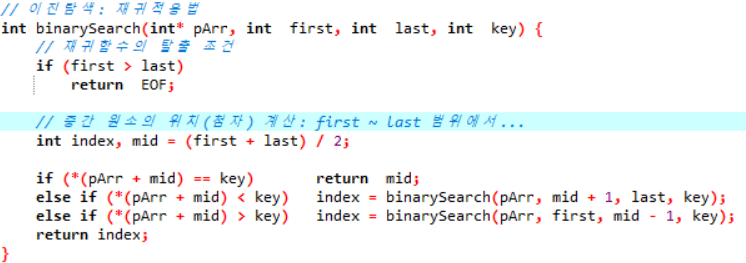
검색할 데이터 : 정렬된 상태

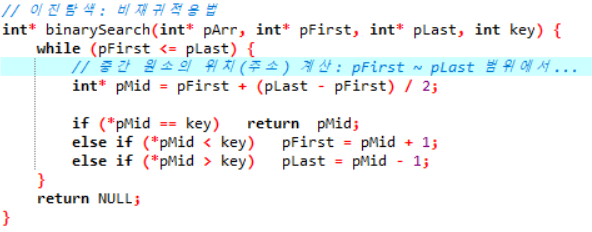
주어진 데이터 : 유일한 키 값을 지님.

**과정**

1. 탐색할 범위 경계값의 중간 값을 구한다.
2. 중간값과 key값을 비교한다.
   1. 중간값>key값 : 탐색범위를 첫번째값~중간값-1로 바꾼다.
   2. 중간값<key값 : 탐색범위를 중간값+!~마지막값으로 바꾼다.







**고급 정렬 알고리즘**

**쉘 정렬(Shell sort)** <- 삽입정렬의 변형

: 일정한 간격(interval)으로 데이터들끼리 부분집합을 구성하고, 각 부분집합에 있는 원소들에 대해서 삽입 정렬을 수행한다.**(삽입정렬 + interval = 쉘 정렬)**

* 삽입정렬을 수행하는 것보다 부분집합으로 나누어 정렬하면, 비교와 교환 연산을 감소
* 7-정렬, 4-정렬등의 용어가 사용됨.

**쉘 정렬 수행 과정**

1) 첫 번째 원소와 첫 번째 원소로부터 interval만큼 간격이 있는 원소들을 부분 집합에 포함

2) 부분 집합에 포함된 원소들을 삽입 정렬 수행

3) 두 번째 원소와 두 번째 원소로부터 h의 배수만큼 간격이 있는 원소들을 부분 집합에 포함

4) 부분 집합에 포함된 원소들을 삽입 정렬 수행

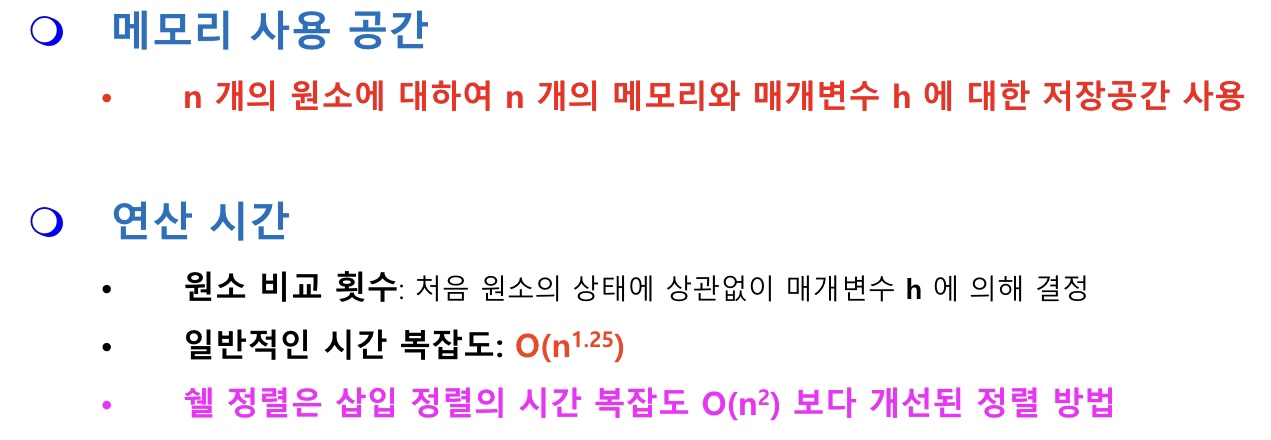
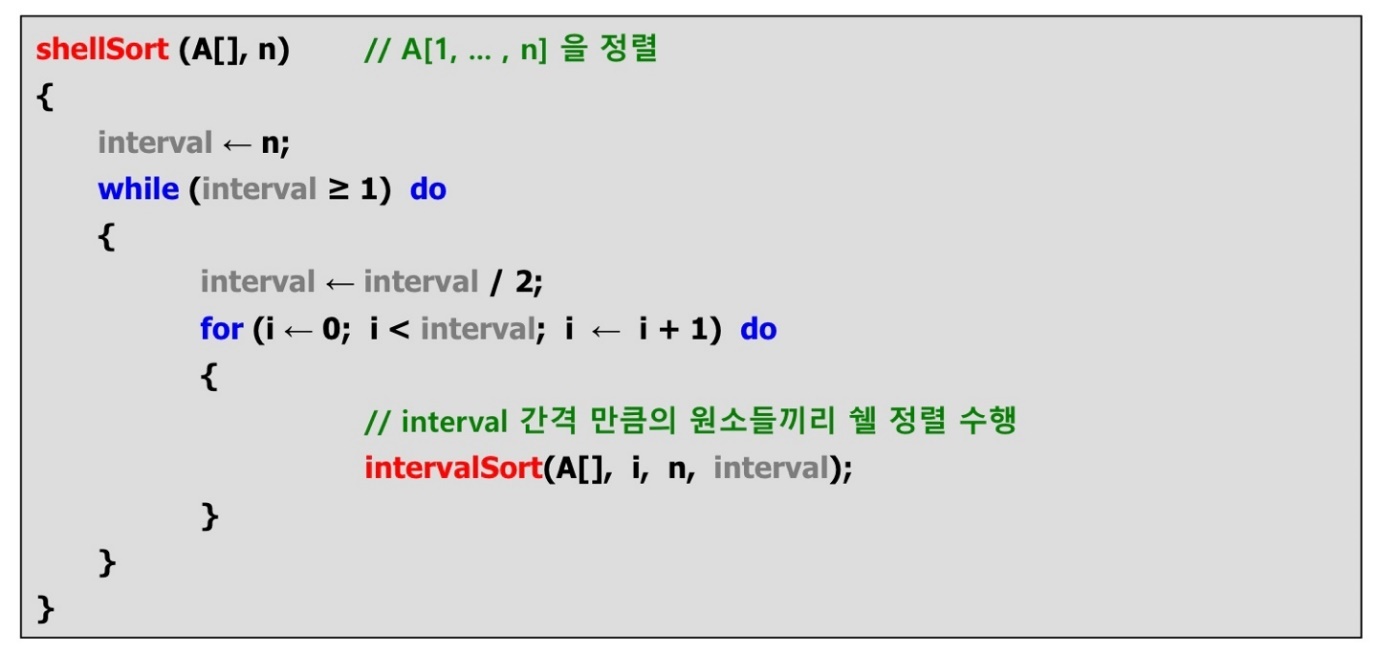
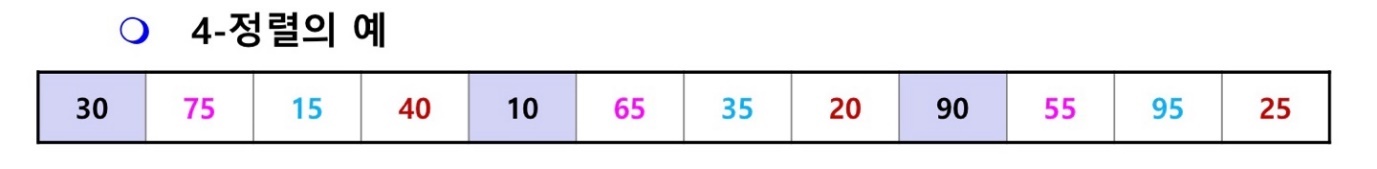
5) 위 과정을 반복

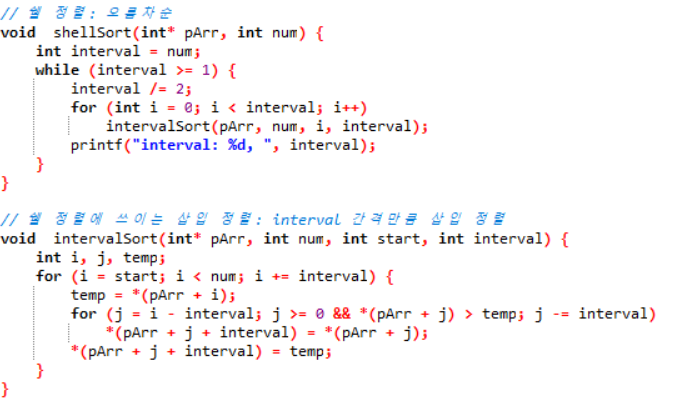
**메모리 사용 공간**

n개의 원소에 대하여 n개의 메모리와 매개변수 h에 대한 저장공간 사용

**연산 시간**

: 원소의 상태와 상관없이 간격에 의해 결정





**퀵 정렬(Quick Sort**)\_재귀적

* 기준값을 중심으로 부분집합을 나누고, 각각을 정렬함.

: 기준 값을 중심으로 작은 원소들은 왼쪽 부분집합으로, 큰 원소들은 오른쪽 부분 집합으로 분할하여 정렬 하는 방식의 정렬 알고리즘

**퀵 정렬 수행 과정**

(1) Left index를 리스트 가장 왼쪽에, Riaht index를 리스트 가장 오른쪽에 위치시킨다.

2) Left 에 위치하 원소가 pivot보다 큰 값이 나올 때까지 Left를 리스트의 오른쪽 방향으로 이동시킨다

Richt에 위치한 원소가 pivot보다 작거나 같은 값이 나올 때까지 리스트의 왼쪽 방향으로 이동시킨다.

3) Left와 Right가 교차하지 않았다면, 두 인덱스에 위치한 원소를 교환한 후, 2번 과정으로 돌아간다.

4) Left 와 Richt가 교차했다면, pivot과 Richt에 위치한 원소를 교환한다. 교환한 후의 bivot의 위치를 기 준으로 왼쪽과 오른쪽의 부분집합을 구분하고 두 부부 집합에 대하여 퀵 정렬 알고리즘을 재귀 호출하다.

**메모리 사용 공간**

n개의 원소에 대하여 n개의 메모리 사용

**연산 시간**

1) Best Case : 왼쪽 부분집합과 오른쪽 부분집합이 정확히 이등분 되는 경우: O(nlogn)

2) Worst Case : 부분집합이 1개와 n-1개로 치우쳐 분할되는 경우가 반복되는 경우 : O(n^2)

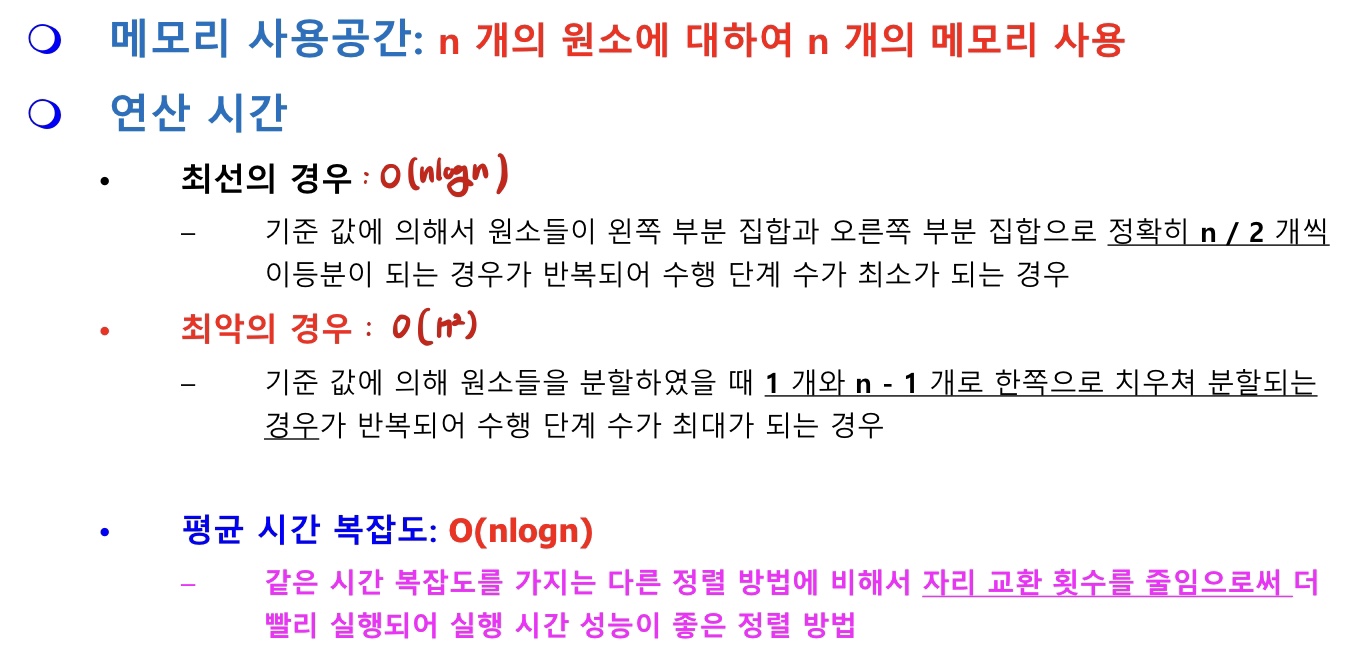
**기준 값 : Pivot**

* 왼쪽 부분 집합 : 기준 값보다 작은 원소들을 이동
* 오른쪽 부분 집합 : 기준 값보다 큰 원소들을 이동

**‘분할&정복’ 알고리즘**

* 분할(정렬할 자료들을 기준 값을 중심으로 두 개의 부분 집합으로 분할)
* 정복(부분 집합의 원소들 중에서 기준 값을 중심으로 왼쪽, 오른쪽 부분집합으로 정렬/부분 집합의 크기가 1이하로 충분히 작지 않으면 순환호출을 이용하여 다시 분할)

****

****

**텍스트, 폰트, 스크린샷, 번호이(가) 표시된 사진

자동 생성된 설명**

**병합 정렬(Merge Sort)**\_재귀적

* 일단 정확히 반으로 나누고 나중에 정렬

: 하나의 배열을 두 개의 균등한 크기로 분할하고 분할된 부분 집합을 정렬한 다음, 두 개의 정렬된 부분 집합를 합하여 전체가 정렬되게 하는 방법이다

**병합 정렬 수행 과정**

1) 정렬되지 않은 리스트를 원소 1개의 부분리스트로 분할한다.

2) 부분 리스트가 하나만 남을 때까지 반복 병합하면서 정렬된 부분 리스트를 생성한다.

3) 마지막 하나 남은 부분 리스트가 정렬된 리스트가 된다.

**메모리 사용 공간**

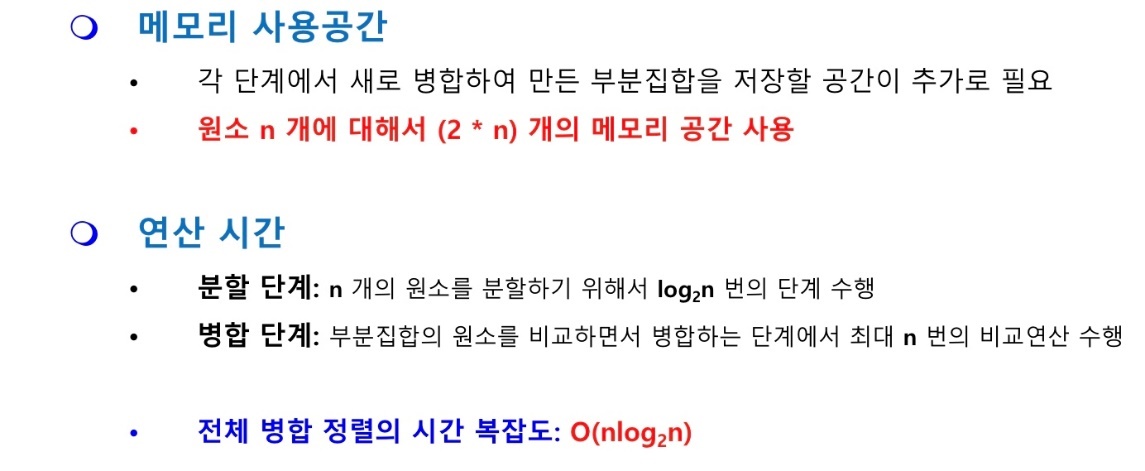
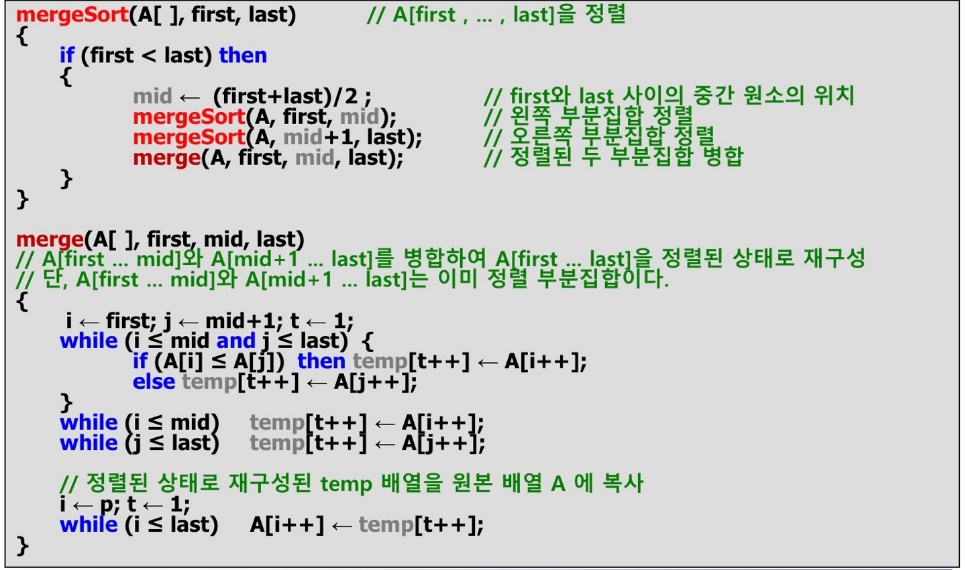
n개의 원소에 대하여 (2tn)개의 메모리 공간이 사용된다.

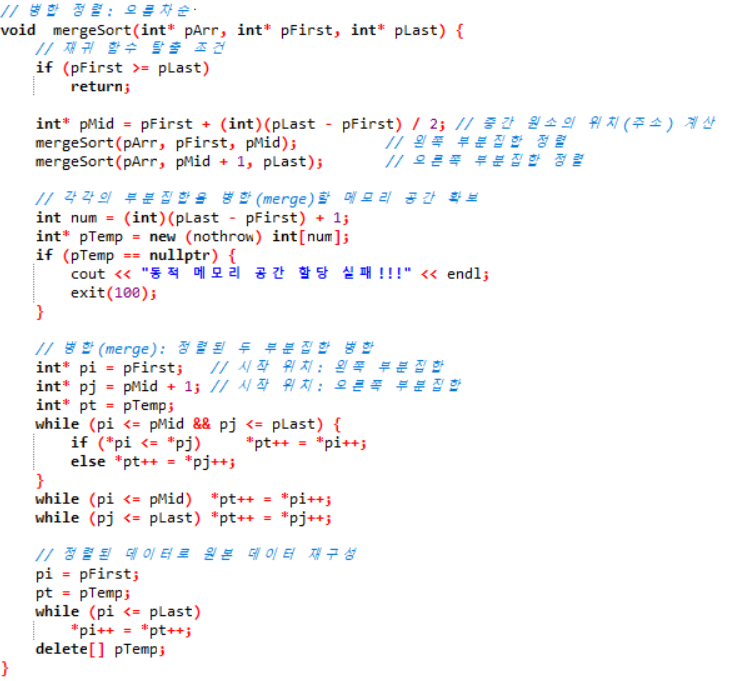
**병합 정렬 방법의 종류**

* 2-way 병합 : 2개의 정렬된 자료의 집합을 결합하여 하나의 집합으로 만드는 방법
  + 분할 : 입력자료를 같은 크기의 부분집합 2개로 분할함.
  + 정복 : 부분집합의 원소들을 정렬함.

\* 만약 부분집합의 크기가 충분히 작지 않으면, 순환호출을 이용하여 다시 분할 정복 기법을 적용한다.

* + 결합 : 정렬된 부분집합들을 하나의 집합으로 통합.
* N-way 병합 : n개의 정렬된 자료의 집합을 결합하여 하나의 집합으로 만드는 방법



****

**특수 정렬 알고리즘**

**비교정렬**

두 원소를 비교하는 정렬의 하한선 : Big theta(nlogn) -> “Worst Case의 시간 복잡도가 O(nlogn)보다 더 빠를 수는 없는가? -> 원소들이 특수한 성질을 만족할 경우 O(n)정렬도 가능함.

* **계수 정렬(Counting Sort)** : 원소들의 크기가 모드 -O(n)~O(n) 범위에 있을 때
* **기수 정렬(Radix Sort)** : 원소들이 모두 k이하의 자리수를 가졌을 때
* **버킷 정렬(Bucket Sort)** : 원소들이 균등 분포를 이룰 때

**계수 정렬(Counting Sort)**

: 각 숫자의 개수를 센 후, 작은 수부터 갯수만큼 출력

**전제조건**

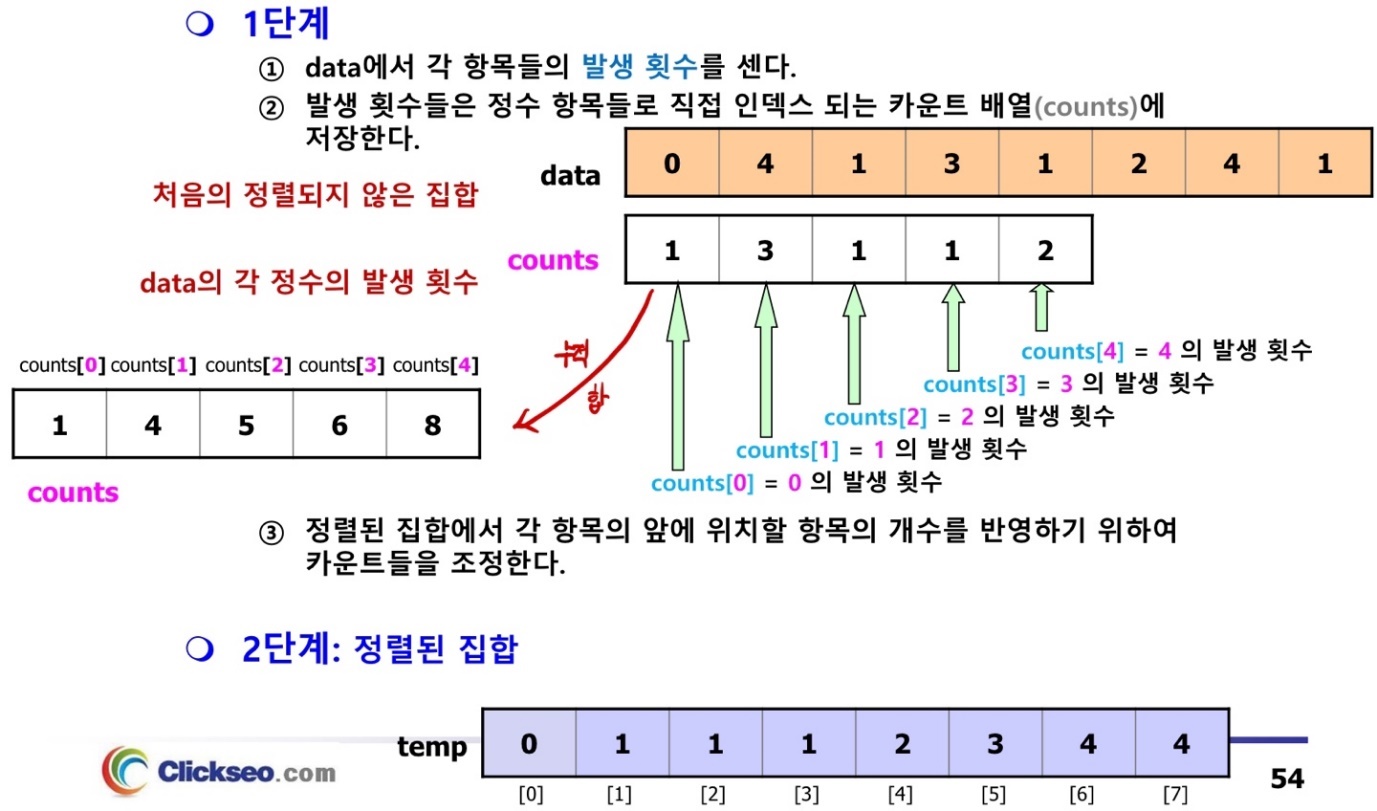
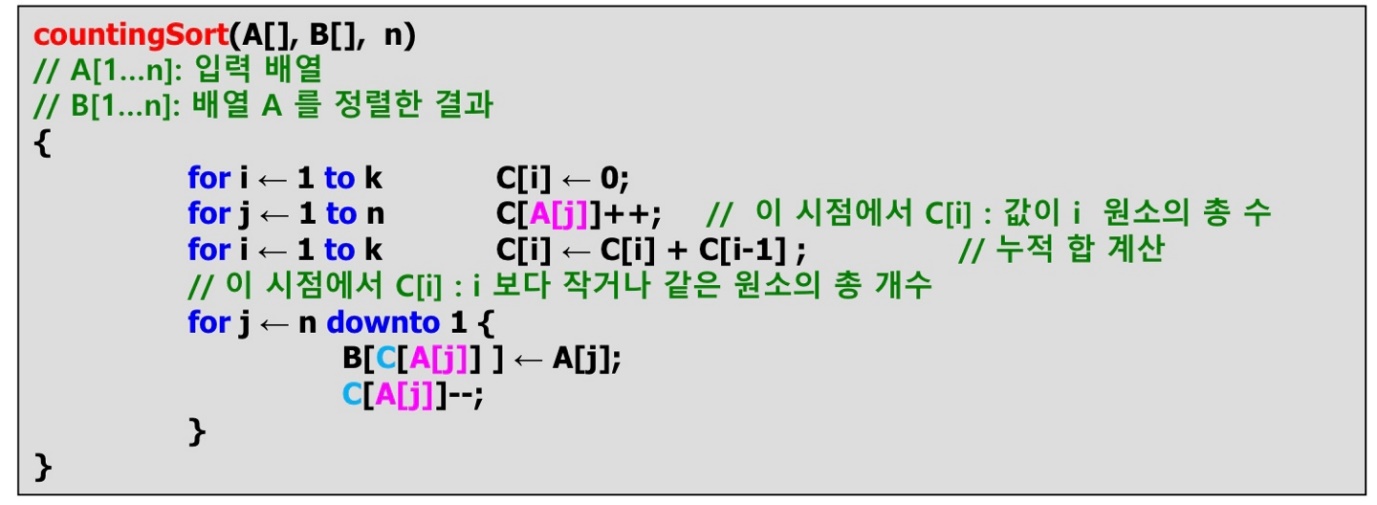
* 정수로 표현가능한 배열만 정렬할 수 있다.
* 카운트들을 위한 충분한 공간을 할당하려면 배열 내의 최댓값을 알아야 한다.

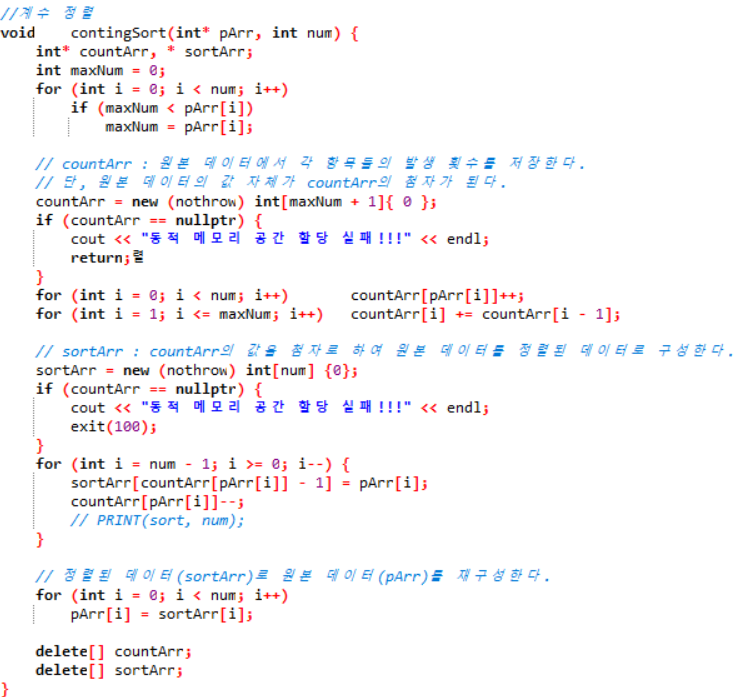
**과정**

1. 각 data의 개수를 센다.
2. 카운트 배열에 개수를 저장한다.

* 배열의 인덱스 : dat 값
* 배열의 데이터 : data의 개수

1. 누적합 배열 생성
2. for문으로 배열 정리, 출력





**기수 정렬(Radix Sort)**

: 입력이 모두 k 이하의 자리수를 가진 특수한 경우

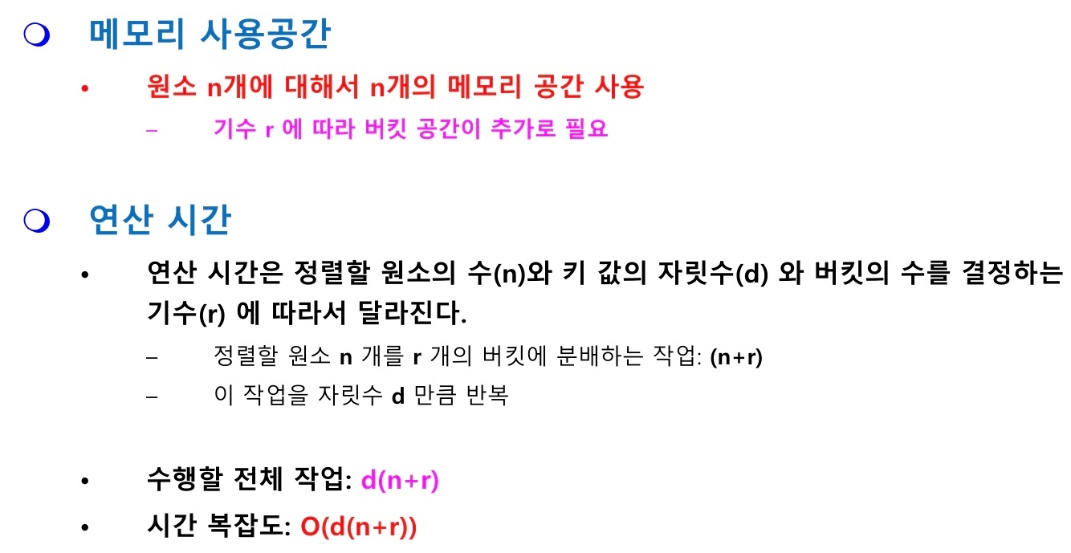
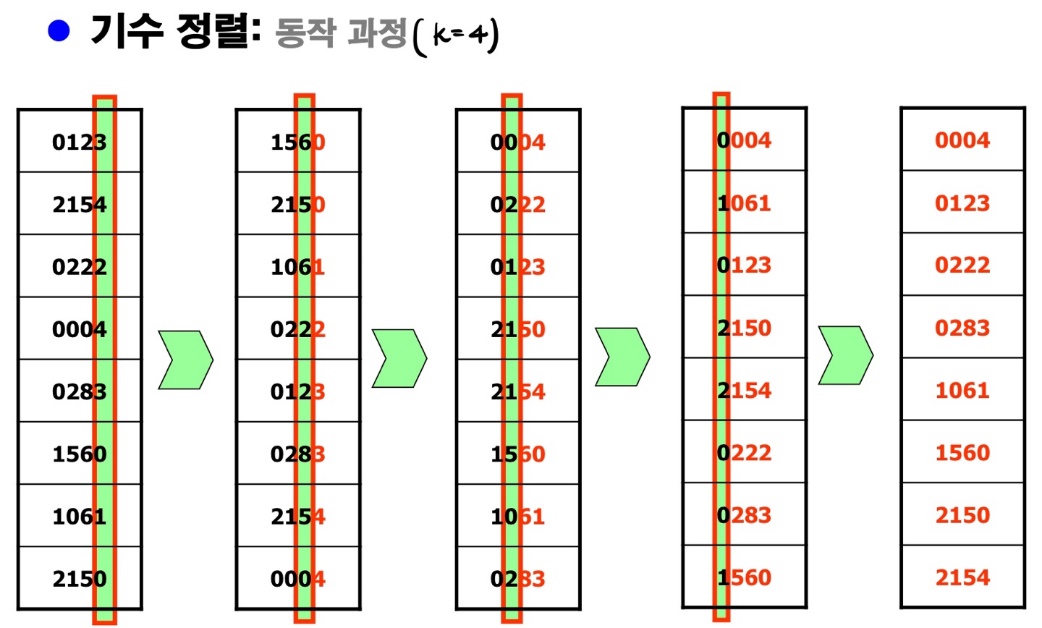
* 시간 복잡도 : O(n)

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

**안전성 정렬(Stable Sort)**

: 같은 값을 가진 원소들은 정렬 후에도 원래의 순서가 유지되는 성질을 가진 정렬을 일컫는다.



**버킷 정렬(Bucket Sort)**

: 원소들이 균등 분포를 하는 [0,1) 범위의 실수의 경우

