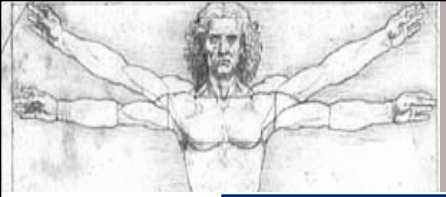
KNU KYUNGPOOK NATIONAL UNIVERSITY

Lecture #2: Sorting & Time Complexity

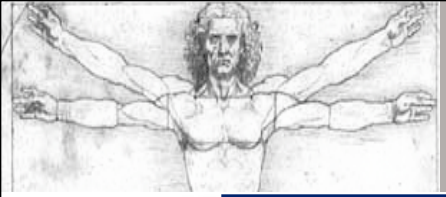
School of Computer Science and Engineering
Kyungpook National University (KNU)

Woo-Jeoung Nam



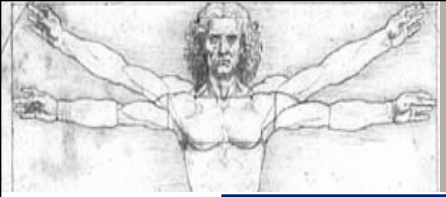
Agenda

- 시간 복잡도(Time complexity)
  - 정렬 알고리즘
  - 시간 복잡도와 점근적 표기법



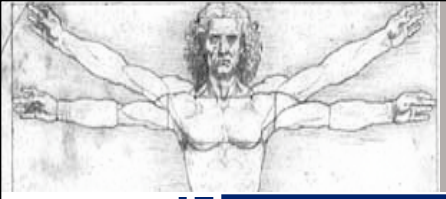
잠시 그전에...

- https://sites.google.com/view/prmi-knu/home



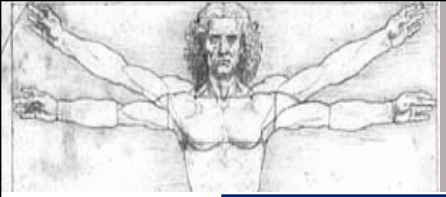
정렬 알고리즘

- 정렬 알고리즘
  - 어떤 데이터들이 주어졌을 때 이를 정해진 순서대로 나열하는 문제
- 예시
  - 검색과 색인화
    - 성적을 정리하고 검색해야 하는 경우, 성적이 오름차순으로 정렬된 데이터를 사용하면, 특정 성적 범위 내에서 빠르게 검색 가능
  - 데이터베이스 관리
    - 대규모 이커머스 웹사이트에서 수많은 제품 정보를 데이터베이스에 저장해야 하는 경우, 제품 이름, 가격, 브랜드 등으로 정렬된 데이터를 사용하면, 사용자가 원하는 조건에 따라 제품을 빠르게 검색
  - 알고리즘 설계
    - 이진 검색 알고리즘은 정렬된 배열에서만 사용 가능. 따라서, 정렬 알고리즘이 필요하며, 이진 검색 알고리즘은 이러한 정렬 알고리즘과 함께 사용
  - 문제 해결
    - 수백만 개의 숫자가 있는 배열에서 중앙값을 찾아야 하는 경우, 배열을 정렬한 후 중간 위치의 값을 가져오면 됨. 또한, 최빈값을 찾는 경우, 데이터를 정렬하여 가장 많이 출현한 값을 빠르게 찾을 수 있음



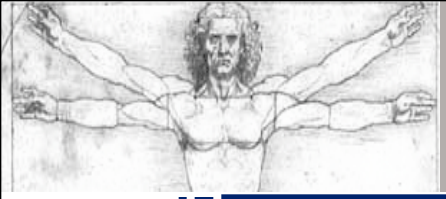
정렬 알고리즘

Name	Best	Avg	Worst	Run-time(정수 60,000개) 단위: sec
삽입정렬	n	n^2	n^2	7.438
선택정렬	n^2	n^2	n^2	10.842
버블정렬	n^2	n^2	n^2	22.894
셸 정렬	n	$n^{1.5}$	n^2	0.056
퀵 정렬	$n\log_2 n$	$n\log_2 n$	n^2	0.014
힙 정렬	$n\log_2 n$	$n\log_2 n$	$n\log_2 n$	0.034
병합정렬	$n\log_2 n$	$n\log_2 n$	$n\log_2 n$	0.026



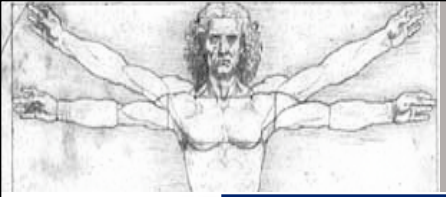
정렬 알고리즘

- 정렬 알고리즘의 종류
  - 삽입 정렬(Insertion Sort)
    - 배열을 순회하며 각 요소를 적절한 위치에 삽입하는 작업을 반복
    - 시간 복잡도: O(n^2)
    - 장점: 구현이 쉽고, 대부분의 요소가 이미 정렬되어 있을 경우에는 매우 효율적
    - 단점: 대규모 데이터에서는 성능이 저하됨
  - 선택 정렬(Selection Sort)
    - 배열을 전체를 순회하며 최솟값을 찾아 맨 앞으로 옮기는 작업을 반복
    - 시간 복잡도: O(n^2)
    - 장점: 구현이 쉬우며, 작은 배열에서는 빠르게 동작할 수 있음
    - 단점: 대규모 데이터에서는 성능이 저하됨
  - 버블 정렬(Bubble Sort)
    - 배열을 순회하며 인접한 두 요소를 비교하여 크기순으로 정렬하는 작업을 반복
    - 시간 복잡도: O(n^2)
    - 장점: 구현이 쉬우며, 구현 방법이 직관적임
    - 단점: 대규모 데이터에서는 성능이 저하됨



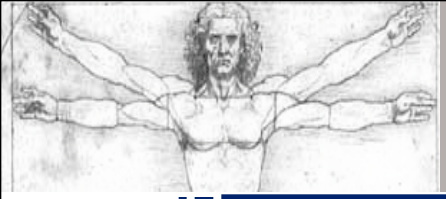
정렬 알고리즘

- 퀵 정렬(Quick Sort)
  - 배열을 분할하여 각각을 정렬하는 작업을 반복
  - 분할은 피벗(pivot) 값을 중심으로 작은 값은 왼쪽, 큰 값은 오른쪽으로 배치하여 수행
  - 시간 복잡도: 평균 O(nlogn), 최악 O(n^2)
  - 장점: 대부분의 경우 빠른 속도를 보이며, 메모리 사용량이 적음
  - 단점: 최악의 경우 시간 복잡도가 O(n^2)이 될 수 있음
- 병합 정렬(Merge Sort)
  - 배열을 분할하여 각각을 정렬한 후 병합하는 작업을 반복
  - 분할은 배열의 중간을 기준으로 수행
  - 시간 복잡도: O(nlogn)
  - 장점: 최악의 경우에도 시간 복잡도가 일정하며, 안정적으로 동작함
  - 단점: 메모리 사용량이 많음
- 힙 정렬(Heap Sort)
  - 힙 자료구조를 이용하여 정렬하는 알고리즘, 완전 이진 트리(Complete Binary Tree)의 일종으로, 모든 노드의 값이 그 자식 노드의 값보다 큰(혹은 작은) 트리
  - 시간 복잡도: O(nlogn)
  - 장점: 평균 및 최악의 경우 시간 복잡도가 O(nlogn)로 일정하고 대용량 데이터 처리에 효율적
  - 단점: 메모리 사용량이 크고 구현이 어려움



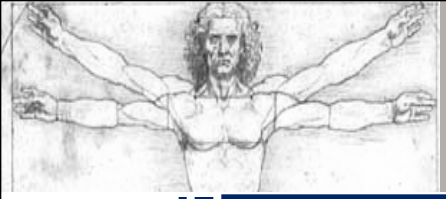
정렬 알고리즘 – 삽입 정렬

- 배열의 모든 요소를 순회하며, 각 요소를 앞의 요소들과 비교하여 적절한 위치에 삽입
  - 1. 배열의 두 번째 요소부터 시작하여, 앞의 요소와 비교
  - 2. 앞의 요소보다 작을 경우, 앞의 요소를 한 칸씩 오른쪽으로 이동
  - 3. 적절한 위치를 찾았을 경우, 현재 요소를 해당 위치에 삽입
  - 4. 1 ~ 3 단계를 배열의 마지막 요소까지 반복
- 삽입 정렬의 귀납적 증명
  - (1) Inductive Hypothesis:
    - 배열 A[:i+1]은 i번째 단계가 종료되면, 정렬이 된 상태이다.
  - (2) Base case:
    - 배열 A[:1]은 0번째 단계에서 이미 정렬이 되어 있다. (당연하다 - 원소 1개를 가진 배열)
  - (3) Inductive step:
    - 어떤 0 < k < n에 대해서,
      귀납적 가설이 i=k-1에 대해 성립하면, i=k에 대해서도 성립한다.
    - 즉, 만약 A[:k]가 k-1번째 단계에서 정렬이 되면, A[:k+1]은 k번째 단계에서 정렬
  - (4) Conclusion:
    - 귀납적 가설은 i = 0, 1, …, n-1에 대해서 성립
    - 특히, i = n-1에 대해서 성립한다.
    - n-1번째 단계가 종료되면 (즉, 알고리즘의 동작이 끝날 때), 배열 A[:n] = A는 정렬

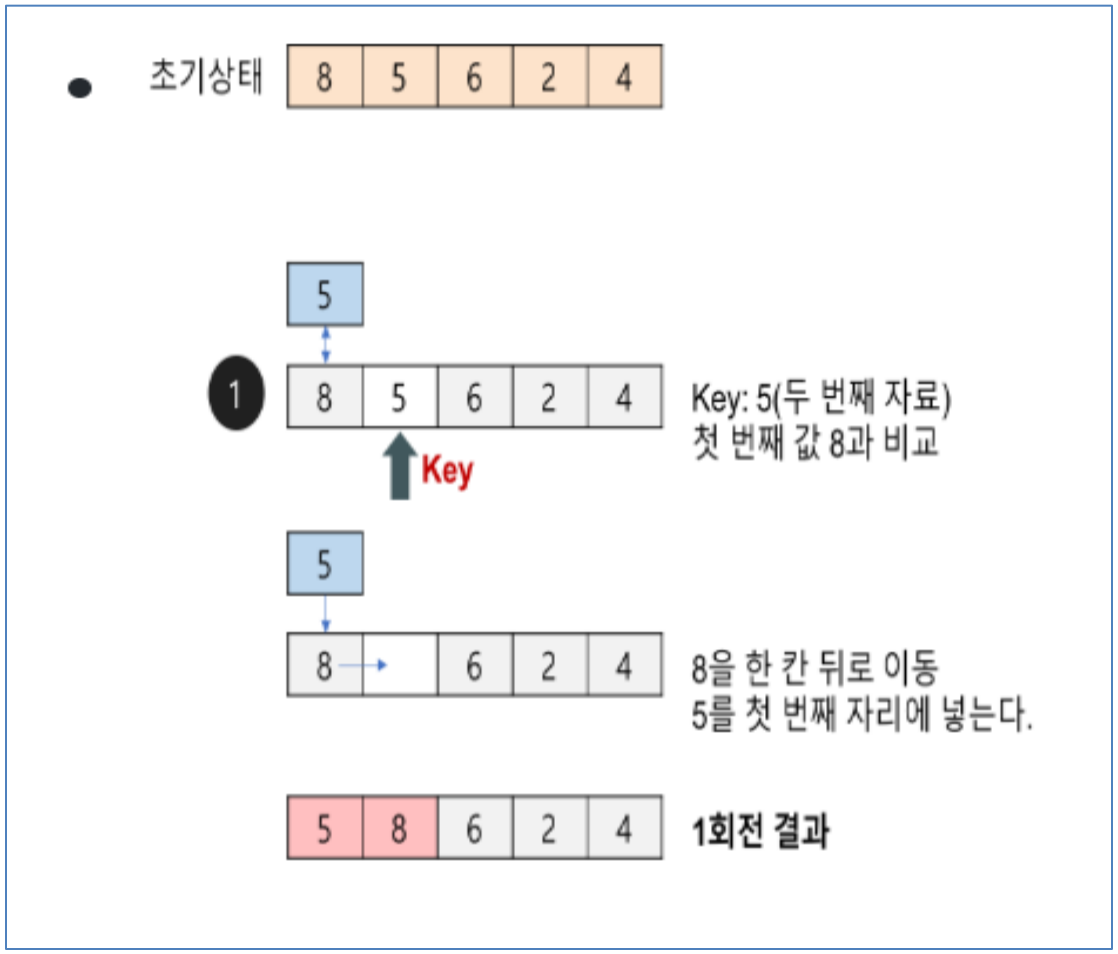


정렬 알고리즘 – 삽입 정렬

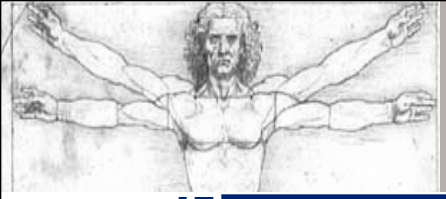
- 귀납적 증명(Induction Proof)
  - 수학적 증명의 하나로, 일반적인 경우가 성립함을 증명하기 위해, 초기 단계가 성립하고, 귀납 단계에서도 성립함을 보여주는 방법
    - 초기 단계: 증명하려는 명제가 n=1, 2, 3 등의 작은 경우에서 성립함을 보임
    - 귀납 가정: n=k인 경우, 명제가 성립한다고 가정함
    - 귀납 단계: n=k+1인 경우, 명제가 성립함을 보임
    - 결론: 초기 단계와 귀납 단계를 모두 만족하므로, 모든 경우에서 명제가 성립함을 증명
  - 예시: "모든 자연수 n에 대해, 1부터 n까지의 합은 n(n+1)/2와 같다."
    - 초기 단계: n=1인 경우, 1 = 1(1+1)/2 이므로, 명제가 성립함
    - 귀납 가정: n=k인 경우, 1부터 k까지의 합은 k(k+1)/2와 같다고 가정
    - 귀납 단계: n=k+1인 경우, 1부터 k+1까지의 합을 구하면 (k+1) + k(k+1)/2 = (k+1)(k+2)/2
      - 따라서, 명제가 성립
    - 결론: 초기 단계와 귀납 단계를 모두 만족하므로, 명제가 모든 자연수 n에 대해 성립함을 증명



정렬 알고리즘 – 삽입 정렬

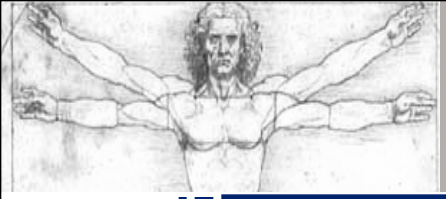


Source: https://gmlwjd9405.github.io/

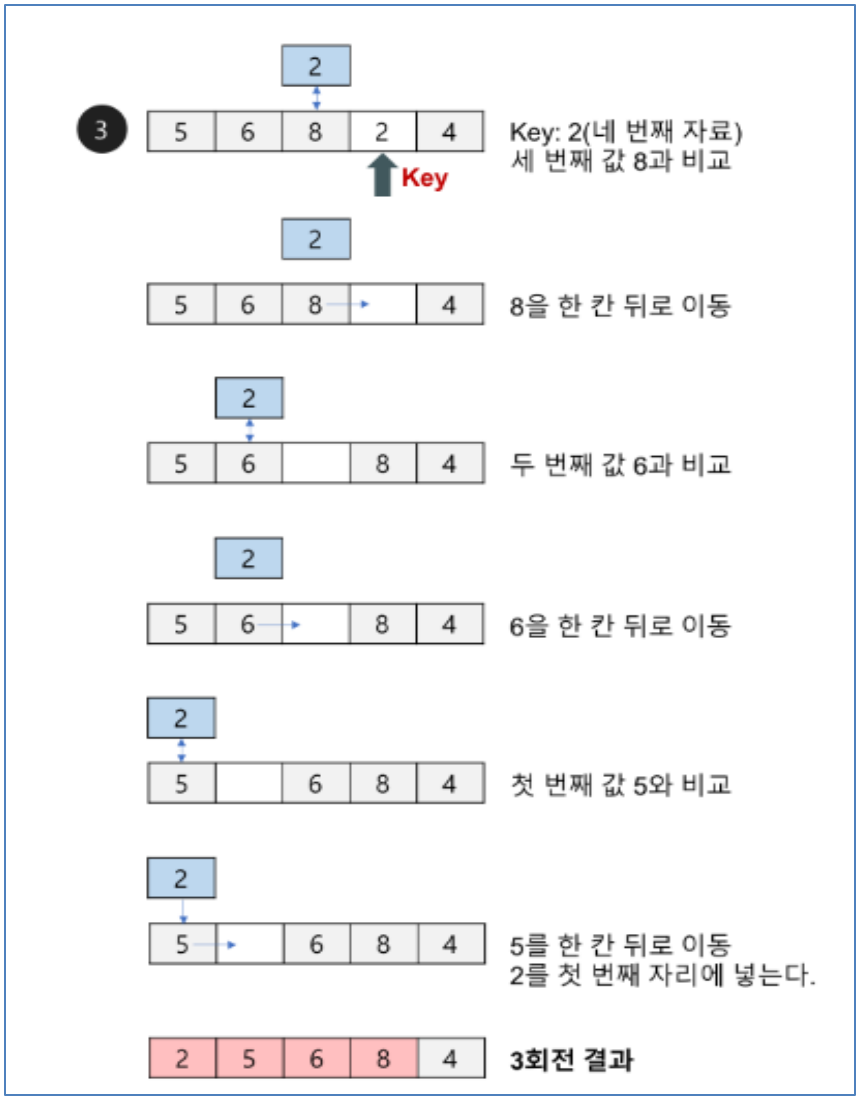


정렬 알고리즘 – 삽입 정렬

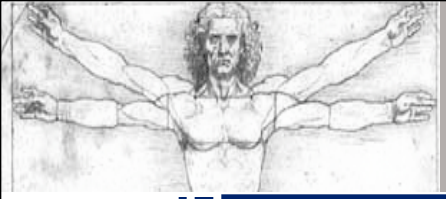
Source: https://gmlwjd9405.github.io/



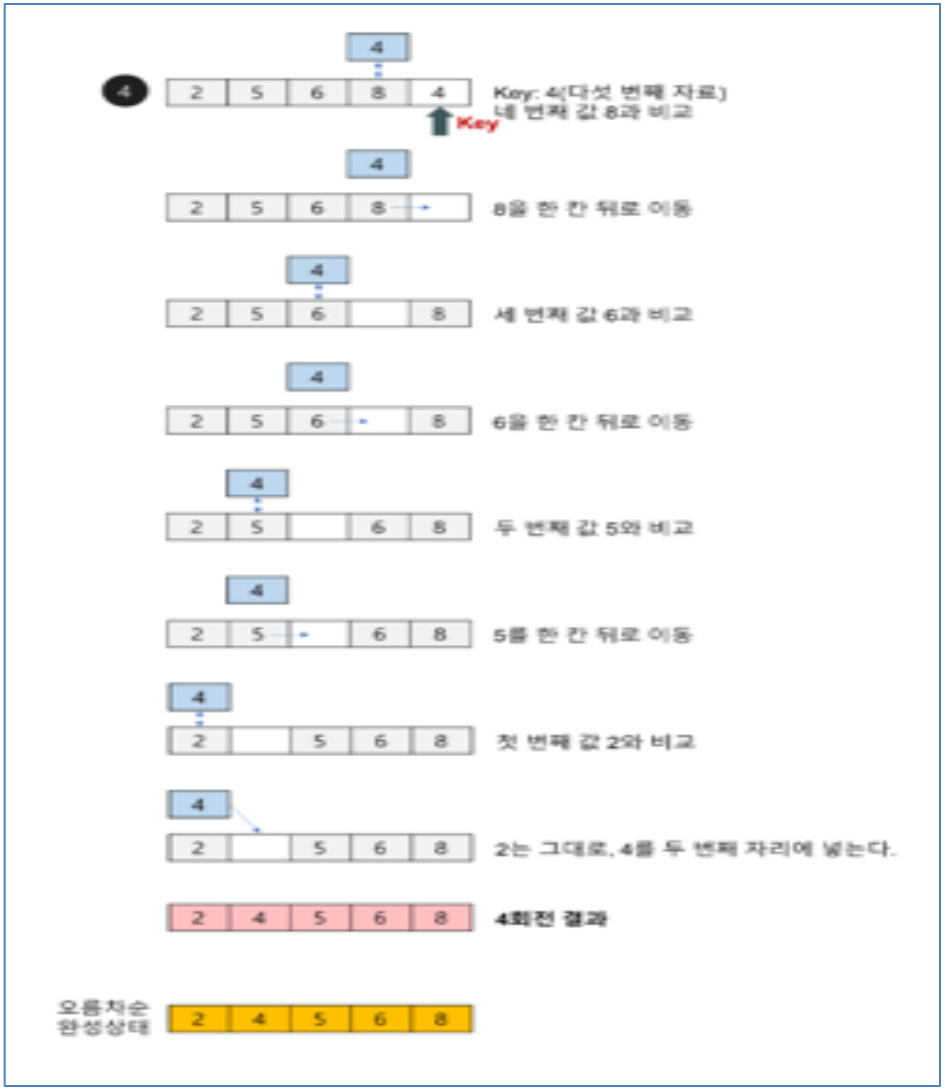
정렬 알고리즘 – 삽입 정렬



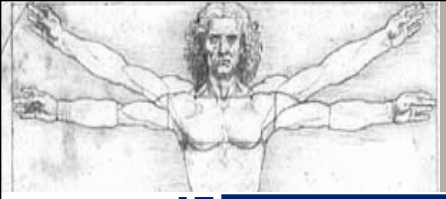
Source: https://gmlwjd9405.github.io/



정렬 알고리즘 – 삽입 정렬



Source: https://gmlwjd9405.github.io/



정렬 알고리즘 – 삽입 정렬

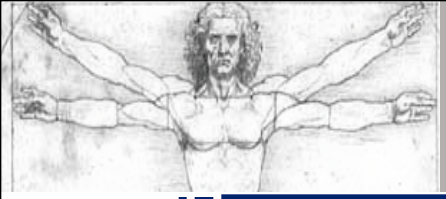
```python
import numpy as np
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]  # 삽입할 요소
        j = i - 1

        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]  # 한 칸씩 뒤로 이동
            j -= 1

        arr[j + 1] = key  # 삽입

    return arr
A = np.random.randint(1,2000, size=100)
print(A)
B = insertion_sort(A)
print(B)
```

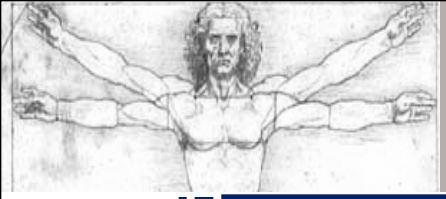
```
[1787 1286 1425  662    3 1223   21 1420 1417  443  952  906  514  770
 1904  755  539  715 1585  752 1920  885  860 1040  815  350 1769  595
   93 1532 1325   30 1432  662  439  625 1974  424 1295  198  457 1310
  944  509 1886 1672 1683 1070 1503 1877 1882  456  506 1880 1601 1644
  405  707  789  672 1778 1451 1181 1011  156  124  929 1613  507 1307
  757 1913 1632  470 1034  810  910 1940 1794 1387 1209  398  712  118
  212  258 1583 1349 1122 1261  764 1794  739  416 1666 1128  414 1871
 1477  944]
[   3   21   30   93  118  124  156  198  212  258  350  398  405  414
  416  424  439  443  456  457  470  506  507  509  514  539  595  625
  662  662  672  707  712  715  739  752  755  757  764  770  789  810
  815  860  885  906  910  929  944  944  952 1011 1034 1040 1070 1122
 1128 1181 1209 1223 1261 1286 1295 1307 1310 1325 1349 1387 1417 1420
 1425 1432 1451 1477 1503 1532 1583 1585 1601 1613 1632 1644 1666 1672
 1683 1769 1778 1787 1794 1794 1871 1877 1880 1882 1886 1904 1913 1920
 1940 1974]
```



정렬 알고리즘 – 삽입 정렬

- 시간이 얼마나 걸리나?
  - Pseudo-code에서 대략적인 시간을 계산해보자
  - N is length of array
  - t_j denote the number of of times the while loop test is executed for that value of j

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
$i = j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

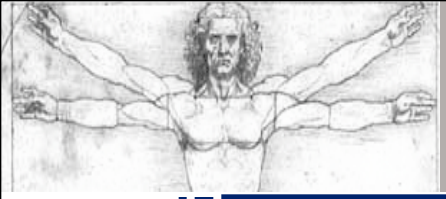


정렬 알고리즘 – 삽입 정렬

- 시간이 얼마나 걸리나?

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
$i = j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
$i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
$A[i+1] = key$	c_8	$n-1$

The running $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j-1) + c_6 \sum_{j=2}^{n}(t_j-1) + c_7(n-1)$



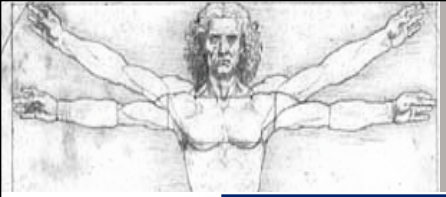
정렬 알고리즘 – 삽입 정렬

- 시간이 얼마나 걸리나?
  - 최적의 조건일때는 얼마나 걸리지?
  - 배열 A가 이미 정렬 되있을 경우

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
$i = j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

- Running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$



정렬 알고리즘 – 삽입 정렬

- 시간이 얼마나 걸리나?
  - 최악의 조건일때는?
  - If A is sorted in reverse order

Note that,

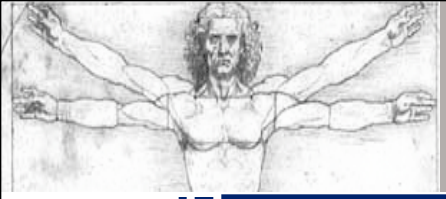
$$\sum_{j=2}^{n} t_j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} t_j - 1 = \frac{n(n-1)}{2}$$

Worst case

The array is in reverse sorted order.

- Always find that $A[i] > key$ in while loop test.
- Have to compare *key* with all elements to the left of the jth position $\Rightarrow$ compare with $j - 1$ elements.
- Since the while loop exits because i reaches 0, there's one additional test after the $j - 1$ tests $\Rightarrow t_j = j$.
- $\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$ and $\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j - 1)$.
- $\sum_{j=1}^{n} j$ is known as an ***arithmetic series***, and equation (A.1) shows that it equals $\frac{n(n+1)}{2}$.



정렬 알고리즘 – 삽입 정렬

- 시간이 얼마나 걸리나?
  - 최악의 조건일때는?
  - 직관적으로 이해해보자

$$T(n) = an^2 + bn + c$$

A quadratic function!

- Since $\sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1$, it equals $\frac{n(n+1)}{2} - 1$.

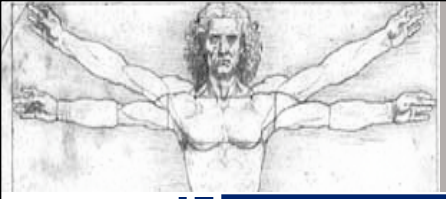
  [The parentheses around the summation are not strictly necessary. They are there for clarity, but it might be a good idea to remind the students that the meaning of the expression would be the same even without the parentheses.]

- Letting $k = j - 1$, we see that $\sum_{j=2}^{n}(j-1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$.

- Running time is

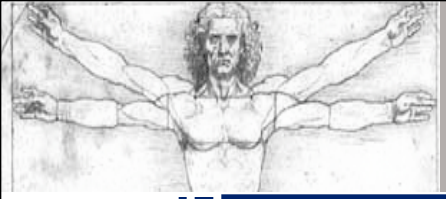
$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

- Can express $T(n)$ as $an^2 + bn + c$ for constants a, b, c (that again depend on statement costs) $\Rightarrow T(n)$ is a *quadratic function* of n.



정렬 알고리즘 – 점근적 표기법

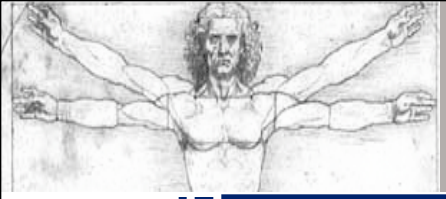
- 일일이 시간복잡도에 대해 계산하고 비교하는것
  - 굉장히 지루하고 귀찮타!
- 대략적으로 어떻게 표현하지? – 점근적 표기법 (Asymptotic notation)
  - 빅오(Big-O): O notation
    - 최악의 경우 시간 복잡도를 나타냄
    - 입력 데이터의 크기 n이 충분히 크다면, 알고리즘의 수행 시간은 f(n)의 상한에 비례
    - “Less than or equal to”
  - 빅오메가(Big-Omega): Ω (Big Omega notation)
    - 최선의 경우 시간 복잡도를 나타냄
    - 입력 데이터의 크기 n이 충분히 크다면, 알고리즘의 수행 시간은 f(n)의 하한에 비례
    - “greater than or equal to”
  - 빅세타(Big-Theta): Θ notation (Big theta notation)
    - 평균적인 경우 시간 복잡도
    - 입력 데이터의 크기 n이 충분히 크다면, 알고리즘의 수행 시간은 f(n)의 상한과 하한에 비례
    - “equal to (not exactly but satisfactory)”



정렬 알고리즘 – 점근적 표기법

- **Order of growth**
  - 알고리즘이 입력 크기에 따라서 어떻게 실행 시간이 증가하는지를 나타내는 지표
  - 시간 복잡도 함수의 가장 높은 차수의 항만을 고려 (빅오 표기법 사용)

n	$0.1n^2$	$0.1n^2 + n + 100$
10	10	120
20	40	160
50	250	400
100	1,000	1,200
1,000	100,000	101,100

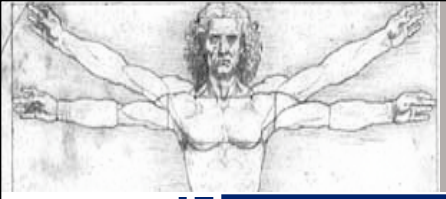


정렬 알고리즘 – 점근적 표기법

- **Big-Oh notation (빅-오 표기법)**
  - 시간 복잡도(알고리즘이 입력 크기에 대해 얼마나 빠르게 실행되는지)를 표현하기 위한 수학적 기호
  - 최악의 경우 알고리즘이 실행되는 시간의 상한을 나타내며, 알고리즘의 효율성을 평가하는 데 매우 유용

Number of operations	Asymptotic Running Time
$\frac{1}{10} \cdot n^2 + 100$	$O(n^2)$
$0.063 \cdot n^2 - .5\,n + 12.7$	$O(n^2)$
$100 \cdot n^{1.5} - 10^{10000}\sqrt{n}$	$O(n^{1.5})$
$11 \cdot n\log(n) + 1$	$O(n\log(n))$

우리는 이 알고리즘이 다른 알고리즘들 보다, "점근적으로 더 빠르다"라고 이야기 할 수 있다.

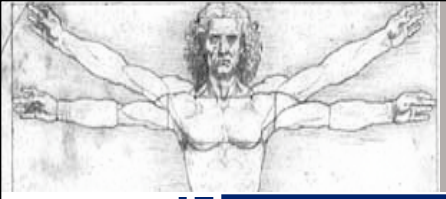


정렬 알고리즘 – 점근적 표기법

- **Big-Oh notation (빅-오 표기법)**
  - $T(n)$, $g(n)$를 양의 정수에 대한 함수라고 할 때, $T(n)$을 실행시간 (수행시간)이라고 하면 n이 커지면 $T(n)$도 커진다
  - 만약, 다음의 조건을 만족한다면, "$T(n)$은 $O(g(n))$"라고 할 수 있다:
    - 충분히 큰 n에 대해서 (for large enough n), $T(n)$은 아무리 커도 $g(n)$의 상수 배 만큼의 값을 가진다.
    - 즉, 어떤 n을 대입하더라도 $T(n)$을 $cg(n)$ 보다 작거나 같게 할 수 있는 양수 c가 존재한다.
      - 그렇다면 $T(n)$을 $O(g(n))$이라고 나타낼 수 있다.

Definition of *O-notation*

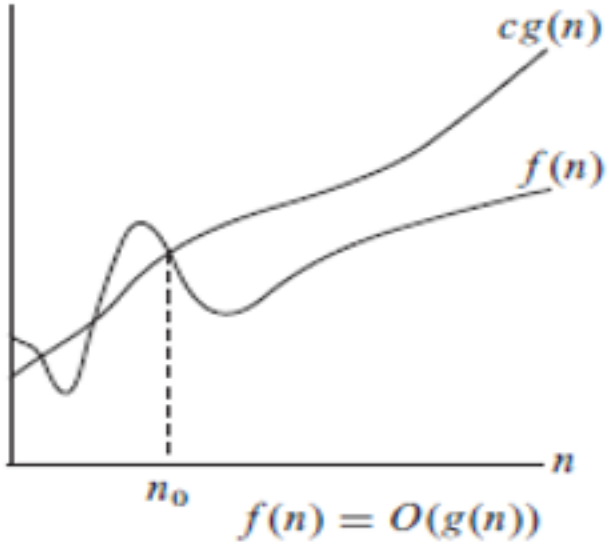
- $O(g(n)) = \{f(n)$: there exist constants $c > 0, n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$

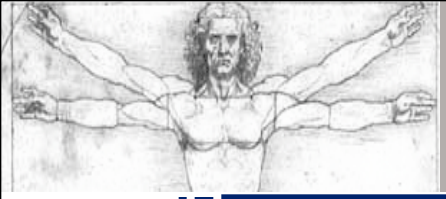


정렬 알고리즘 – 점근적 표기법

- **Big-Oh notation (빅-오 표기법)**
  - $T(n)$, $g(n)$를 양의 정수에 대한 함수라고 할 때, $T(n)$을 실행시간 (수행시간)이라고 하면 n이 커지면 $T(n)$도 커진다
  - 만약, 다음의 조건을 만족한다면, "$T(n)$은 $O(g(n))$"라고 할 수 있다:
    - 충분히 큰 n에 대해서 (for large enough n), $T(n)$은 아무리 커도 $g(n)$의 상수 배 만큼의 값을 가진다.
    - 즉, 어떤 n을 대입하더라도 $T(n)$을 $cg(n)$ 보다 작거나 같게 할 수 있는 양수 c가 존재한다.
      - 그렇다면 $T(n)$을 $O(g(n))$이라고 나타낼 수 있다.

f(n)	O(g(n))
$6n + 8$	$\in O(n)$
$2n + 5$	$\in O(n^2)$
$3n^2 + 7n + 1$	$\in O(n^2)$
$5 \cdot 4^n + n^2$	$\in O(2^n)$
$3n^2 + 4n + 1$	$\neq O(n)$

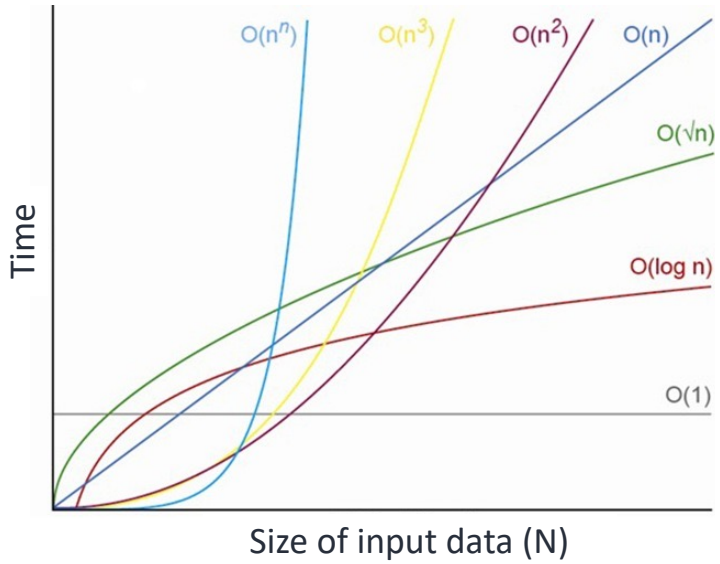


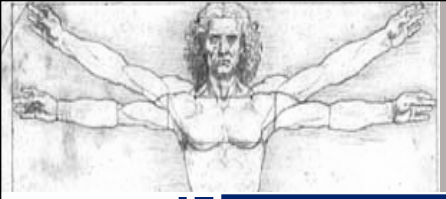


정렬 알고리즘 – 점근적 표기법

- **Big-Oh notation (빅-오 표기법)**
  - $T(n)$, $g(n)$를 양의 정수에 대한 함수라고 할 때, $T(n)$을 실행시간 (수행시간)이라고 하면 n이 커지면 $T(n)$도 커진다
  - 만약, 다음의 조건을 만족한다면, "$T(n)$은 $O(g(n))$"라고 할 수 있다:
    - 충분히 큰 n에 대해서 (for large enough n), $T(n)$은 아무리 커도 $g(n)$의 상수 배 만큼의 값을 가진다.
    - 즉, 어떤 n을 대입하더라도 $T(n)$을 $cg(n)$ 보다 작거나 같게 할 수 있는 양수 c가 존재한다.
      - 그렇다면 $T(n)$을 $O(g(n))$이라고 나타낼 수 있다.

f(n)	O(g(n))
$6n + 8$	$\in O(n)$
$2n + 5$	$\in O(n^2)$
$3n^2 + 7n + 1$	$\in O(n^2)$
$5 \cdot 4^n + n^2$	$\in O(2^n)$
$3n^2 + 4n + 1$	$\neq O(n)$





정렬 알고리즘 – 삽입 정렬

- 삽입정렬의 시간복잡도를 계산하면?
  - 우리는 항상 최악의 시나리오를 고려한다

$$T(n)$$

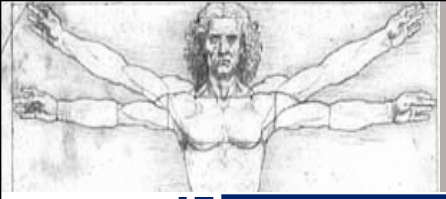
$$= c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right)$$

$$+ c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1)$$

$$= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} + \frac{c_5}{2} - \frac{c_6}{2} + c_7\right)n$$

$$- (c_2 + c_3 + c_4 + c_7)$$

$$= an^2 + bn + c = \boldsymbol{O(n^2)}$$



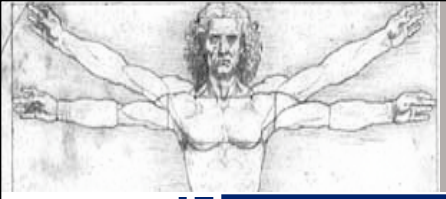
정렬 알고리즘 – 병합 정렬

- 병합 정렬(Merge Sort)
  - 분할 정복(Divide and Conquer) 방식을 이용한 정렬 알고리즘
  - 말그대로 쪼갰다가 합친다
    - 입력 배열을 반으로 나눔
    - 각각의 부분 배열을 재귀적으로 정렬
    - 두 부분 배열을 병합하여 정렬된 배열을 생성

Divide by splitting into two subarrays $A[p..q]$ and $A[q+1..r]$, where q is the halfway point of $A[p..r]$.

Conquer by recursively sorting the two subarrays $A[p..q]$ and $A[q+1..r]$.

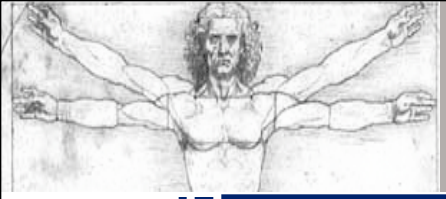
Combine by merging the two sorted subarrays $A[p..q]$ and $A[q+1..r]$ to produce a single sorted subarray $A[p..r]$. To accomplish this step, we'll define a procedure MERGE(A, p, q, r).



정렬 알고리즘 – 병합 정렬

- 예시
  - 입력 배열 : [38, 27, 43, 3, 9, 82, 10]
  - 반으로 나누기
    - [38, 27, 43, 3, 9, 82, 10] -> [38, 27, 43, 3], [9, 82, 10]
      - [38, 27, 43, 3] -> [38, 27], [43, 3] -> [38], [27], [43], [3]
      - [9, 82, 10] -> [9], [82, 10] -> [9], [82], [10]
    - 재귀적으로 더 이상 안 나누어 질때까지 나눈다
  - 두 부분 배열을 병합하여 정렬된 배열을 생성
    - [38], [27] -> [27, 38]
    - [43], [3] -> [3, 43]
    - [27, 38], [3, 43] -> [3, 27, 38, 43]
    - [82], [10] -> [10, 82]
    - [9], [10, 82] -> [9, 10, 82]
    - [3, 27, 38, 43], [9, 10, 82] -> [3, 9, 10, 27, 38, 43, 82]

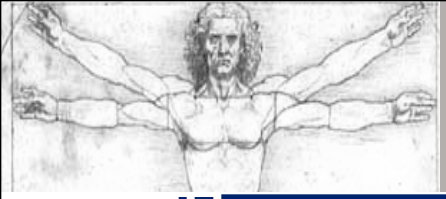
```
MERGE-SORT(A, p, r)
 if p < r                          // check for base case
     q = ⌊(p + r)/2⌋               // divide
     MERGE-SORT(A, p, q)           // conquer
     MERGE-SORT(A, q + 1, r)       // conquer
     MERGE(A, p, q, r)             // combine
```



정렬 알고리즘 – 병합 정렬

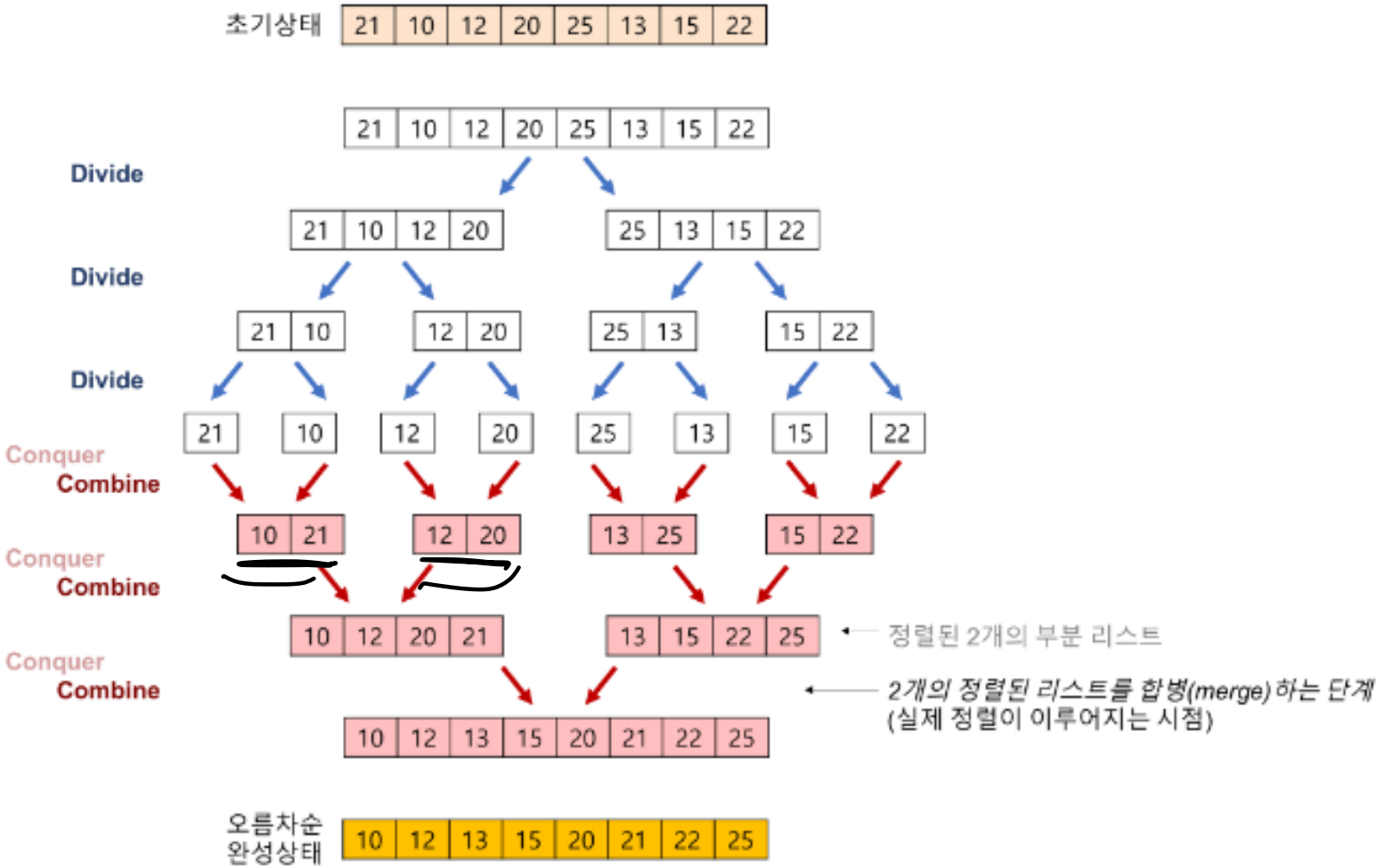
- 예시
  - 입력 배열 : [38, 27, 43, 3, 9, 82, 10]
  - 반으로 나누기
    - [38, 27, 43, 3, 9, 82, 10] -> [38, 27, 43, 3], [9, 82, 10]
      - [38, 27, 43, 3] -> [38, 27], [43, 3] -> [38], [27], [43], [3]
      - [9, 82, 10] -> [9], [82, 10] -> [9], [82], [10]
    - 재귀적으로 더 이상 안 나누어 질때까지 나눈다
  - 두 부분 배열을 병합하여 정렬된 배열을 생성
    - [38], [27] -> [27, 38]
    - [43], [3] -> [3, 43]
    - [27, 38], [3, 43] -> [3, 27, 38, 43]
    - [82], [10] -> [10, 82]
    - [9], [10, 82] -> [9, 10, 82]
    - [3, 27, 38, 43], [9, 10, 82] -> [3, 9, 10, 27, 38, 43, 82]

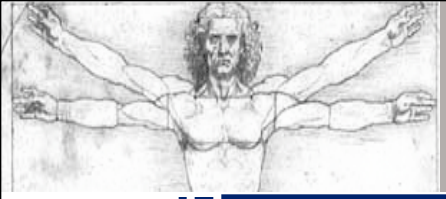
```
MERGE-SORT(A, p, r)
 if p < r                          // check for base case
    q = ⌊(p + r)/2⌋                // divide
    MERGE-SORT(A, p, q)            // conquer
    MERGE-SORT(A, q + 1, r)        // conquer
    MERGE(A, p, q, r)              // combine
```



정렬 알고리즘 – 병합 정렬

- **Merge 함수는?**



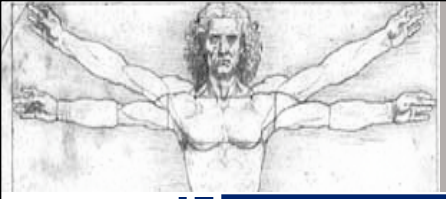


정렬 알고리즘 – 병합 정렬

- Merge 함수는?

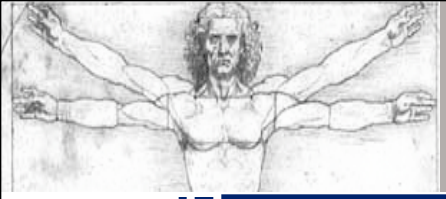
Pseudocode

```
MERGE(A, p, q, r)
  n1 = q − p + 1
  n2 = r − q
  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
  for i = 1 to n1
      L[i] = A[p + i − 1]
  for j = 1 to n2
      R[j] = A[q + j]
  L[n1 + 1] = ∞
  R[n2 + 1] = ∞
  i = 1
  j = 1
  for k = p to r
      if L[i] ≤ R[j]
          A[k] = L[i]
          i = i + 1
      else A[k] = R[j]
          j = j + 1
```



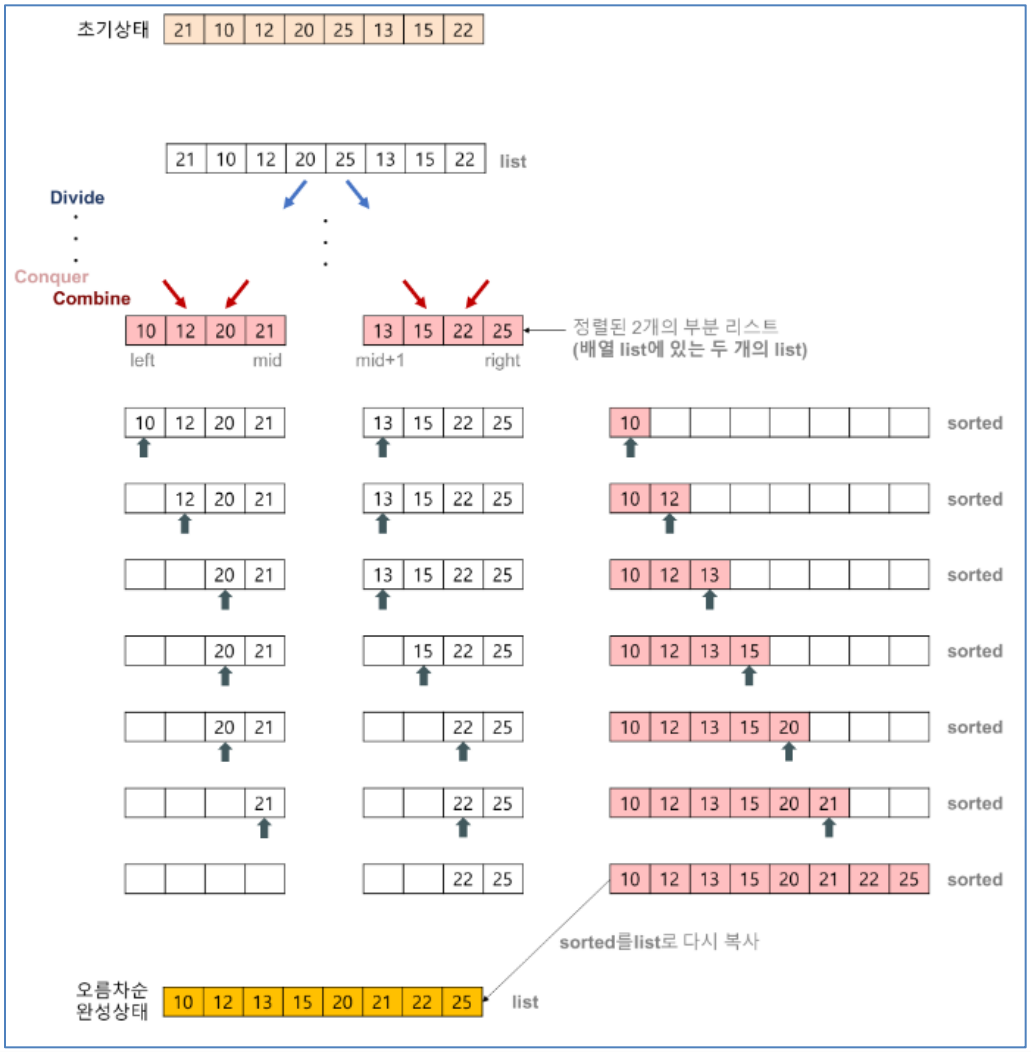
정렬 알고리즘 – 병합 정렬

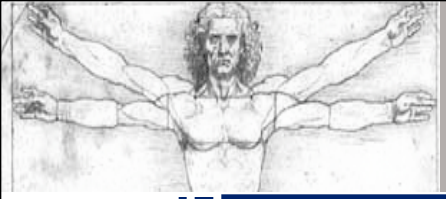
- **Merge 함수는?**
  - 2개의 리스트의 값들을 처음부터 하나씩 비교하여 두 개의 리스트의 값 중에서 더 작은 값을 새로운 리스트(sorted)로 옮긴다.
  - 둘 중에서 하나가 끝날 때까지 이 과정을 되풀이한다.
  - 만약 둘 중에서 하나의 리스트가 먼저 끝나게 되면 나머지 리스트의 값들을 전부 새로운 리스트(sorted)로 복사한다.
  - 새로운 리스트(sorted)를 원래의 리스트(list)로 옮긴다.



정렬 알고리즘 – 병합 정렬

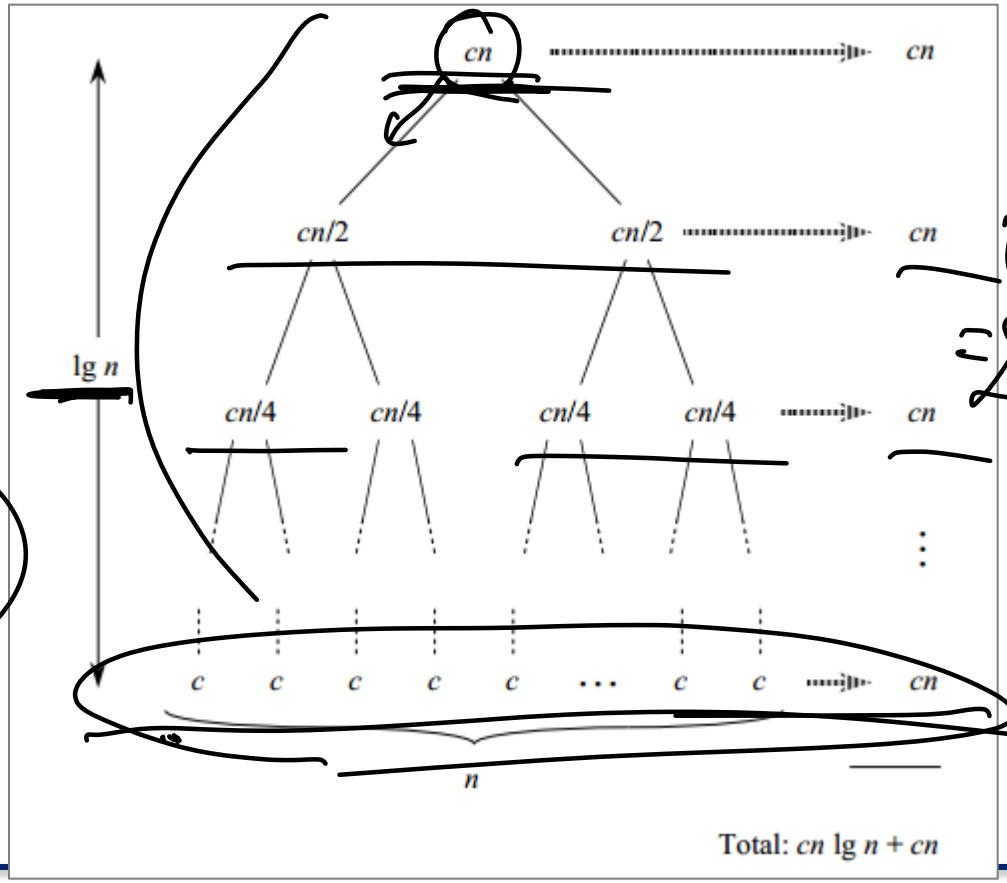
- **Merge 함수는?**



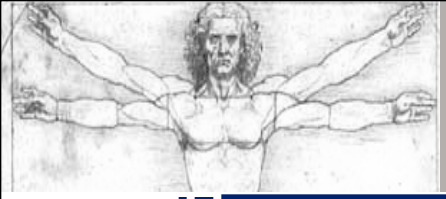


정렬 알고리즘 – 병합 정렬

- 병합정렬의 시간복잡도는 $O(n\log(n))$
  - 노드 레벨의 수
    - 노드의 개수는 다음과 같이 늘어난다: $2^0, 2^1, 2^2, 2^3 \ldots$
    - 노드 총 레벨 수는 $\log_2 n + 1$

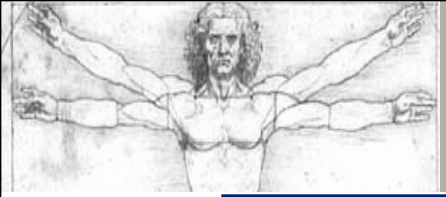


T(cn)
= 2T(n/2)
+ cn.



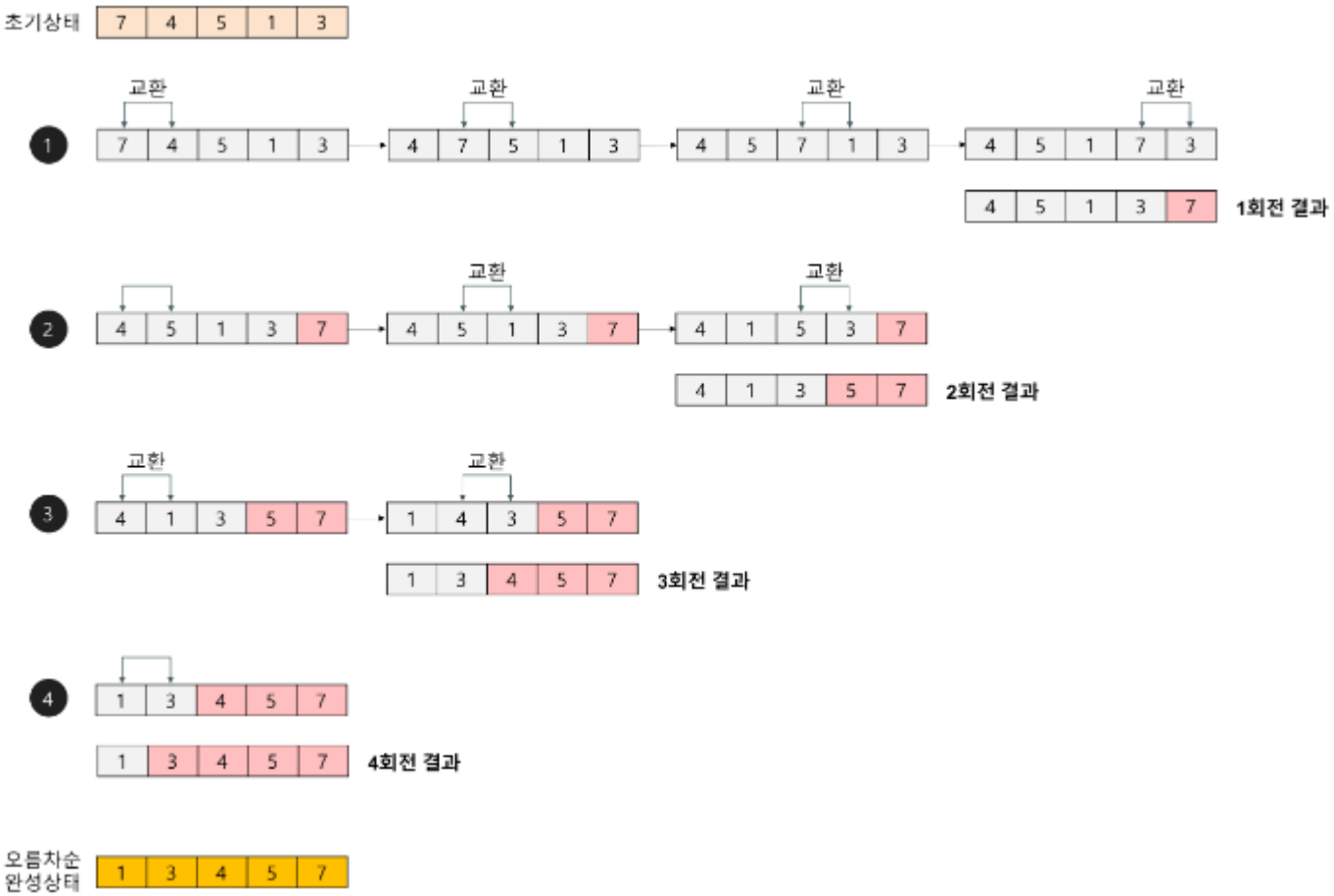
정렬 알고리즘 – 병합 정렬

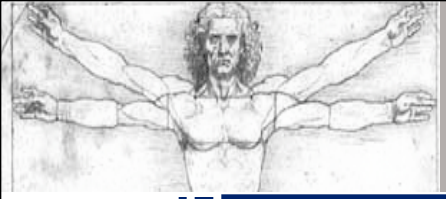
- 병합정렬의 시간복잡도는 $O(n\log(n))$
  - 분할
    - 병합 정렬은 입력 배열을 계속해서 반으로 나누어, 하위 배열이 1개의 원소만 남을 때까지 반복적으로 나눔
    - 반으로 나누는 과정은 $O(\log(n))$
  - 정복
    - 하위 배열의 정렬은 분할된 배열 크기의 합이 n인 경우 $O(n)$의 시간이 소요
    - 배열 전체의 크기가 n일 때, 하위 배열은 최대 $\log(n)$개의 단계를 거침
    - 이 단계에서는 $O(n\log(n))$
  - 병합
    - 두 배열을 병합하는 과정에서는 각 배열의 크기만큼 순서대로 비교하며 새로운 배열에 넣어주면 됨
    - 병합 과정의 시간 복잡도는 $O(n)$
  - 전체 시간 복잡도
    - $O(\log(n)) + O(\mathrm{nlog}(n)) + O(n) = O(\mathrm{nlog}(n))$



정렬 알고리즘 – 버블정렬(Bubble Sort)

- 비효율적이지만 인기있는 알고리즘, 구현하기 쉬움
  - 인접한 두 원소를 비교하여 작은 원소를 앞으로, 큰 원소를 뒤로 보내는 작업을 반복함



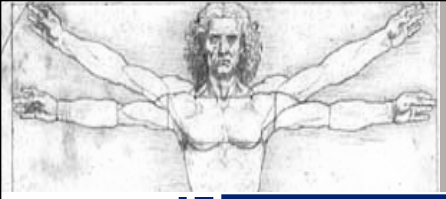


정렬 알고리즘 – 버블 정렬

- **Pseudo code**

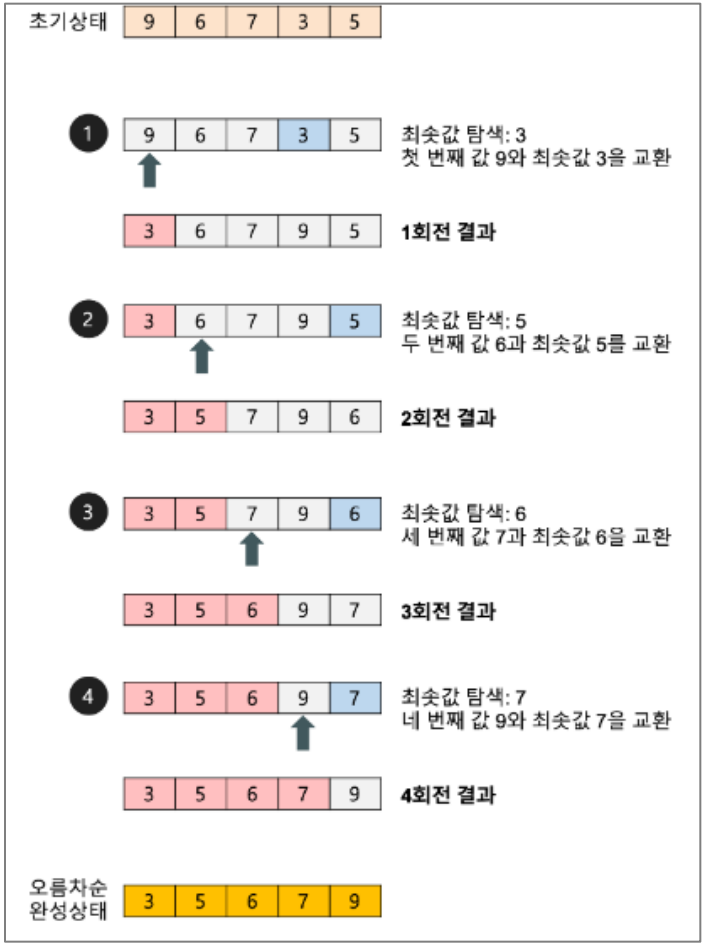
$\boldsymbol{BubbleSort}(arr)$:
  $n = \boldsymbol{len}(arr)$
  $\boldsymbol{for}\ i = 0\ to\ n$
    $\boldsymbol{for}\ j = n\ \boldsymbol{downto}\ i + 1$
      if A[j] < A[j − 1]
      Swap(arr[j], arr[j − 1])

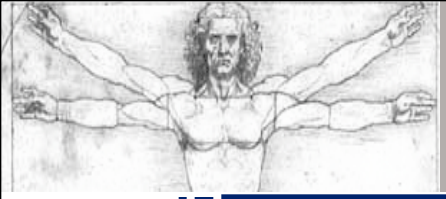
- 시간 복잡도
  - 최선, 평균 최악 모두 $\mathbf{O}(n^2)$
  - 이는 배열의 모든 원소를 비교하며 위치를 바꾸는 작업을 반복하기 때문



정렬 알고리즘 – 선택정렬

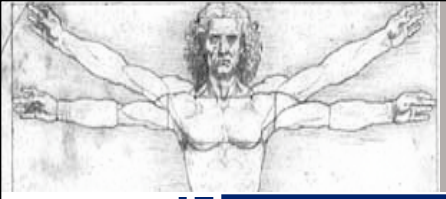
- **Selection sort**
  - 가장 작은 또는 큰 원소를 찾아서 앞쪽으로 이동시키는 정렬
    - 주어진 배열 중에서 최솟값을 찾음
    - 그 값을 맨 앞에 위치한 값과 교체
    - 맨 처음 위치를 뺀 나머지 리스트를 같은 방법으로 교체
    - 하나의 원소만 남을 때까지 위의 1~3 과정을 반복
- 시간 복잡도
  - **Bubble sort**와 마찬가지로 모든 배열을 순회하기 때문에 $O(n^2)$





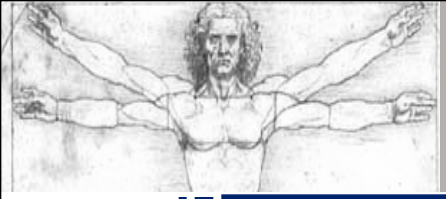
정렬 알고리즘 – 셸정렬(Shell sort)

- **Shell sort**
  - 삽입 정렬(Insertion Sort)의 성능을 개선한 알고리즘으로, 데이터를 일정한 간격으로 나누어서 삽입 정렬을 수행하는 방식으로 동작
    - 먼저 정렬해야 할 리스트를 일정한 기준에 따라 분류
    - 연속적이지 않은 여러 개의 부분 리스트를 생성
    - 각 부분 리스트를 삽입 정렬을 이용하여 정렬
    - 모든 부분 리스트가 정렬되면 다시 전체 리스트를 더 적은 개수의 부분 리스트로 만든 후에 알고리즘을 반복
    - 위의 과정을 부분 리스트의 개수가 1이 될 때까지 반복
- 시간 복잡도
  - Bubble sort와 마찬가지로 모든 배열을 순회하기 때문에 $O(n^2)$

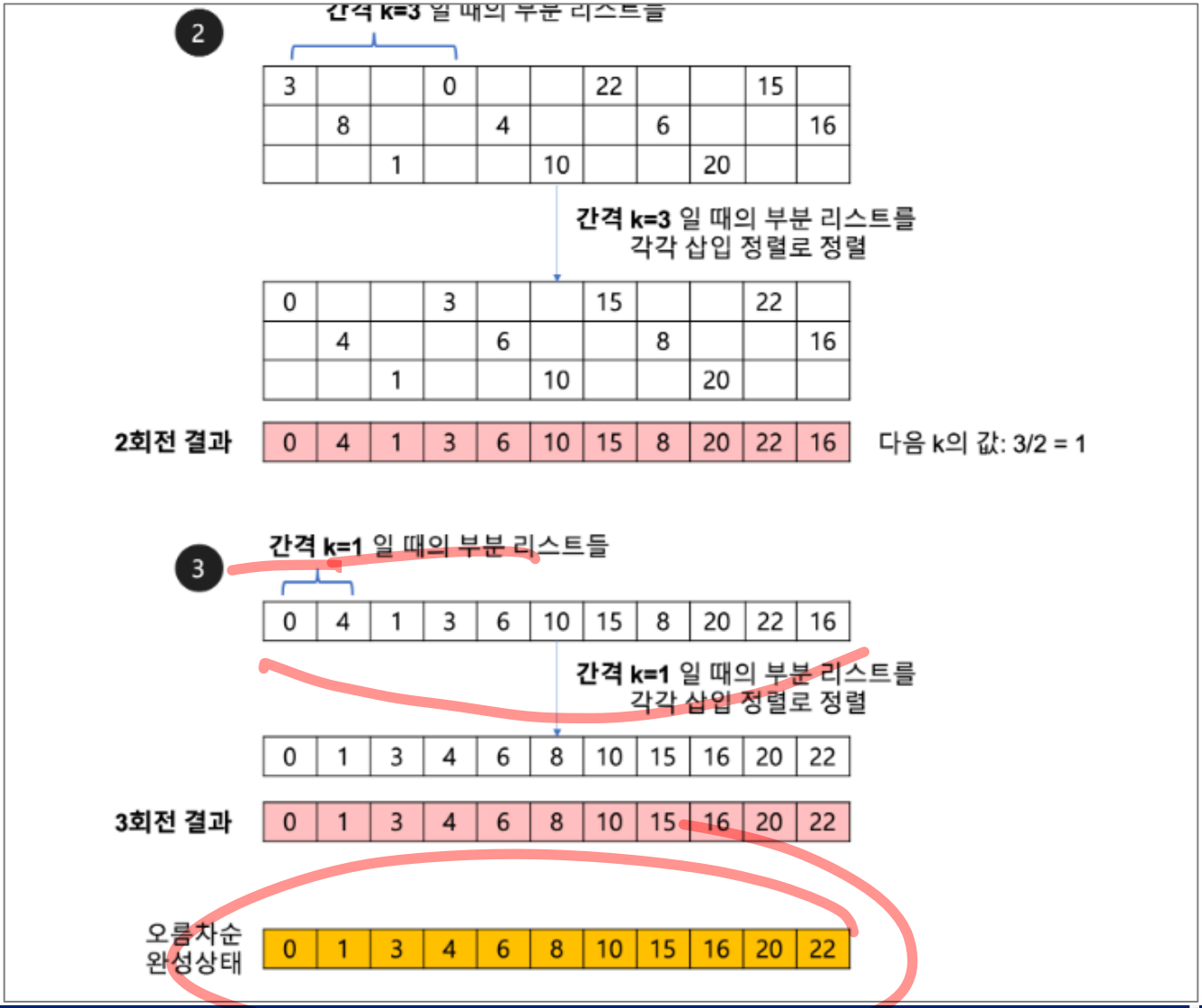


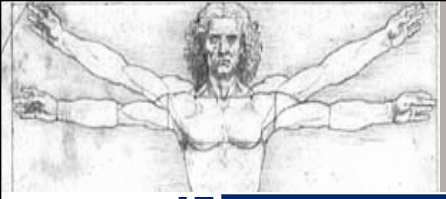
정렬 알고리즘 – 셸정렬(Shell sort)

- **Shell sort**
  - 삽입 정렬(Insertion Sort)의 성능을 개선한 알고리즘으로, 데이터를 일정한 간격으로 나누어서 삽입 정렬을 수행하는 방식으로 동작
    - 먼저 정렬해야 할 리스트를 일정한 기준에 따라 분류
    - 연속적이지 않은 여러 개의 부분 리스트를 생성
    - 각 부분 리스트를 삽입 정렬을 이용하여 정렬
    - 모든 부분 리스트가 정렬되면 다시 전체 리스트를 더 적은 개수의 부분 리스트로 만든 후에 알고리즘을 반복
    - 위의 과정을 부분 리스트의 개수가 1이 될 때까지 반복



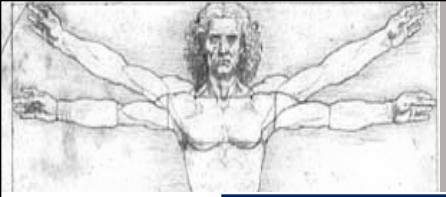
정렬 알고리즘 – 셸정렬(Shell sort)





정렬 알고리즘 – 셸정렬(Shell sort)

- 시간 복잡도
  - $O(nlogn)$에서 $O(n^2)$ 사이로 나타남
  - 간격(Interval)이 어떻게 설정되느냐에 따라 성능이 달라짐



Agenda

- 다음 시간
  - Growth of function & 점근적 표기법
  - 분할정복