정렬 알고리즘 (sorting algorithm)

2022. Fall

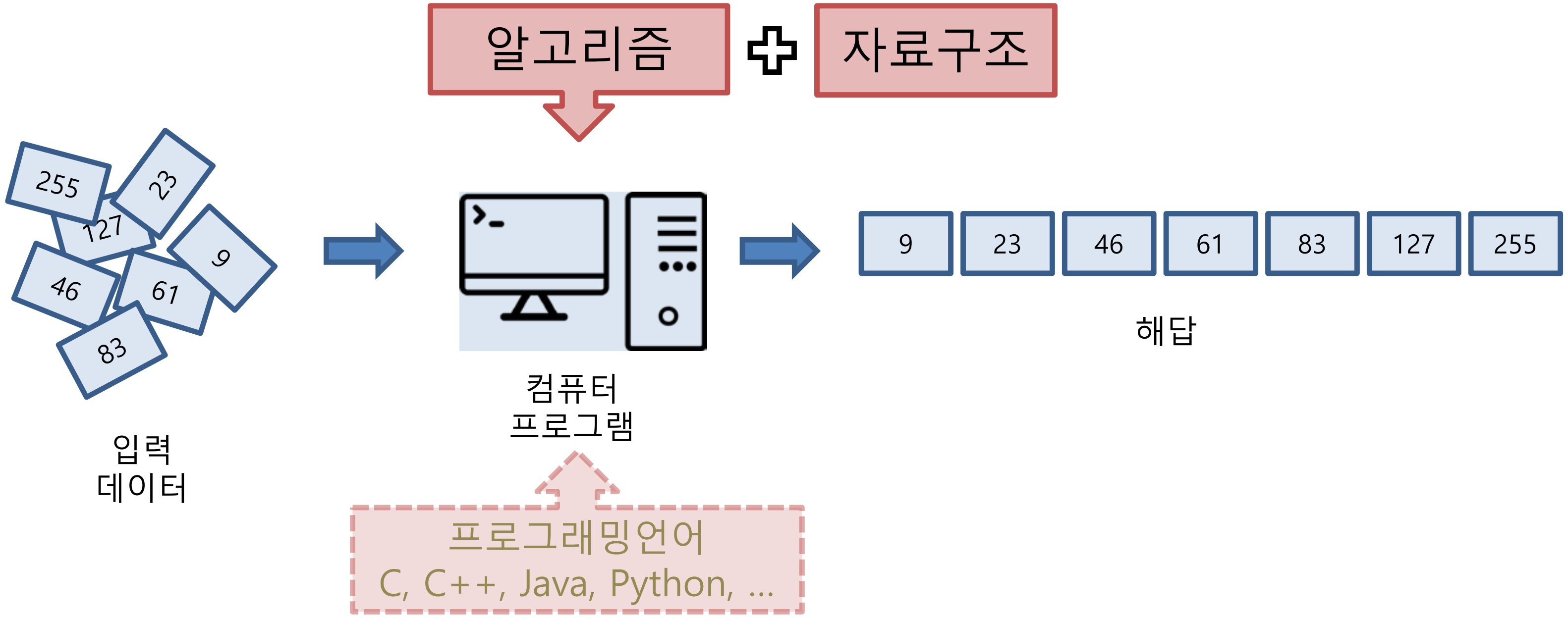
국민대학교 소프트웨어학부
최 준수

Sorting Problem

- 정렬 문제

정렬 문제

모든 데이터 처리의 기본이 되는 문제



Sorting Algorithms

- Comparison-based sorting algorithms
  - 데이터의 크기를 비교하여 정렬하는 알고리즘
    - Non-comparison-based sorting algorithms

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	No	Partitioning	Quicksort is usually done in-place with $O(\log n)$ stack space.[5][6]
Merge sort	$n \log n$	$n \log n$	$n \log n$	n	Yes	Merging	Highly parallelizable (up to $O(\log n)$ using the Three Hungarians' Algorithm).[7]
In-place merge sort	—	—	$n \log^2 n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.[8]
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No	Partitioning & Selection	Used in several STL implementations.
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection	
Insertion sort	n	n^2	n^2	1	Yes	Insertion	$O(n + d)$, in the worst case over sequences that have *d* inversions.
Block sort	n	$n \log n$	$n \log n$	1	Yes	Insertion & Merging	Combine a block-based $O(n)$ in-place merge algorithm[9] with a bottom-up merge sort.
Timsort	n	$n \log n$	$n \log n$	n	Yes	Insertion & Merging	Makes *n-1* comparisons when the data is already sorted or reverse sorted.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with $O(n)$ extra space, when using linked lists, or when made as a variant of Insertion Sort instead of swapping the two items.[10]
Cubesort	n	$n \log n$	$n \log n$	n	Yes	Insertion	Makes *n-1* comparisons when the data is already sorted or reverse sorted.
Shellsort	$n \log n$	$n^{4/3}$	$n^{3/2}$	1	No	Insertion	Small code size.
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Exchange sort	n^2	n^2	n^2	1	No	Exchanging	Tiny code size.

Sorting Algorithms

- Comparison-based sorting algorithms

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Tree sort	$n \log n$	$n \log n$	$n \log n$ (balanced)	n	Yes	Insertion	When using a self-balancing binary search tree.
Cycle sort	n^2	n^2	n^2	1	No	Selection	In-place with theoretically optimal number of writes.
Library sort	$n \log n$	$n \log n$	n^2	n	No	Insertion	Similar to a gapped insertion sort. It requires randomly permuting the input to warrant with-high-probability time bounds, which makes it not stable.
Patience sorting	n	$n \log n$	$n \log n$	n	No	Insertion & Selection	Finds all the longest increasing subsequences in $O(n \log n)$.
Smoothsort	n	$n \log n$	$n \log n$	1	No	Selection	An adaptive variant of heapsort based upon the Leonardo sequence rather than a traditional binary heap.
Strand sort	n	n^2	n^2	n	Yes	Selection	
Tournament sort	$n \log n$	$n \log n$	$n \log n$	n[11]	No	Selection	Variation of Heapsort.
Cocktail shaker sort	n	n^2	n^2	1	Yes	Exchanging	A variant of Bubblesort which deals well with small values at end of list
Comb sort	$n \log n$	n^2	n^2	1	No	Exchanging	Faster than bubble sort on average.
Gnome sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Odd–even sort	n	n^2	n^2	1	Yes	Exchanging	Can be run on parallel processors easily.

[ref] https://en.wikipedia.org/wiki/Sorting_algorithm

Design & Analysis of Algorithms

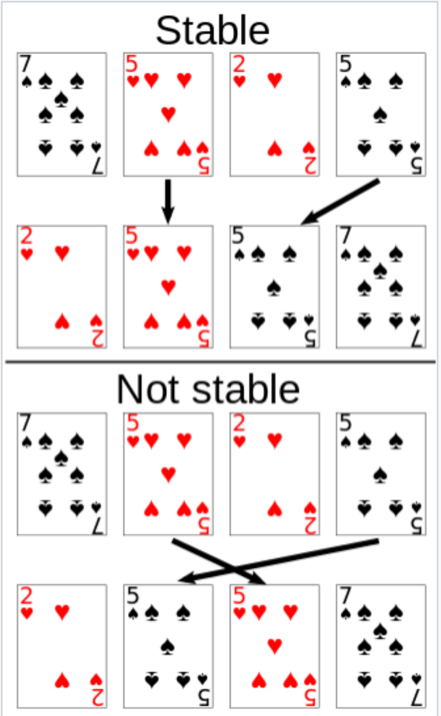
- Design of Algorithm
  - 문제를 해결하는 방법(알고리즘)은 아주 다양함
    - 예: 정렬 알고리즘
      - 아마 수백가지 알고리즘 존재하지 않을까?
- Analysis of Algorithm
  - 문제를 해결하도록 설계(design)한 알고리즘이 얼마나 효율적(time, memory)인지를 분석함
    - 얼마나 빨리 문제를 해결하는지?
    - 얼마나 메모리를 덜 사용하고 해결할 수 있는지?
    - 컴퓨터의 대표적인 자원인 CPU(실행시간)와 메모리를 얼마나 효율적으로 사용하는지를 나타냄
  - 효율성을 어떻게 나타내는가?
    - 입력되는 데이터의 크기(주로 데이터 개수, 변수 n으로 표시됨)의 수식으로 표현됨
    - big-O notation으로 표현됨
      - 수식이므로 간단하게 표현하면 좋을 것 같음. 비교하기 쉽게
        - » 숫자면 비교하기 편리할 텐데…

- Analysis of Algorithm
  - 이론적인 실행 시간(time complexity) 분석
    - best-case : 가장 좋은 데이터가 입력이 되었을 경우
    - average-case : 모든 데이터에 대한 평균
    - worst-case : 가장 최악의 데이터가 입력되었을 경우
  - 이론적인 사용하는 메모리 양(space complexity) 분석
    - 입력 데이터를 저장하는 메모리 양은 제외
    - 알고리즘에서 추가적으로 사용하는 메모리 양

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	No	Partitioning	Quicksort is usually done in-place with $O(\log n)$ stack space.[5][6]

Stability of Sorting Algorithm

- Stable sorting algorithm
  - 크기가 같은 데이터가 정렬된 이후에도 입력된 순서를 그대로 유지하게 하는 정렬 알고리즘



[ref] https://en.wikipedia.org/wiki/Sorting_algorithm

알고리즘	Stable ?
Quick	No
Merge	Yes
Heap	No
Insertion	Yes
Selection	No
Shell	No
Bubble	Yes
Exchange	No
Cocktail Shaker	Yes
Comb	No

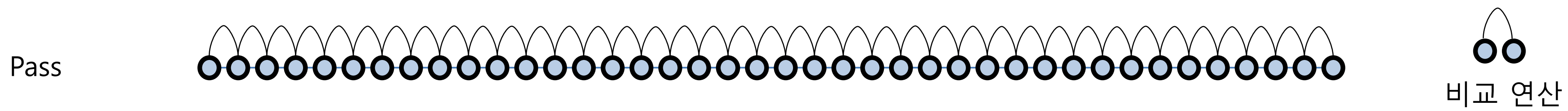
In-Place Algorithm

- In-Place algorithm
  - 입력 데이터를 저장하는 메모리 이외는 상수 크기의 메모리만 필요한 알고리즘
    - 상수: 입력 데이터의 크기(n)와는 무관하다는 의미
      - 다음 표에서는 1로 표현함
      - 주로 알고리즘 중간 계산에 필요한 변수 등에 필요한 메모리

알고리즘	Extra Memory	In-Place ?
Quick	$\log n$	No
Merge	n	No
Heap	1	Yes
Insertion	1	Yes
Selection	1	Yes
Shell	1	Yes
Bubble	1	Yes
Exchange	1	Yes
Cocktail Shaker	1	Yes
Comb	1	Yes

Bubble Sort

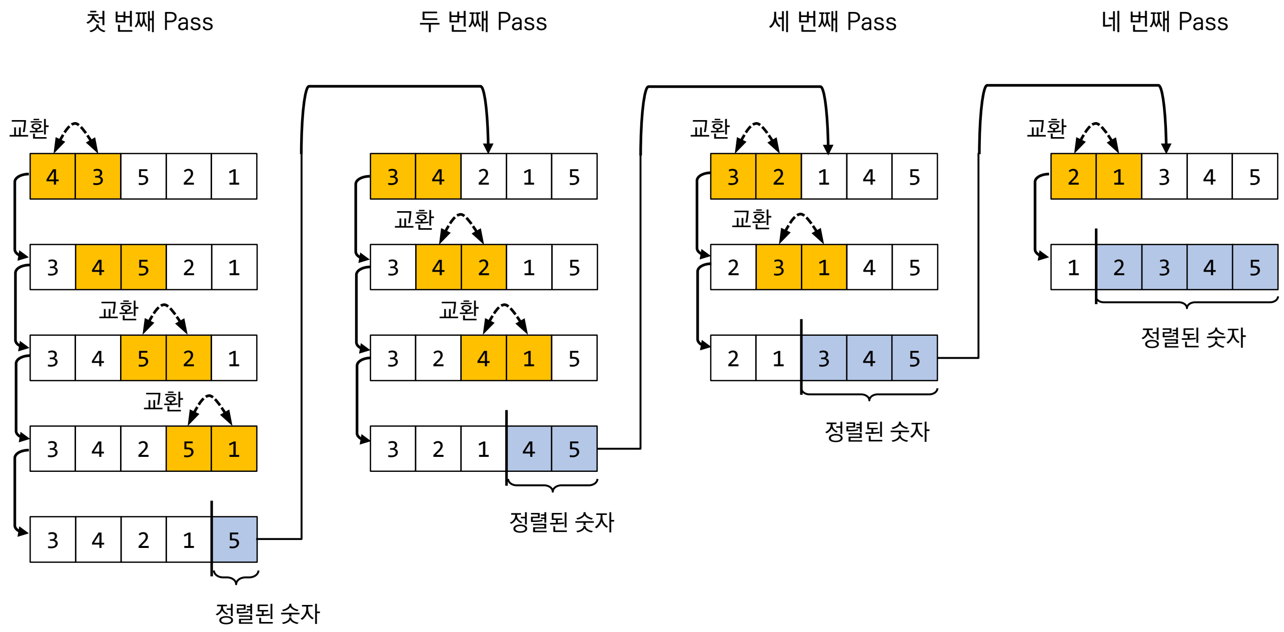
- 기본 아이디어
  - 인접한 두 숫자를 비교하여 두 수의 정렬순서가 맞지 않는 경우에는 교환(swap)함
    - 마치 깊은 물 속의 큰 물방울이 표면으로 떠 오르는 것과 같이
      큰 데이터들이 배열의 왼쪽에서 오른쪽 이동하기 때문에 bubble sort 라 부름
  - Pass
    - 맨 왼쪽 인접한 두 숫자를 비교하기 시작하여
      - 맨 끝 인접한 두 숫자를 비교할 때 까지 연속적으로 인접한 두 숫자를 비교함
      - 비교한 두 숫자의 정렬순서가 맞지 않을 경우에는 교환함
    - Pass의 결과
      - 제일 큰 숫자가 맨 오른쪽 끝으로 이동함
      - 이 숫자는 더 이상 다음 pass에 포함시킬 필요가 없음



비교 횟수: n-1

Bubble Sort

- Method
  - 인접한 두 수의 크기를 비교
    - 두 수를 정렬 순서에 맞도록 필요시 교환



비교 연산 실행 횟수: 4 3 2 1

총 횟수: $4 + 3 + 2 + 1 = 10$

Bubble Sort

- Psudeo-Code
  - 4-line bubble sort

```
BubbleSort( A )
    n = A.size                          // A[0], A[1], …, A[n-1]

    for(pass = 1; pass < n; pass++)     // pass = 1,2,…,n-1
        for(i = 1; i <= n – pass; i++)
            if(A[i-1] > A[i])           // > : 비교 연산자
                swap(A[i-1], A[i]);     // A[i-1]와 A[i]를 교환
```

Why?

- Stable
- In-place

- Analysis
  - 비교 연산자(>)의 실행 횟수는?
    - 입력 데이터의 개수 : n

pass	비교 연산자 실행 횟수
1	$n-1$
2	$n-2$
…	
$n-1$	1
총 횟수	$n(n-1)/2$

best case
average case
worst case

Improvement (1)

- Improved bubble sort algorithm (1)

```
BubbleSort( A )
    n = A.size                          // A[0], A[1], …, A[n-1]

    for(pass = 1; pass < n; pass++)     // pass = 1,2,…,n-1
    {
        swapped = false;                // 이번 pass에서 데이터 교환했는지 유무
        for(i = 1; i <= n - pass; i++)
            if(a[i-1] > a[i])           // > : 비교 연산자
            {
                swap(a[i-1], a[i]);
                swapped = true;         // 데이터 교환했음
            }

        if(swapped == false)            // 이번 pass에서 데이터를 교환하지 않았다면
            break;                      // 데이터가 정렬되어 있음. 따라서 종료함
    }
```

best case

pass	비교 연산자 실행 횟수
1	$n-1$
총 횟수	$n-1$

- Analysis
  - best case:
    - 입력데이터가 오름차순으로 정렬되어 있음
    - 비교 연산자 총 실행 횟수 : $n-1$
  - worst case
    - 입력데이터가 내림차순으로 정렬되는 등의 수많은 경우
    - 비교 연산자 비교횟수 : $n(n-1)/2$

Improvement (2, by J. Choi)

- Improved bubble sort algorithm (2)

```
BubbleSort( A )
    n = A.size                               // A[0], A[1], …, A[n-1]

     lastSwappedPos = n;
    for(lastSwappedPos > 0)                  // 마지막으로 교환한 데이터의 위치
    {
        swappedPos = 0;                      // 이번 pass에서 데이터 교환한 위치
        for(i = 1; i < lastSwappedPos; i++)  // 직전 pass에서 교환한 마지막 위치까지만 실행
            if(a[i-1] > a[i])                // > : 비교 연산자
            {
                swap(a[i-1], a[i]);
                swappedPos = i;              // 데이터 교환한 위치
            }

        lastSwappedPos = swappedPos;       // 이번 pass에서 교환한 마지막 데이터의 위치
                                           // 다음 pass에서는 이 위치 바로 앞까지만 실행하면 됨
    }
```

9	6	3	1	2	4	5	7	8

6	3	1	2	4	5	7	8	9

3	1	2	4	5	6	7	8	9

1	2	3	4	5	6	7	8	9

토끼와 거북이 데이터

- Bubble Sort
  - 토끼 데이터 (rabbit data)
    - 왼쪽에 있는 큰 데이터들은 빠르게(몇 번의 pass를 통하여) 오른쪽 제 위치로 이동함

Bubble sorting animation (wiki)
https://en.wikipedia.org/wiki/File:Bubble_sort_animation.gif

9	6	3	7	8	4	5	2	1
6	3	7	8	4	5	2	1	9
3	6	7	4	5	2	1	8	9

Pass 1

Pass 2

  - 거북이 데이터 (turtle data)
    - 오른쪽에 있는 작은 데이터들은 매우 느리게 왼쪽 제 위치로 이동함

9	6	3	7	8	4	5	2	1
6	3	7	8	4	5	2	1	9
3	6	7	4	5	2	1	8	9

Pass 1

Pass 2

… …

2	1	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9

Pass (n-1)

Cocktail Shaker Sort

- Cocktail Shaker Sort (Bidirectional bubble sort)
  - Bubble sort 의 단점을 개선
    - 거북이(turtle)데이터를 빠르게 제 위치로 이동시킴
    - Bubble sort의 각 pass를
      - 한 번은 왼쪽에서 오른쪽으로
      - 그 다음에는 오른쪽에서 왼쪽으로 실행

Cocktail Shaker sorting animation (wiki)
https://en.wikipedia.org/wiki/File:Sorting_shaker_sort_anim.gif

각 pass 에서 정렬 진행 방향

- Analysis
  - Worst case: $O(n^2)$
  - 입력 데이터가 많이 정렬되어 있는 상태라면 빠르게 정렬할 수 있음
    - 모든 숫자가 최종 위치에서 최대로 k $(k \geq 1)$ 만큼 떨어져 있는 경우
      - time complexity: $O(kn)$

Comb Sort

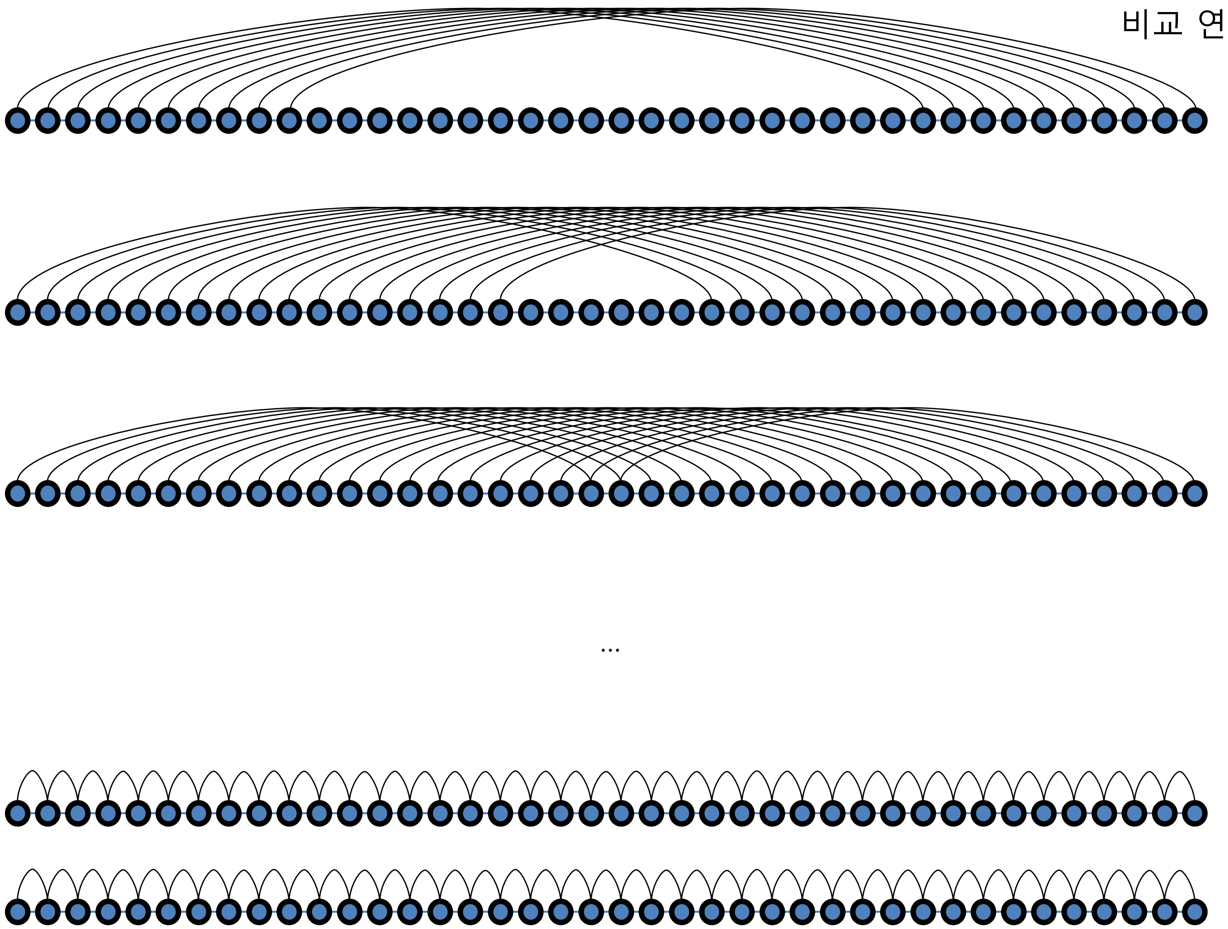
- 기본 아이디어
  - bubble sort에서 거북이(tutle) 데이터를 줄이고자 함
    - 토끼(rabbit) 데이터는 bubble sort에서 문제가 되지 않음
  - bubble sort에서는 인접한 두 데이터의 크기를 비교함.
    - 비교하는 두 데이터의 거리를 gap이라고 함
    - bubble sort에서의 gap은 1임
  - Comb sort
    - gap의 크기를 1보다 크게.
    - bubble sort의 각 pass가 진행됨에 따라 gap의 크기를 줄여감
      - "shrink factor" k 만큼 줄여감
      - gap 크기 : $\left[\frac{n}{k}, \frac{n}{k^2}, \frac{n}{k^3}, \dots, 1\right]$
      - gap의 크기에 따라 comb sort의 효율성이 달라짐
        - » 권장되는 k 값 : 1.3

Comb Sort

비교 연산

비교연산자 '>' 실행 횟수

$n = 40$



$\text{gap} = \left\lfloor \frac{40}{1.3} \right\rfloor = 30$		10
$\text{gap} = \left\lfloor \frac{30}{1.3} \right\rfloor = 23$		17
$\text{gap} = \left\lfloor \frac{23}{1.3} \right\rfloor = 17$		23
$\text{gap} = 13, 10, 7, 5, 3, 2$	…	27, 30, 33, 35, 37, 38
$\text{gap} = \left\lfloor \frac{2}{1.3} \right\rfloor = 1$		39
정렬될때까지 반복		데이터에 따라 다름

Bubble sort (gap=1) 단계
- 위에서 gap>1 인 단계에서 거북이 데이터를 제 위치의 인근에 옮겨 놓았기 때문에 bubble sort에서 실행되는 pass 횟수는 줄어듬

국민대학교 KOOKMIN UNIVERSITY

Comb Sort

- Psuedocode

```
function combsort(array input) is

    gap := input.size // Initialize gap size
    shrink := 1.3 // Set the gap shrink factor
    sorted := false

    loop while sorted = false
        // Update the gap value for a next comb
        gap := floor(gap / shrink)
        if gap ≤ 1 then
            gap := 1
            sorted := true // If there are no swaps this pass, we are done
        end if

        // A single "comb" over the input list
        i := 0
        loop while i + gap < input.size // See Shell sort for a similar idea
            if input[i] > input[i+gap] then
                swap(input[i], input[i+gap])
                sorted := false
                // If this assignment never happens within the loop,
                // then there have been no swaps and the list is sorted.
             end if

             i := i + 1
         end loop
     end loop
end function
```

Why?

- Not Stable
- In-place

비교연산자 '>'

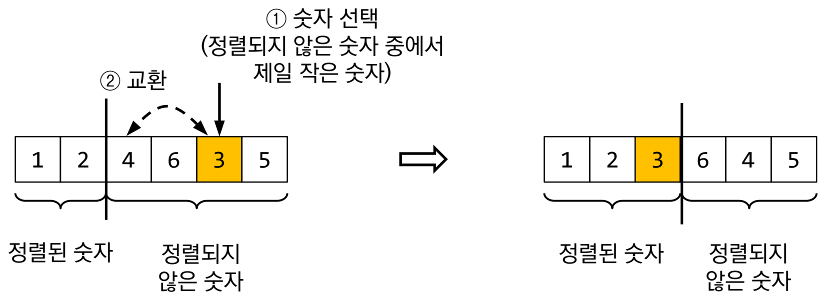
Q) 앞 페이지 그림에서 40개의 정수가
1 2 3 4 .. 39 40
일 때, 비교연산자의 실행 횟수는 몇 번인가?

A) 10 + 17 + 23 + 27 + 30 + 33 + 35 + 37 + 38 + 39 = 289

[ref] https://en.wikipedia.org/wiki/Comb_sort

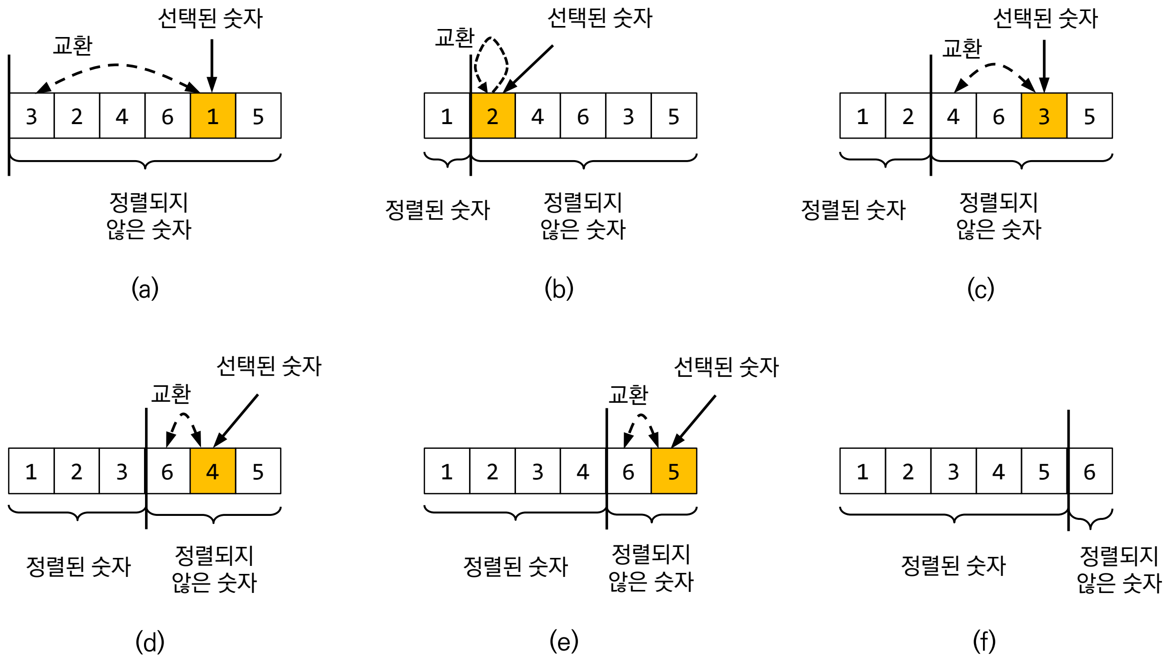
Selection Sort

- 기본 아이디어
  - 정렬 중간 과정에 데이터가 두 부분으로 나누어 짐
    - 왼쪽 : 정렬이 된 데이터
    - 오른쪽 : 정렬이 되지 않은 데이터
      - 오른쪽 데이터에서 제일 작은 데이터를 검색하여 선택하고
      - 오른쪽 데이터의 제일 앞 숫자를 그 숫자를 교환함



Selection Sort

- Selection Sort
  – 각 pass에서 제일 작은 데이터가 선택되어 제 위치로 옮겨지는 과정



Selection Sort

- Psuedo-Code & Analysis

```
InsertionSort( A )
    n = A.size                   // A[0], A[1], …, A[n-1]

    for( i = 0; i < n-1; j++)
        jMin = i

        for( j = i + 1; j < n; j++)  // select the min. of A[i+1], …, A[n-1]
            if( A[j] < A[jMin] )     // 비교 연산
                jMin = j

        if(jMin != i)
            swap(A[jMin], A[j])
```

Why?

- Not Stable
- In-place

i	비교연산자 실행 횟수
	Best/Worst Case
0	n-1
1	n-2
2	n-3
n-2	1
총 횟수	n(n-1)/2

KMU 국민대학교 KOOKMIN UNIVERSITY

Selection Sort

- Why not stable?

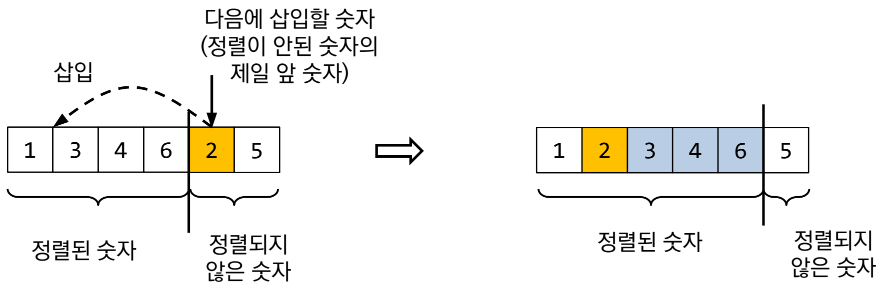
2	2	4	6	1	5	⇨	1	2	4	6	2	5	⇨	…	⇨	1	2	2	4	5	6

KMU 국민대학교 KOOKMIN UNIVERSITY

Insertion Sort

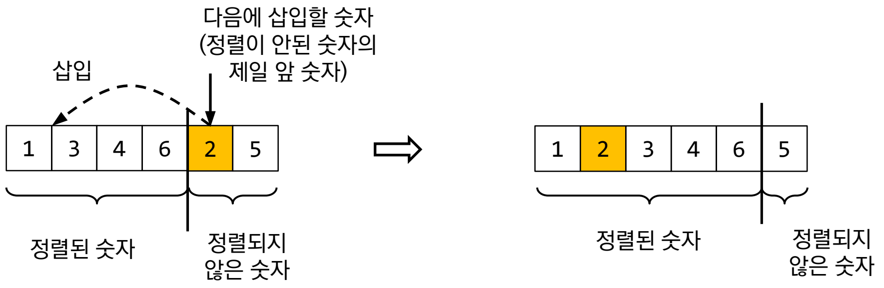
- 기본 아이디어
  – 정렬 중간 과정에 데이터가 두 부분으로 나누어 짐
    - 왼쪽 : 정렬이 된 데이터
    - 오른쪽 : 정렬이 되지 않은 데이터
      – 오른쪽 데이터의 제일 앞 숫자를 그 왼쪽의 정렬된 데이터의 제 위치에 삽입
        » 중간의 모든 데이터는 오른쪽으로 한 칸씩 이동

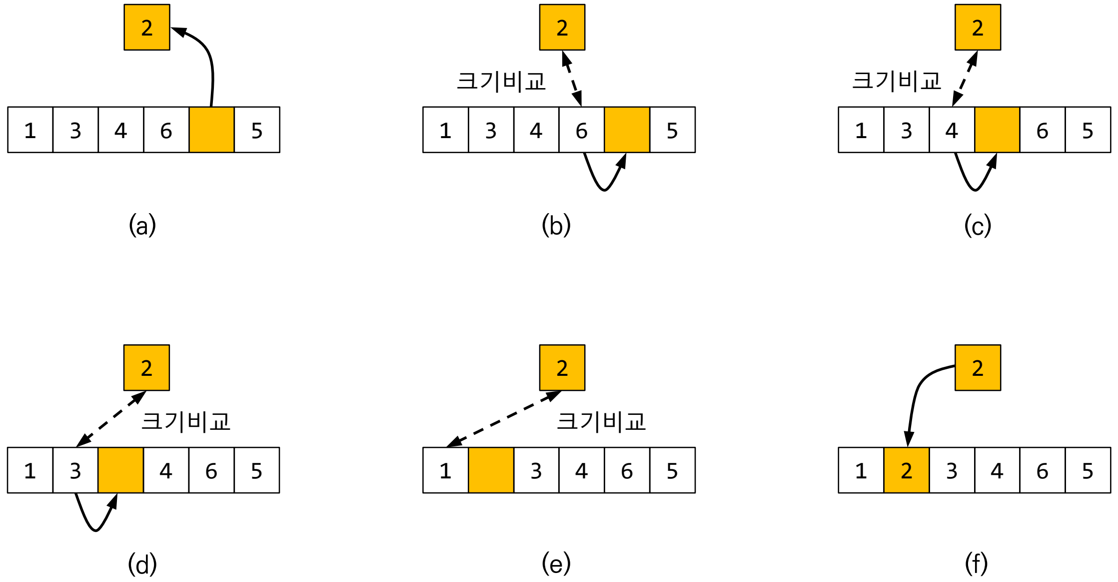
[ref] Introduction to Algorithms, Cormen, Leiserson, Rivest



Insertion Sort

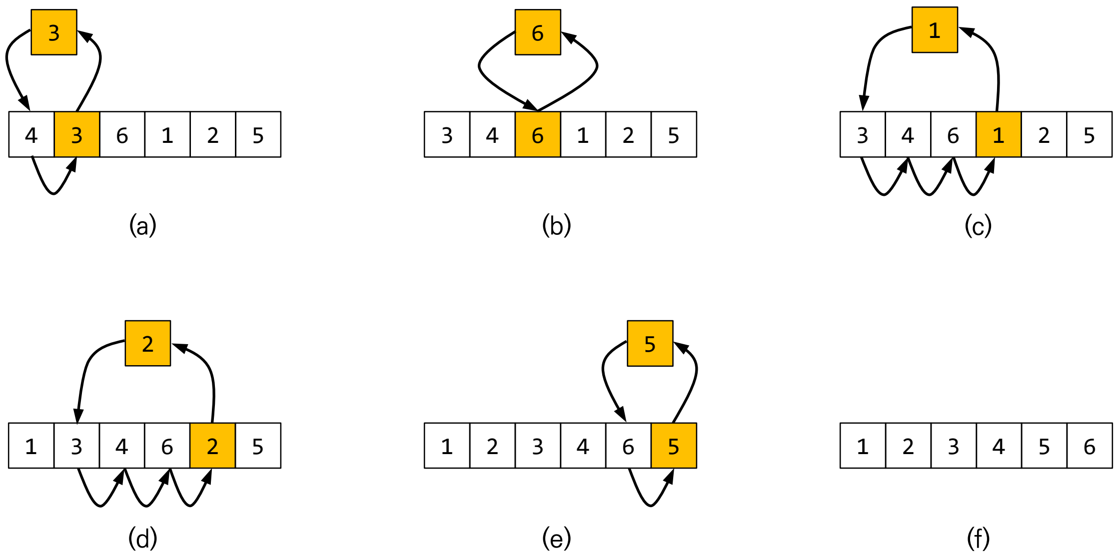
- 한 숫자 inserting 과정





Insertion Sort

- Insertion Sort
  - 각 pass에서 한 데이터가 insert 되는 과정



Insertion Sort

- Psuedo-Code
  - J. Bentley
    - 3-Line Insertion Sort

```
InsertionSort( A )
    n = A.size                    // A[0], A[1], …, A[n-1]

    for( i = 1; i < n; j++)
        for( j = i; j > 0 && A[j-1] > A[j]; j--)
            swap(j-1, j);
```

  - Improvement
    - too many swap operations in Bentley's code

```
InsertionSort( A )
    n = A.size                    // A[0], A[1], …, A[n-1]

    for( i = 1; i < n; j++)
        tmp = A[j]
        for( j = i; j > 0 && A[j-1] > tmp; j--)
            A[j] = A[j-1]
        A[j] = tmp
```

앞 페이지 그림 예시에 해당하는 코드

Analysis

- Analysis
  - 비교 연산자의 실행 횟수는

```
InsertionSort( A )
    n = A.size                     // A[0], A[1], …, A[n-1]

    for( i = 1; i < n; j++)
        for( j = i; j > 0 && A[j-1] > A[j]; j--)
            swap(j-1, j);
```

Why?

- Stable
- In-place

- Best case
  - 입력 데이터가 정렬된 경우

1	2	3	…	n-1	n

- Worst case
  - 입력 데이터가 역순으로 정렬된 경우

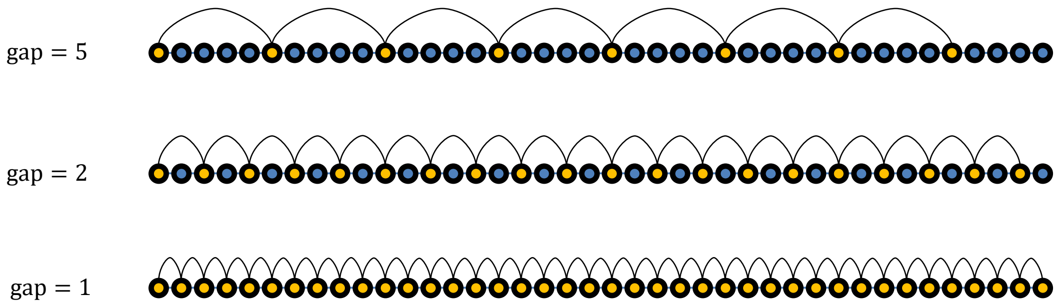
n	n-1	…	3	2	1

i	비교연산자 실행 횟수	
	Best case	Worst Case
1	1	1
2	1	2
3	1	3
n-1	1	n-1
총 횟수	n-1	n(n-1)/2

KMU 국민대학교 KOOKMIN UNIVERSITY

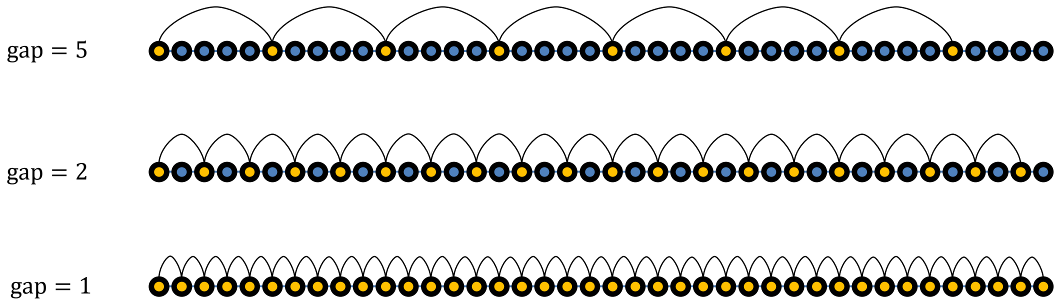
Shell Sort

- 개발자
  - Donald Shell [1959]
- 기본 아이디어
  - Insertion sort를 기본으로 함
  - 서로 멀리 떨어져 있는 숫자들을 정렬(insertion sort로)하기 시작하여
    - 점점 두 숫자들 사이의 거리(gap 이라 부름)을 좁혀서 정렬함
    - 최종적으로 gap=1 이 되면, 원래의 insertion sort를 실행하는 것과 동일함
      - 미리 gap>1 단계에서 거북이 데이터를 목표 위치 근처로 많이 이동시켜 놓았으므로
      - 이 단계 시작 전에는 정렬된 상태에 많이 가까운 상태임



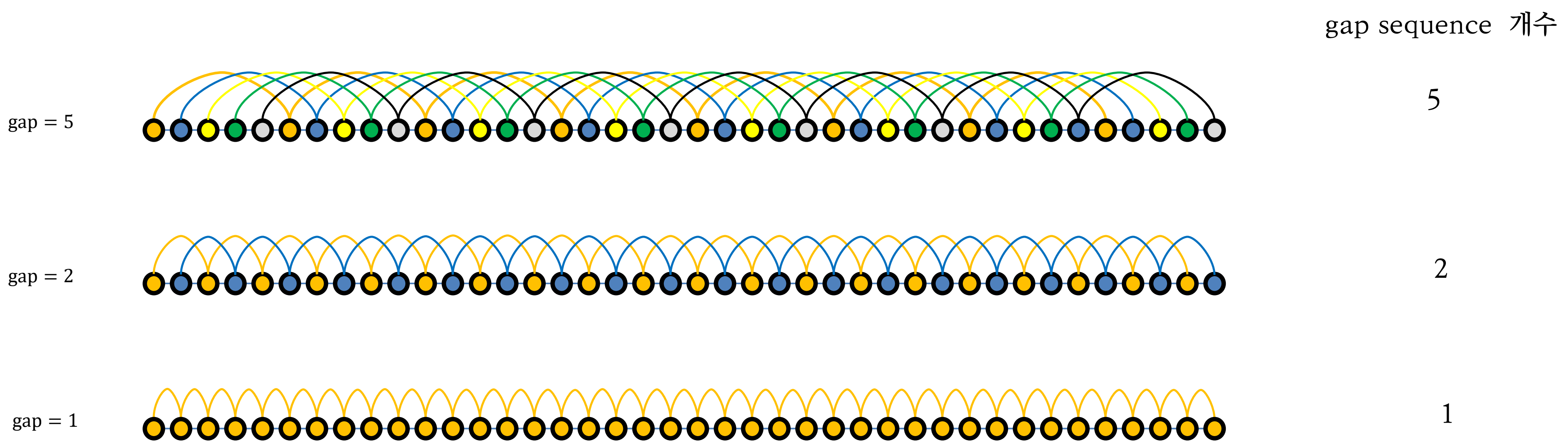
Shell Sort

- Gap, Gap Sequence
  - Gap
    - 정렬할 원소들이 떨어져 있는 거리
    - 처음에는 큰 gap을 사용하여 시작하지만, 점점 작게 만들고, 최종적으로는 1이 됨.
  - Gap Sequence
    - 주어진 시작 숫자부터 gap 만큼 떨어진 모든 숫자들의 집합
    - 같은 gap sequence에 속하는 숫자들을 insertion sort로 정렬함



Shell Sort

- Shell Sort
  - Pass
    - 주어진 gap에 대하여
      - 같은 gap을 가지는 모든 gap sequence에 insertion sort를 실행함



Shell Sort

- Shell Sort
  - gap의 크기는 어떻게 결정되는가?
    - gap의 크기에 따라 Shell sort의 효율성이 달라짐.

OEIS	General term ($k \geq 1$)	Concrete gaps	Worst-case time complexity	Author and year of publication
	$\left\lfloor \frac{N}{2^k} \right\rfloor$	$\left\lfloor \frac{N}{2} \right\rfloor, \left\lfloor \frac{N}{4} \right\rfloor, \ldots, 1$	$\Theta\left(N^2\right)$ [e.g. when $N = 2^p$]	Shell, 1959[4]
	$2\left\lfloor \frac{N}{2^{k+1}} \right\rfloor + 1$	$2\left\lfloor \frac{N}{4} \right\rfloor + 1, \ldots, 3, 1$	$\Theta\left(N^{\frac{3}{2}}\right)$	Frank & Lazarus, 1960[8]
A000225	$2^k - 1$	$1, 3, 7, 15, 31, 63, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Hibbard, 1963[9]
A083318	$2^k + 1$, prefixed with 1	$1, 3, 5, 9, 17, 33, 65, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Papernov & Stasevich, 1965[10]
A003586	Successive numbers of the form $2^p 3^q$ (3-smooth numbers)	$1, 2, 3, 4, 6, 8, 9, 12, \ldots$	$\Theta\left(N \log^2 N\right)$	Pratt, 1971[1]
A003462	$\frac{3^k - 1}{2}$, not greater than $\left\lceil \frac{N}{3} \right\rceil$	$1, 4, 13, 40, 121, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Knuth, 1973,[3] based on Pratt, 1971[1]
A102549	Unknown (experimentally derived)	$1, 4, 10, 23, 57, 132, 301, 701$	Unknown	Ciura, 2001[15]

[ref] https://en.wikipedia.org/wiki/Shell_sort

Shell Sort

- Pseudo-Code

```
ShellSort( A )
    n = A.size                  // A[0], A[1], …, A[n-1]
    shrinkRatio = 2
    gap = A.size / shrinkRatio

    while( gap > 0 )
    {
        for( i = gap; i < n; i++)
        {
            tmp = A[i]
            for( j = i; (j >= gap) && (A[j - gap] > tmp); j -= gap )
                A[j] = A[j - gap]
            A[j] = tmp
         }
         gap /= shrinkRatio
     }
```

Why?

- Not Stable
- In-place

비교연산자 '>'

이 비교 연산자의 실행 횟수를 계산하시오. (과제)

(주의) 이 Pseudo-Code는 short-circuit evaluation이 실행됨
즉, 다음 수식

```
(j >= gap) && (A[j - gap] > tmp)
```

에서 (j >= gap) 이 false 이면 뒤쪽의 비교연산자를
포함하는 수식은 계산하지 않음

Shell Sort

- Example
  - n : 8, A = {8, 7, 6, 5, 4, 3, 2, 1}

gap	A	'<' 연산횟수
입력	8 7 6 5 4 3 2 1	0
4	4 7 6 5 8 3 2 1	1
	4 3 6 5 8 7 2 1	1
	4 3 2 5 8 7 6 1	1
	4 3 2 1 8 7 6 5	1
6	2 3 4 1 8 7 6 5	1
	2 1 4 3 8 7 6 5	1
	2 1 4 3 8 7 6 5	1
	2 1 4 3 8 7 6 5	1
	2 1 4 3 6 7 8 5	2
	2 1 4 3 6 5 8 7	2
1	1 2 4 3 6 5 8 7	1
	1 2 4 3 6 5 8 7	1
	1 2 3 4 6 5 8 7	2
	1 2 3 4 6 5 8 7	1
	1 2 3 4 5 6 8 7	2
	1 2 3 4 5 6 8 7	1
	1 2 3 4 5 6 7 8	2
총		22

프로그래밍 과제