

Algorithm: Homework #1

Due Date: April 16, 2021

1. Consider the following recurrence relation for generating a sequence $\{a_n\}$, $n \geq 2$:

$$a_n = a_{n-1} + a_{n-2} + 1, \text{ where } a_0 = a_1 = 1.$$

We want to write some algorithms that input an non-negative integer n and then output a_n .

For each of the following cases, give a pseudo-code and show its time complexity.

(1) A recursive algorithm of which the time complexity is exponential in n , i.e., $O(r^n)$, where $r > 1$.

(2) An iterative algorithm of which the time complexity is linear in n , i.e., $O(n)$.

(Bonus1)* A recursive algorithm of which the time complexity is linear in n , i.e., $O(n)$.

(Bonus2)* An algorithm of which the time complexity is logarithmic in n , i.e., $O(\log n)$.

(1)

```
long long Babol (int n) {
    if (n < 2) return 1;
    return Babol (n-1) + Babol (n-2) + 1;
}
```

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

$$\leftarrow T(n) - T(n-1) - T(n-2) = 1$$

$$(r^2 - r - 1)(r-1) = 0 \rightarrow$$

$$\begin{aligned} r_1 &= \frac{1+\sqrt{5}}{2} \\ r_2 &= \frac{1-\sqrt{5}}{2} \\ r_3 &= 1 \end{aligned}$$

$$T(n) = C_1 r_1^n + C_2 r_2^n + C_3$$

$$T(0) = C_1 + C_2 + C_3 = 1$$

$$T(1) = C_1 \cdot \frac{1+\sqrt{5}}{2} + C_2 \cdot \frac{1-\sqrt{5}}{2} + C_3 = 1$$

$$T(2) = C_1 \cdot \frac{3+\sqrt{5}}{2} + C_2 \cdot \frac{3-\sqrt{5}}{2} + C_3 = 3$$

$$T(1) - T(0) = \frac{\sqrt{5}}{2} C_1 - \frac{\sqrt{5}}{2} C_2 = 0$$

$$\leftarrow C_1 = C_2$$

$$T(2) - T(1) = \frac{2}{2} C_1 + \frac{2}{2} C_2 = C_1 + C_2 = 2$$

$$\boxed{C_1 = C_2 = 1, C_3 = -1}$$

$$T(n) = \left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n - 1 \in O(r^n)?$$

$$n \geq 1 \text{ 인 모든 } n \text{에 대해서 } \left(\left| \frac{1+\sqrt{5}}{2} \right| + \left| \frac{1-\sqrt{5}}{2} \right| \right)^n + 1 \geq \left(\frac{1+\sqrt{5}}{2} \right)^n + \left(\frac{1-\sqrt{5}}{2} \right)^n$$

$$\left(\frac{1+\sqrt{5}}{2} \right)^n + \left(\frac{1-\sqrt{5}}{2} \right)^n - 1 \leq \left(\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2} \right)^n = (\sqrt{5})^n$$

\therefore 따라서, $n \geq N$ 인 모든 n 에 대해서 $T(n) \leq cr^n$ 이 성립하는 $c=1, N=1, r=\sqrt{5} > 1$
존재하기 때문에 시간복잡도 $T(n)$ 은 $O(r^n)$ 에 속한다.

```

(2) long long Babo2(int n) {
    long long *arr = new long long[n + 1];
    arr[0] = arr[1] = 1;
    for (int i = 2; i <= n; i++)
        arr[i] = arr[i - 1] + arr[i - 2] + 1;
    return arr[n];
}

```

basic operation 을 assignment operation 으로 보면 시간 복잡도는 $T(n) = n+1$ 가 된다.

$$T(n) = n + 1 \in O(n) ?$$

$$n \geq 1 \text{ 인 모든 } n \text{에 대해서 } n+n \geq n+1$$

$$n+1 \leq n+n = 2n$$

\therefore 따라서, $n \geq N$ 인 모든 n 에 대해서 $n+1 \leq cn$ 이 성립하는 $c=2, N=1$ 이 존재하기 때문에 시간 복잡도 $T(n)$ 은 $O(n)$ 에 속한다.

(Bonus 1)

```

long long *arr = new long long[n+1]();
arr[0] = arr[1] = 1;
}

long long Babo3(int n) {
    if (arr[n]) return arr[n];
    return arr[n] = Babo3(n-1) + Babo3(n-2) + 1;
}

```

$$T(n) = \begin{cases} 1 & (\text{If you did } T(n) \text{ already}) \\ T(n-1) + T(n-2) + 1 & (\text{Otherwise}) \end{cases}$$

$$T(n) \text{은 시작이므로 } T(n) = T(n-1) + T(n-2) + 1 \text{ 이다.}$$

여기서 $T(n-1)$ 을 수행한 뒤 $T(n-2)$ 를 탈テン더

$$\begin{aligned} T(n-1) \text{ 를 하면서 } T(n-2) \text{ 를 이미 했기 때문에 } T(n) &= T(n-1) + T(n-2) + 1 \\ &= T(n-1) + 1 + 1 \text{ 이 된다.} \end{aligned}$$

$$\begin{aligned} T(n) &= T(n-1) + 2 \\ &= T(n-2) + 2 + 2 \\ &\vdots \\ &= T(0) + \underbrace{2 + \dots + 2}_{n+1} \\ &= \boxed{2n+1} \end{aligned}$$

$$T(n) = 2n+1 \in O(n) ?$$

$$n \geq 1 \text{ 인 모든 } n \text{에 대해서 } 2n+n \geq 2n+1$$

$$2n+1 \leq 2n+n = 3n$$

\therefore 따라서, $n \geq N$ 인 모든 n 에 대해서 $2n+1 \leq cn$ 이 성립하는 $c=3, N=1$ 이 존재하기 때문에 시간 복잡도 $T(n)$ 은 $O(n)$ 에 속한다.

(Bonus 2)

```

long long Babo4 (int n) {
    matrix A = [[1 0 0], [0 1 0], [0 0 1]], B = [[1 1 1], [1 0 0], [0 0 1]]';
    for (--n; n > 0; n /= 2) {
        if (n & 1) A = A * B;
        B = B * B;
    }
    return A[0][0] + A[0][1] + A[0][2];
}

```

주어진 점화식 $a_n = a_{n-1} + a_{n-2} + 1$ 을 단순한 두개의 행렬의 곱으로 변환해보자.

$$\begin{aligned}
a_n &= a_{n-1} + a_{n-2} + 1 \\
&= 1 \times a_{n-1} + 1 \times a_{n-2} + 1 \times 1 \\
&= [1 1 1] \times \begin{bmatrix} a_{n-1} \\ a_{n-2} \\ 1 \end{bmatrix}
\end{aligned}$$

위 행렬의 곱 중에서 좌측항만 적절히 바꾸면 a_{n-1} , 1 을 각각 만들 수 있다.

$$\begin{array}{c|c}
a_{n-1} = 1 \times a_{n-1} + 0 \times a_{n-2} + 0 \times 1 & 1 = 0 \times a_{n-1} + 0 \times a_{n-2} + 1 \times 1 \\
= [1 0 0] \times \begin{bmatrix} a_{n-1} \\ a_{n-2} \\ 1 \end{bmatrix} & = [0 0 1] \begin{bmatrix} a_{n-1} \\ a_{n-2} \\ 1 \end{bmatrix}
\end{array}$$

위에서 얻은 a_n , a_{n-1} , 1 을 순서대로 쌓아서 3×1 행렬을 만들자.

$$\begin{aligned}
\begin{bmatrix} a_n \\ a_{n-1} \\ 1 \end{bmatrix} &= \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} a_{n-1} \\ a_{n-2} \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} a_{n-2} \\ a_{n-3} \\ 1 \end{bmatrix} \\
&\vdots \\
&= \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}^{n-1} \times \begin{bmatrix} a_1 \\ a_0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}^{n-1} \times \begin{bmatrix} 1 \\ 1 \end{bmatrix}
\end{aligned}$$

이로써 a_n 은 행렬 $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ 을 $n-1$ 번 거듭제곱한 결과 값 중 첫번째 행에 있는 요소의 합이 된다.
(다음 장에 계속)

하지만 행렬을 $n-1$ 번 곱하도록 하는 것은 $O(\log n)$ 의 시간복잡도를 가질 수 없다.

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{일때} \quad n=101 \quad \text{일 경우} \quad A^{100} \text{ 을 구해야 하는데} \quad A^{100} = A^{64+32+4} = A^{64} \times A^{32} \times A^4 \text{ 이다.}$$

100을 이진수로 표현하면 0110 이00 이다.
 $\uparrow \uparrow \uparrow \uparrow$
 64 32 8

따라서 $n-1$ 을 2의 거듭제곱의 합으로 나타냈을 때 각 요소로 A의 거듭제곱을 구한다.
 구한 모든 행렬을 곱하면 A^{n-1} 을 $O(\log n)$ 속도로 얻을 수 있다.

이제 함수의 시간 복잡도가 $O(\log n)$ 인지를 수학적으로 증명해보자.

Worst Case는 $n=2^k$ 일 때 A^{n-1} 을 구해야 하는데 A가 거듭제곱 될 때마다 결과에 곱해지므로 n 을 2^k 로 두고 구해보자.

$$n=2^k \quad (k>0) \rightarrow k=\lg n$$

$$\left. \begin{array}{l} W(n)=W\left(\frac{n}{2}\right)+1 \\ T_k=W(n)=W(2^k) \\ T_{k-1}=W(2^{k-1})=W\left(\frac{n}{2}\right) \end{array} \right\} \rightarrow T_k=T_{k-1}+1 \rightarrow T_k-T_{k-1}=1$$

C.E IS $(r-1)(r-1)=0$
 $r=1$

$$T_k=C_1 k^k + C_2 k^k, \quad T_k=W(2^k)=W(n)=C_1 + C_2 \cdot \lg n$$

$$\left. \begin{array}{l} W(1)=C_1+C_2 \cdot 0=1, \quad (C_1=1) \\ W(2)=C_1+C_2=3, \quad (C_2=2) \end{array} \right\} \rightarrow W(n)=1+2 \lg n$$

$$W(n)=1+2 \lg n \in O(\log n)?$$

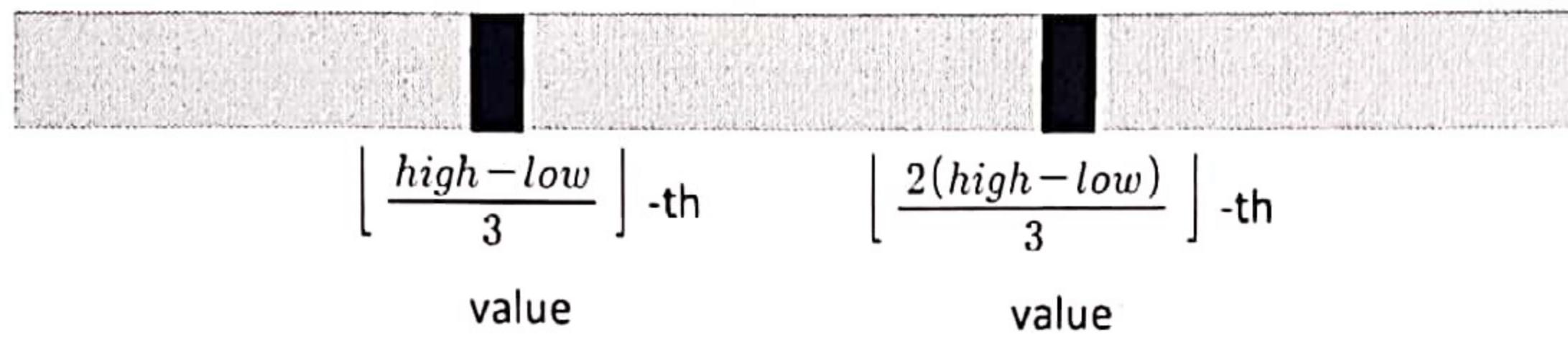
$$n \geq 2 \text{ 인 모든 } n \text{에 대하여 } \lg n \geq 1 \xrightarrow{+2 \lg n} 2 \lg n + \lg n \geq 1 + 2 \lg n$$

$$\underbrace{1+2 \lg n}_{=g(n)} \leq 3 \lg n = (3 \lg 10) \cdot \frac{\lg n}{\lg 10} = (3 \lg 10) \cdot \frac{\lg n}{N} \quad (n \geq 2)$$

\therefore 따라서, $n \geq N$ 인 모든 n 에 대해서 $1+\lg n \leq c \lg n$ 이 성립하는

$c=3 \lg 10, N=2$ 가 존재하기 때문에 시간복잡도 $W(n)$ 은 $O(\log n)$ 에 속한다.

2. (Ternary Search) We have already studied a binary search algorithm in the lecture. Now (1) give the pseudo-code of a ternary search algorithm, in which about $n/3$ values remain after at most two comparisons, and (2) compute its worst-case time complexity. Assume that input array with n values is sorted in non-decreasing order.



```
(1) int Ternary Search (int n, int *arr, int key) {
    int left = 0, right = n - 1, mid1, mid2;
    while (left <= right) {
        if (key == arr[mid1 = (left * 2 + right) / 3])
            return mid1;
        else if (key == arr[mid2 = (left + right * 2) / 3])
            return mid2;
        else if (key < mid1) // Comparison 1
            right = mid2 - 1;
        else if (key > mid2) // Comparison 2
            left = mid1 + 1;
        else {
            right = mid2 - 1;
            left = mid1 + 1;
        }
    }
    return -1;
}
```

- (2)
- Basic Operation: Comparison 1 and Comparison 2
 - Input Size : n , number of item array
 - Assumption : $n = 3^k$ for some integer $k \geq 0$

◦ Worst Case는 $key > t$
가운데 있어서 Comparison
을 모두 수행하는 경우임.

$$W(n) = W\left(\frac{n}{3}\right) + 2 \quad \text{for } n > 1, W(1) = 1$$

$$= W\left(\frac{n}{3^2}\right) + 2 + 2$$

:

$$= W\left(\frac{n}{3^k}\right) + 2 + \dots + 2$$

$$= 1 + 2k \quad \underbrace{\quad}_{k \geq 1}$$

$$= 1 + 2 \log_3 n$$

$$\therefore W(n) = 1 + 2 \log_3 n$$

3. (Three-way MergeSort) We have already studied a balanced two-way MergeSort in the lecture. Now (1) give the pseudo-code of a balanced three-way MergeSort and (2) compute its worst-case and best-case time complexities.

```

(1) [ int *arr = new int[n];
      int *tmp = new int[n]; ] }

void Merge(int left, int right) {
    int endI = (left * 2 + right) / 3;
    int endJ = (left + right * 2) / 3;

    int i = left, j = endI + 1, k = endJ + 1, l = left;
    while (i <= endI && j <= endJ && k <= right) {
        if (arr[i] <= arr[j] && arr[i] <= arr[k]) // Comparison 1
            tmp[l++] = arr[i++];
        else if (arr[j] <= arr[i] && arr[j] <= arr[k]) // Comparison 2
            tmp[l++] = arr[j++];
        else
            tmp[l++] = arr[k++];
    }
}

if (i > endI) { i = k; endI = right; }
else if (j > endJ) { j = k; endJ = right; }
while (i <= endI && j <= endJ)
    tmp[l++] = arr[i] <= arr[j] ? arr[i++] : arr[j++];
// Comparison 3

k = i > endI ? j : i;
while (l <= right)
    tmp[l++] = arr[k++];

for (i = left; i <= right; i++)
    arr[i] = tmp[i];
}

void ThreeWayMergeSort (int left, int right) {
    if (left >= right) return;

    int mid1 = (left * 2 + right) / 3;
    int mid2 = (left + right * 2) / 3;

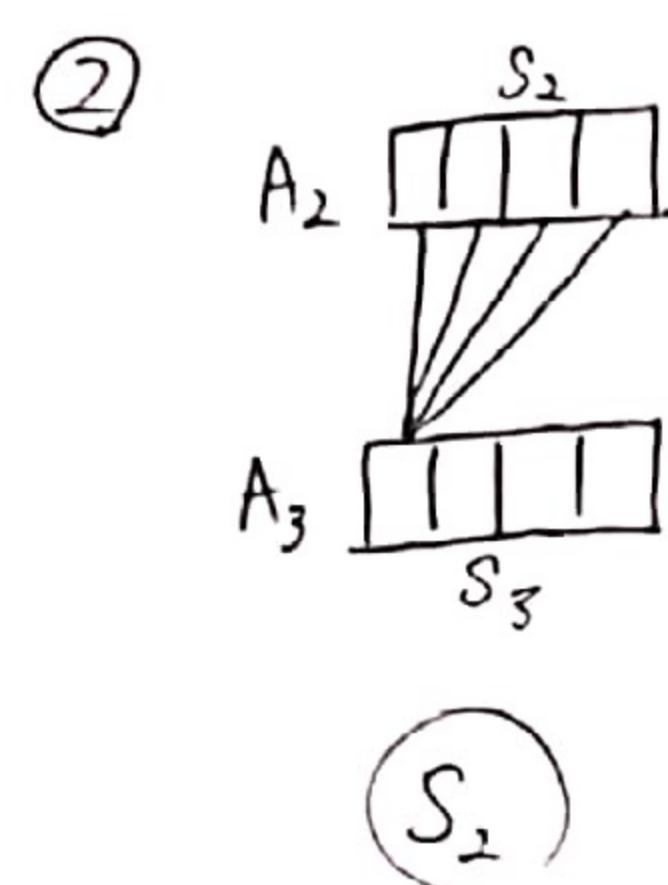
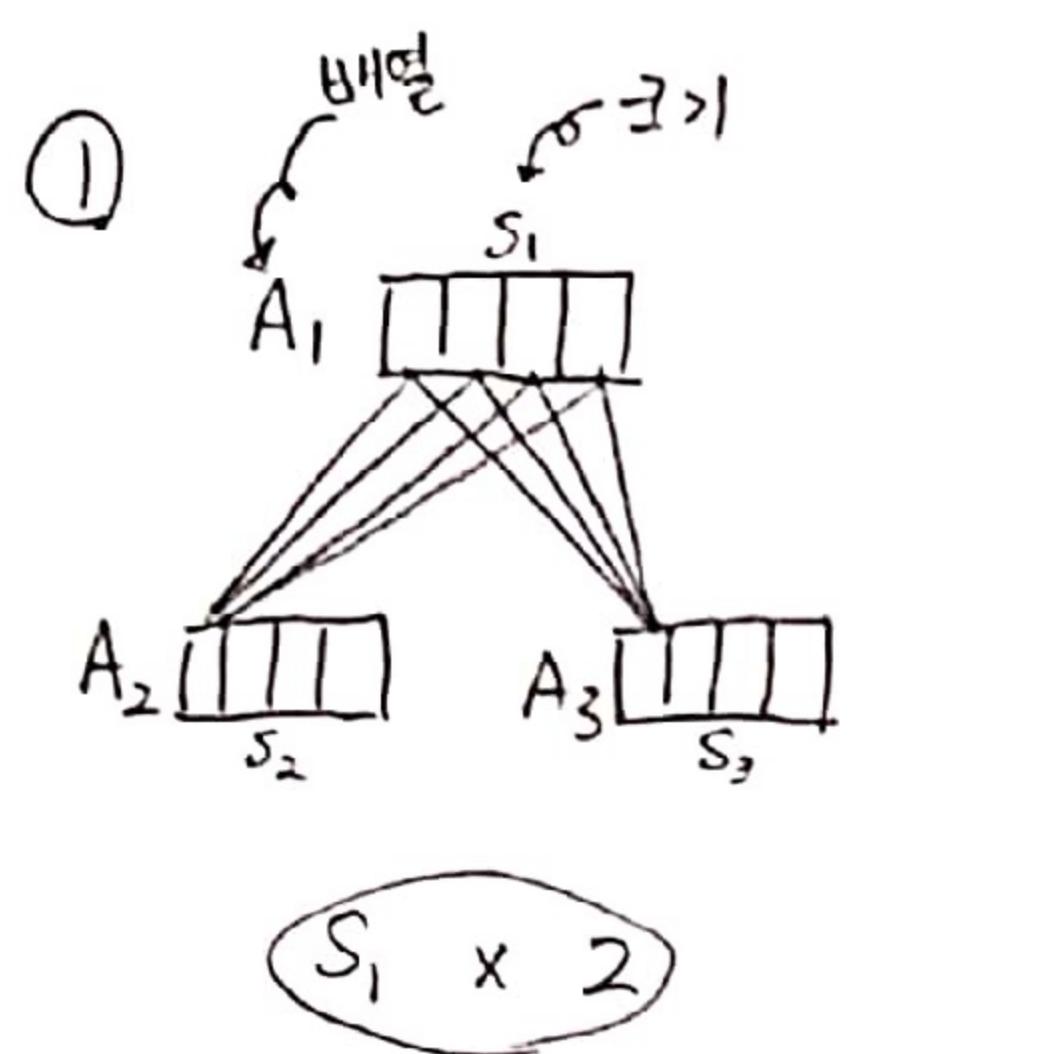
    ThreeWayMergeSort (left, mid1);
    ThreeWayMergeSort (mid1 + 1, mid2);
    ThreeWayMergeSort (mid2 + 1, right);
    Merge (left, right);
}

```

}

(2) < Best Case Time Complexity of Merge >

- Basic Operation : Comparison 1, 2, 3
- Input Size : 2 array of n size

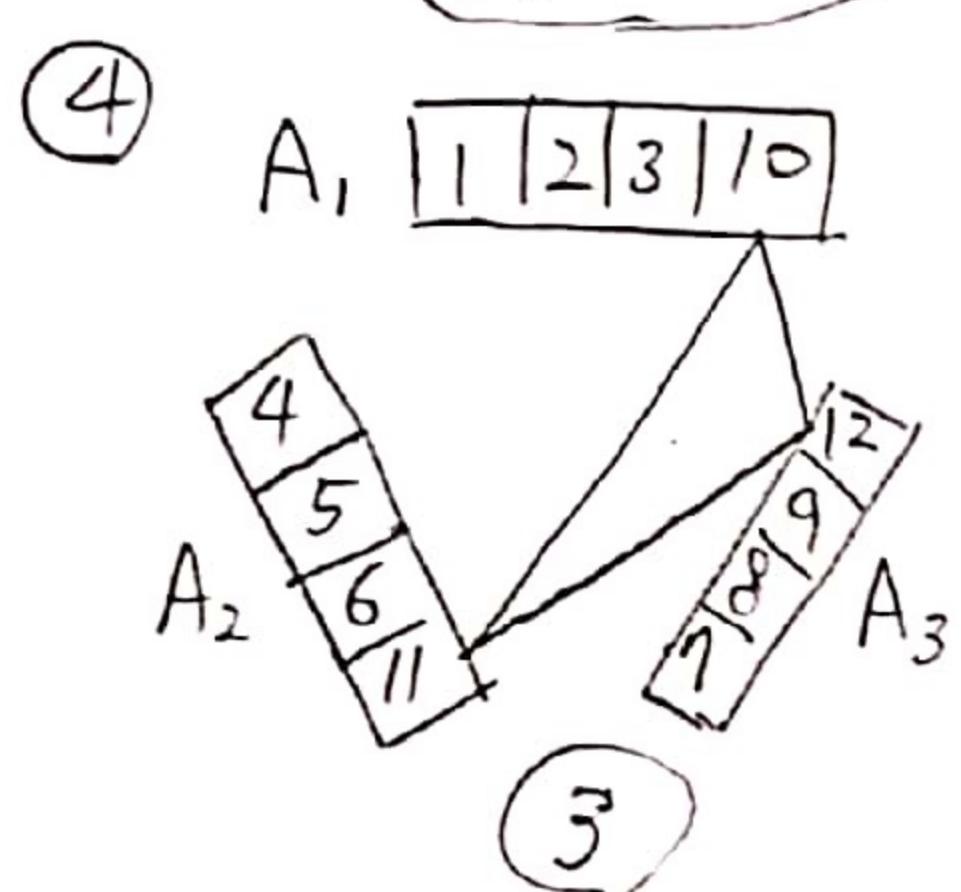
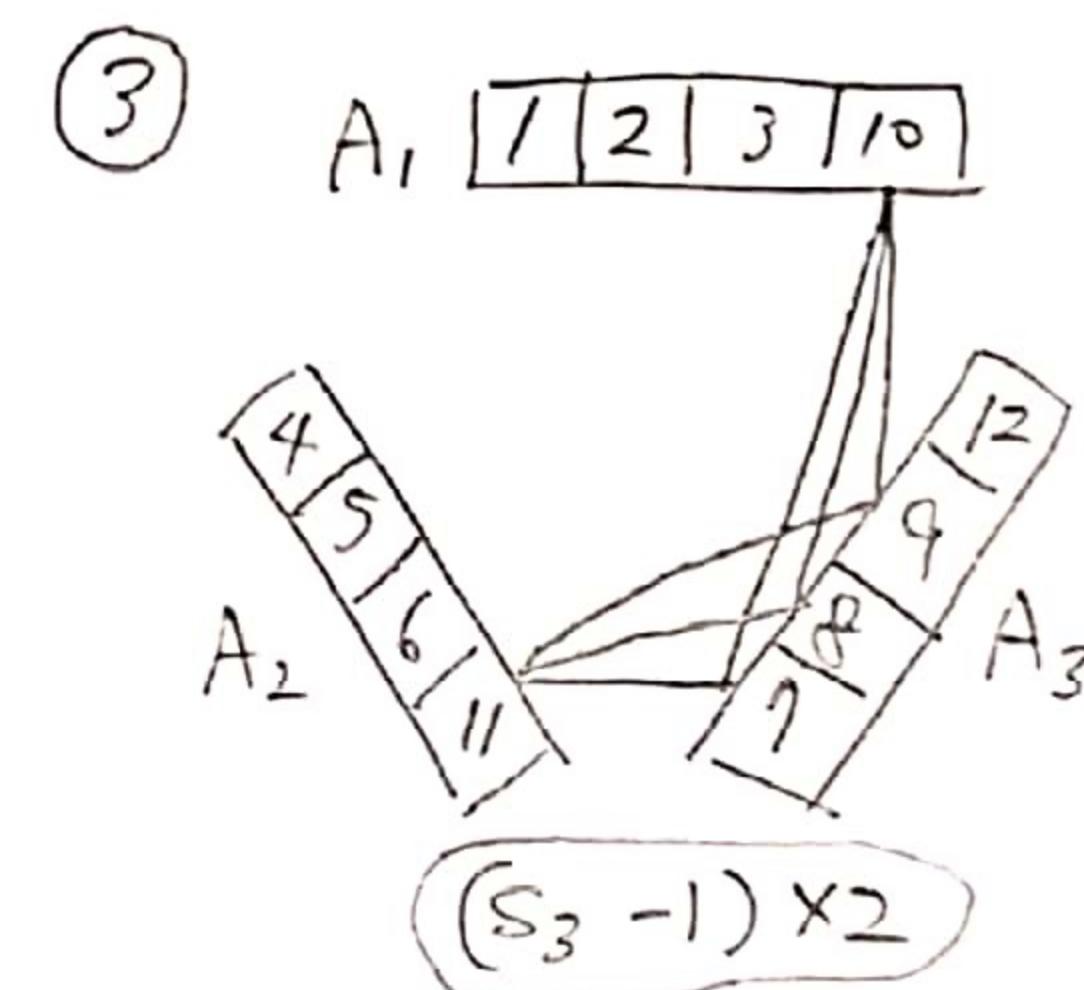
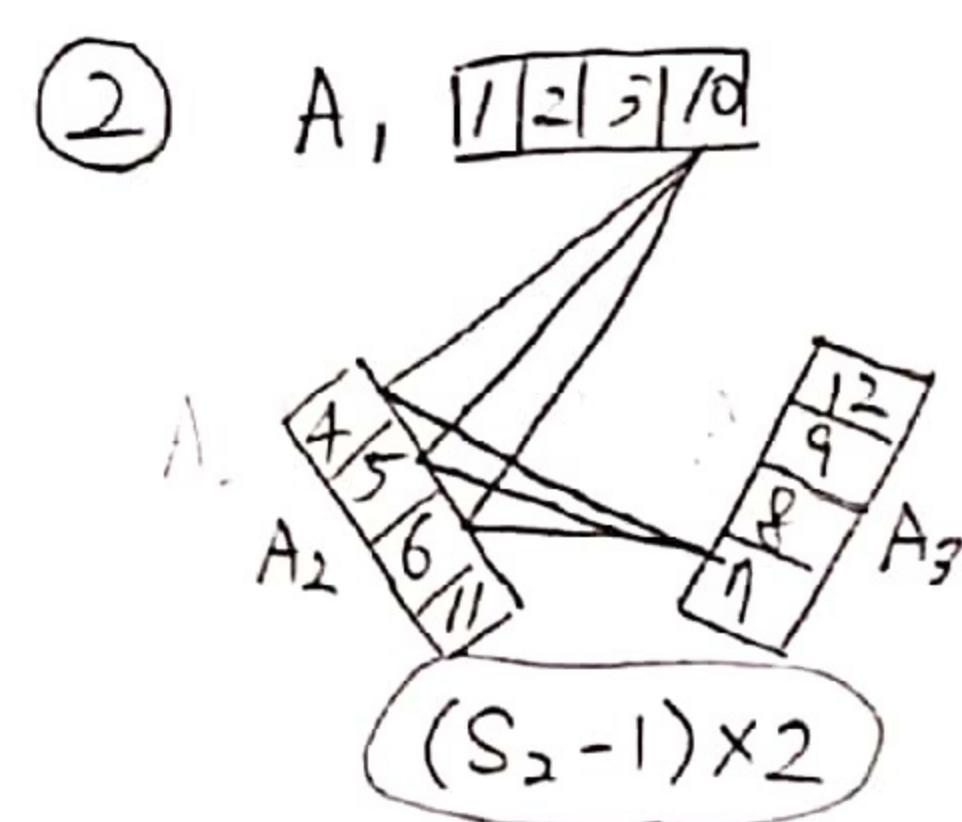
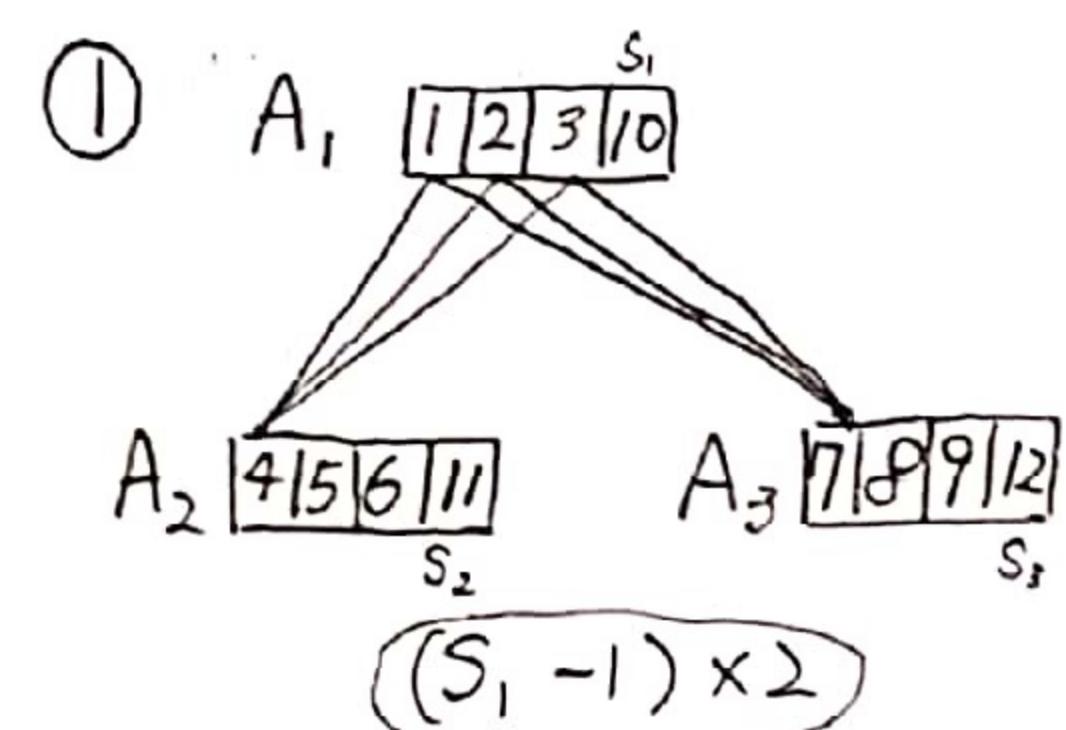


$$\Rightarrow B(S_1, S_2, S_3) = \min(S_1, S_2, S_3) \times 2 + \text{mid}(S_1, S_2, S_3)$$

$$\boxed{B(S_1, S_2, S_3) = S_1 + S_2 + S_3}$$

<Merge의 Best Case>

< Worst Case Time Complexity of Merge >



$$\Rightarrow W(S_1, S_2, S_3) = 2(S_1 - 1) + 2(S_2 - 1) + 2(S_3 - 1) + 3 = 2(S_1 + S_2 + S_3) - 6 + 3$$

$$\boxed{W(S_1, S_2, S_3) = 2(S_1 + S_2 + S_3) - 3}$$

<Merge의 Worst Case>

< Best Case Time Complexity of Three-way MergeSort >

$$B(n) = B(s_1) + B(s_2) + B(s_3) + B(s_1, s_2, s_3), \quad n = 3^k \quad (k \geq 0) \quad \text{일 때} \quad s_1 = s_2 = s_3 = \frac{n}{3}$$

$$\text{따라서 } B(n) = 3 B\left(\frac{n}{3}\right) + n, \quad B(1) = 0$$

$$\begin{aligned} \text{Let } T_k &= B(n) = B(3^k) \rightarrow T_{k-1} = B(3^{k-1}) = B\left(\frac{n}{3}\right) \\ \Rightarrow T_k &= 3T_{k-1} + 3^k \end{aligned}$$

$$\text{That is, } (1) T_k + (-3)T_{k-1} = 3^k = 3^k (1)$$

$$\text{The C.E is } (r-3)(r-3) = 0$$

$$T_k = C_1 3^k + C_2 k 3^k \rightarrow \text{Since } T_k = B(3^k), \quad B(n) = C_1 n + C_2 (\log_3 n) n$$

$$B(1) = C_1 = 0$$

$$B(3) = 3C_1 + 3C_2 = 3$$

$$\Leftrightarrow C_1 = 0, C_2 = 1$$

$$\therefore B(n) = n \log_3 n$$

(다음장계속)

<3wayMergeSort Best Case>

〈Worst Case Time Complexity of Three-way Merge Sort〉

$$W(n) = W(s_1) + W(s_2) + W(s_3) + W(s_1, s_2, s_3), \quad n=3^k \quad (k \geq 0) \quad \text{일때} \quad s_1 = s_2 = s_3 = \frac{n}{3}$$

따라서 $W(n) = 3W\left(\frac{n}{3}\right) + 2n - 3, \quad W(1) = 0$

$$\text{let } T_k = W(n) = W(3^k) \rightarrow T_{k-1} = W(3^{k-1}) = W\left(\frac{n}{3}\right)$$

$$\Rightarrow T_k = 3T_{k-1} + 2 \cdot 3^k - 3$$

That is, $(1) T_k + (-3)T_{k-1} = 2 \cdot 3^k - 3 = 3^k(2) + 1^k(-3)$

The C.E is $(r-3)(r-3)(r-1) = 0$

$$T_k = C_1 3^k + C_2 k 3^k + C_3 1^k \rightarrow \text{since } T_k = W(3^k), \quad W(n) = C_1 n + C_2 (\log_3 n) n + C_3$$

$$W(1) = C_1 + C_3 = 0$$

$$W(3) = 3C_1 + 3C_2 + C_3 = 3$$

$$W(9) = 9C_1 + 18C_2 + C_3 = 24.$$

$$W(3) - W(1) = 2C_1 + 3C_2 = 3$$

$$W(9) - W(3) = 6C_1 + 15C_2 = 21 \rightarrow 2C_1 + 5C_2 = 7$$

$$\Rightarrow 2C_2 = 4,$$

$$C_2 = 2$$

$$C_1 = -\frac{3}{2}$$

$$C_3 = \frac{3}{2}$$

$$\boxed{\therefore W(n) = 2n \log_3 n - \frac{3}{2}n + \frac{3}{2}}$$

Therefore the answer is

$$\boxed{B(n) = n \log_3 n \\ W(n) = 2n \log_3 n - \frac{3}{2}n + \frac{3}{2}}$$

4. (FoolSort) Consider the following pseudo-code of a sorting algorithm called **FoolSort**.

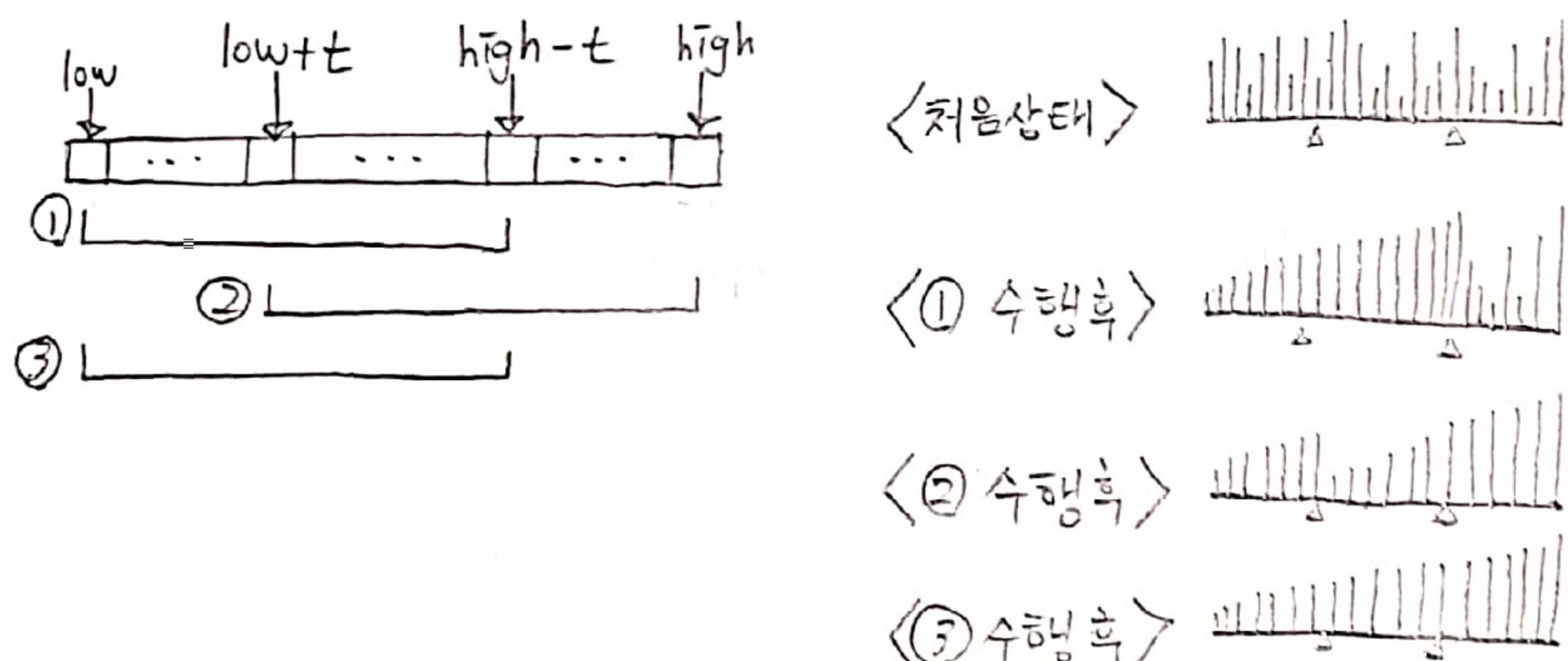
```

FoolSort(array  $L$ , low := 0, high := length( $L$ ) - 1)
    if high > low then
        if  $L[\text{high}] < L[\text{low}]$  then swap ( $L[\text{low}], L[\text{high}]$ );
        if (high - low) > 1 then
             $t := (\text{high} - \text{low} + 1)/3;$ 
            ① FoolSort( $L$ , low, high- $t$ );
            ② FoolSort( $L$ , low+ $t$ , high);
            ③ FoolSort( $L$ , low, high- $t$ );
    return  $L$ ;

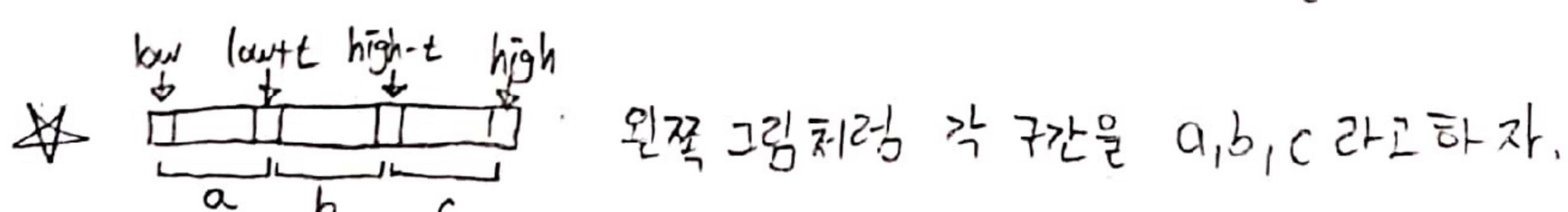
```

- (1) Show that this algorithm always produces a correct sorting result.
- (2) Assume that the basic operation is the comparison between $L[\text{low}]$ and $L[\text{high}]$ given in the second line (underlined part). Let $T(n)$ be the time complexity function of the above algorithm, where n is the length of array L . Give a recurrence relation to compute $T(n)$.
- (3) Solve the recurrence relation for (1) and give the order of $T(n)$. You may assume that $T(1) = 0$ and $T(2) = 1$.

(1) 우선 swap으로 인해서 배열의 처음과 끝을 비교해서 정렬이 된다는 것을 알수있다.
 그럼 이 swap이 이루어 지기위한 Divide and Conquer 부분인
 FoolSort를 3번 재귀호출하는 부분을 막대로 알아보자.



①이 수행될때도 $\frac{2}{3}$ 씩쪼개져서 3번의 FoolSort를 재귀적으로 또 호출된다.
 그래서 ①만 수행 된 것으로 보았을 때 low에서 high- t 까지 정렬된 것이다.



②수행후에서 $L[\text{high}-t+1] < L[\text{low}+t]$ 일정우 Sort가 되지 않는다.
 하지만 이 현상이 일어날 수 있는데 그 이유는 ①수행후에서 보면 알수있다.

만약, c구간 모든 요소가 $L[\text{low}+t]$ 보다 작을경우 c구간 모든 요소는 b구간으로 이동하고

b구간의 요소의 수는 c구간의 요소의 수보다 크거나 같으므로 원래 b구간의 최소값 ($= L[\text{low}+t+1]$)
 보다 크거나 같은 값이 $L[\text{high}-t+1]$ 에 위치하게된다.

- 9 - |따라서 정렬이 잘 일어난다|

(2) $t = \lfloor n/3 \rfloor$

$$T(n) = 1 + T(n-t) + T(n-t) + T(n-t)$$

Let $n = 3^k$ ($k > 0$), $t = n/3$

$$\boxed{\therefore T(n) = 3T\left(\frac{2}{3}n\right) + 1}$$

(3) Let $n = (\frac{3}{2})^k$ ($k > 0$), $S_k = T(n) = T\left((\frac{3}{2})^k\right) \rightarrow S_{k-1} = T\left((\frac{3}{2})^{k-1}\right) = T\left(\frac{2}{3}n\right)$

$$S_k = 3S_{k-1} + 1 \quad \xrightarrow{k = \log_{3/2} n}, \quad T(1) = 0 \text{ and } T(2) = 1$$

That is $(1)S_k + (-3)S_{k-1} = 1$

The C.E is $(r-3)(r-1) = 0$, $r=3, 1$

$$S_k = C_1 3^k + C_2, \quad S_k = T\left((\frac{3}{2})^k\right), \quad \boxed{T(n) = C_1 n \cdot 2^k + C_2}$$

$$\begin{aligned} T(1) &= C_1 \cdot 2^0 + C_2 = 0 \\ T(2) &= C_1 \cdot 2^{1+k} + C_2 = 1 \end{aligned} \quad \Rightarrow \quad \begin{aligned} C_1(2^{1+k} - 1) &= 1, \quad C_1 = 1/(2^{\log_{3/2} 3} - 1) \\ C_2 &= -C_1 \end{aligned}$$

$$T(n) = C_1 \cdot n \cdot 2^k - C_1$$

$$= C_1 (n \cdot 2^{\log_{3/2} n} - 1)$$

$$= C_1 (n \cdot n^{\log_{3/2} 2} - 1)$$

$$= C_1 (n^{1+\log_{3/2} 2} - 1)$$

$$= C_1 (n^{\log_{3/2} 3} - 1)$$

$$\boxed{T(n) = \frac{n^{\log_{3/2} 3} - 1}{2^{\log_{3/2} 3} - 1}}$$

$$T(n) \in O(n^3) ?$$

$$\log_{3/2} 3 \doteq 2.71 < 3 \text{ 이므로}$$

$n \geq 1$ 인 모든 n 에 대해서 $n^{\log_{3/2} 3} - 1 \leq n^3$ 이다.

양변에 C_1 을 곱하면 $\left(\frac{n^{\log_{3/2} 3} - 1}{2^{\log_{3/2} 3} - 1}\right) \leq C_1 n^3$ 이다.
 $= T(n)$

\therefore 따라서, $n \geq N$ 인 모든 n 에 대하여 $T(n) \leq cn^3$ 이 성립하는

$c = 1/(2^{\log_{3/2} 3} - 1)$, $N = 1$ 이 존재하기 때문에 시간복잡도

$T(n)$ 은 $O(n^3)$ 에 속한다.