알고리즘 과제 보고서 3

2022110151 이주연

문제 1. 회문

1. 비순환 방식의 회문 알고리즘 (Iterative ver.)
   A. 유사코드

1. sentences.txt 파일 열기
2. 한 줄씩 문장을 읽어들이기
3. 문자열의 글자 수 세기
4. 반복문을 활용해 왼쪽과 오른쪽에서부터 글자를 비교하며 회문인지 확인하기
    4-0. 공백인 경우는 건너뛰도록 설정하고
    4-1. 회문이 아닌 경우 0을 반환하고
    4-2. 회문인 경우에는 1을 반환하기
5. 이를 바탕으로 회문인지/아닌지 구별되게 출력하기
6. sentences.txt 파일 닫기

   B. 프로그램 구현 (palindromeIter.c, sentences.txt zip에 함께 첨부)
      i. 실행 결과

```
Check if the following words are palindromes(not recursive ver.)
'madam' is A palindrome
'racecar' is A palindrome
'step on no pets' is A palindrome
'ABBA' is A palindrome
'apple' is NOT a palindrome
'hello' is NOT a palindrome
'eye' is A palindrome
'python' is NOT a palindrome
```

      ii. 설명

먼저, 파일을 열어 한 줄씩 단어를 읽어오도록 설정했습니다. 회문인지 아닌지 판별하는 함수(checkPalindrome(str))를 호출해 참(1)인 경우에는 "회문이 맞다"는 결과를, 거짓(0)인 경우에는 "회문이 아니다"는 결과를 출력하도록 설정했습니다.

이때 회문 판별 함수에서는 왼쪽 left 인덱스와 오른쪽 right인덱스를 설정해, 공백은 건너뛰고, while문을 활용해 left는 왼쪽에서 오른쪽으로, right는 오른쪽에서 왼쪽으로 1씩 이동하며(left++, right--) left 인덱스에 위치한 글자와 right 인덱스에 위치한 글자를 비교하도록 설정했습니다. 따라서 같지 않은 경우에는 회문이 아니므로 0이 반환되고, left와 right를 이동시키며 글자를 비교하다 left와 right가 만나는 경우에는 회문이 맞으므로 1이 반환됩니다. (e.g. madam : 0번과 4번 비교 둘다 m이므로 맞음 –(인덱스 이동)-> 1번과 3번 비교 둘다 a로 맞음 -> 결국 회문이 맞으므로 1 반환)

알고리즘 과제 보고서 3

2022110151 이주연

2. 순환 방식의 회문 알고리즘 (Recursive ver.)

A. 유사코드

1. sentences.txt 파일 열기

2. 한 줄씩 문장을 읽기

3. 문자열의 글자수 세기

4. 재귀함수 정의

    4-1. left(시작) 인덱스와 right(끝) 인덱스를 받고

    4-2. 공백인 경우는 건너뛰고

    4-2. 만약 문자가 다르면 회문x라고 판단하고 0 리턴

    4-3. 만약 문자가 같으면

        - left 는 +1, (인덱스 이동)
        - right는 -1 해서 또 다시 함수 호출(즉, 재귀호출)

    4-3. 인덱스를 움직이다, 시작과 끝이 만나면 0리턴 후 종료

5. 이를 바탕으로 회문인지/아닌지 구별되게 출력하기

6. sentences.txt 파일 닫기

B. 프로그램 구현 (palindromeRecur.c, sentences.txt)

i. 실행 결과

```
Check if the following words are palindromes(recursive ver.)
'madam' is A palindrome
'racecar' is A palindrome
'step on no pets' is A palindrome
'ABBA' is A palindrome
'apple' This word is NOT a palindrome
'hello' This word is NOT a palindrome
'eye' is A palindrome
'python' This word is NOT a palindrome
```

ii. 설명

먼저, 파일을 열어 한 줄씩 단어를 읽어오고, 회문 판별 함수를 호출해 참(1)일 때 회문이 "맞다", 거짓(0)일 때는 "아니다"고 출력되도록 설정한 것은 위와 동일합니다.

그러나 회문 판별 함수 내에서 재귀함수를 사용해 차이를 줬습니다. left 혹은 right인덱스의 문자가 공백인 경우 인덱스를 하나 이동시킬 수 있도록 함수를 재귀적으로 호출했고(이때 left는 left+1로, right는 right-1로 인덱스이동), 기본적으로 recurPalindrome(str, left+1, right-1)를 호출해 left는 오른쪽으로, right는 왼쪽으로 이동하며 글자를 확인하다 만약 left인덱스 글자와 right인덱스 글자가 다르면 0, 같은 경우 1이 반환되도록 설정했습니다. (madam: 0번과 4번 비교->재귀호출을 통해 1번과 3번 비교)

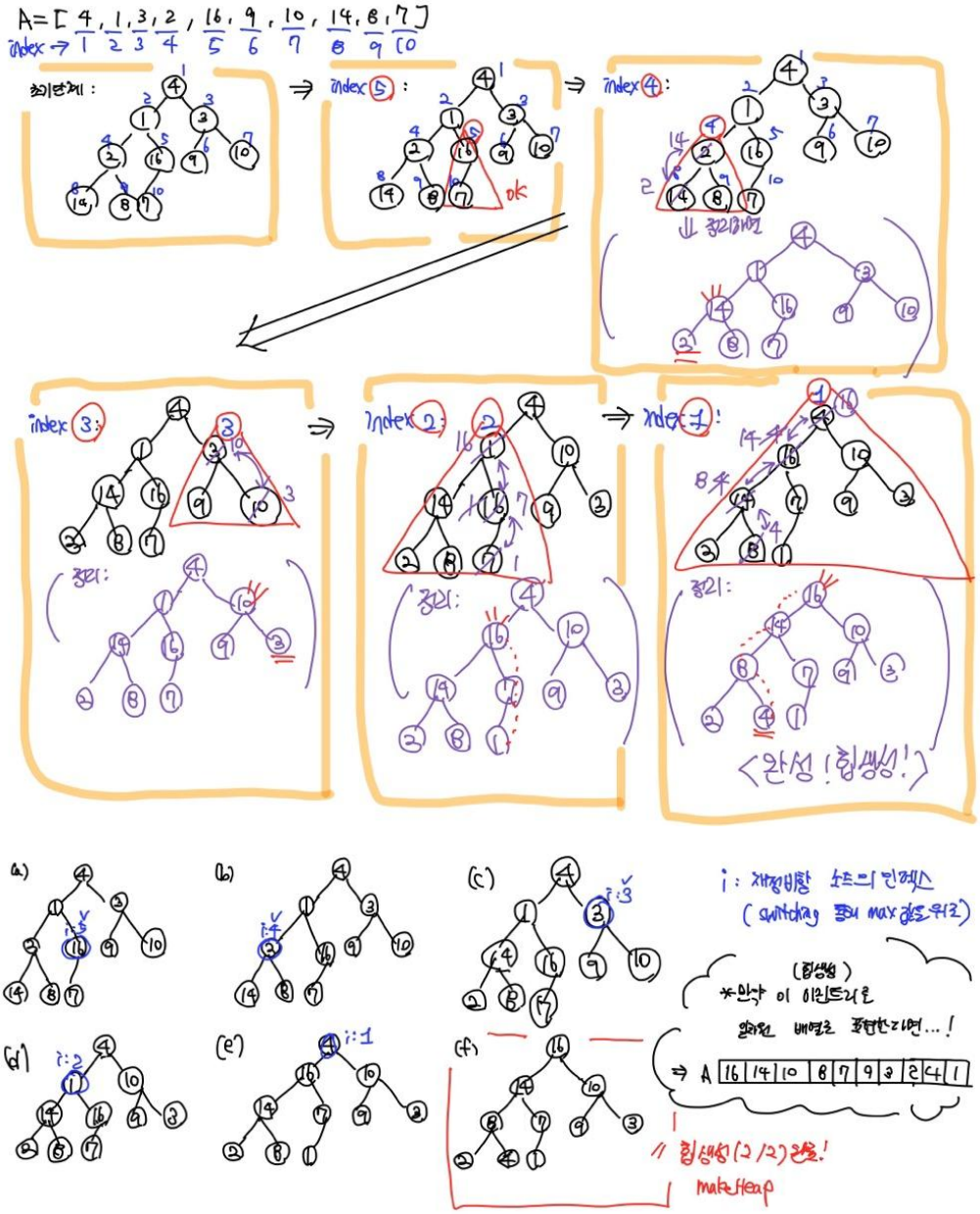
알고리즘 과제 보고서 3

2022110151 이주연

문제 2. 힙정렬 (50점) A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]

1. 힙 생성 (25점)

A. 단계별 이진트리 그리기

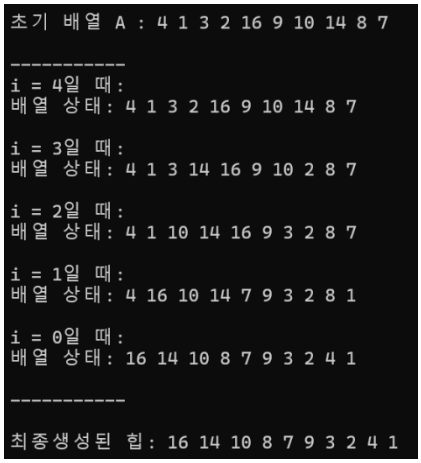


i (즉, 재정비할 노드의 인덱스)를 5->4->3->2->1순서로 옮겨가며 max값을 위로 보내 힙을 생성했습니다. 위 첫 번째 사진은 힙 생성을 위한 과정을 구체적으로 그린 모습이고, 두 번째 사진은 이를 바탕으로 정리한 그림입니다.

알고리즘 과제 보고서 3

2022110151 이주연

B. 코드 구현 (생성된 힙 일차원 배열로 출력)



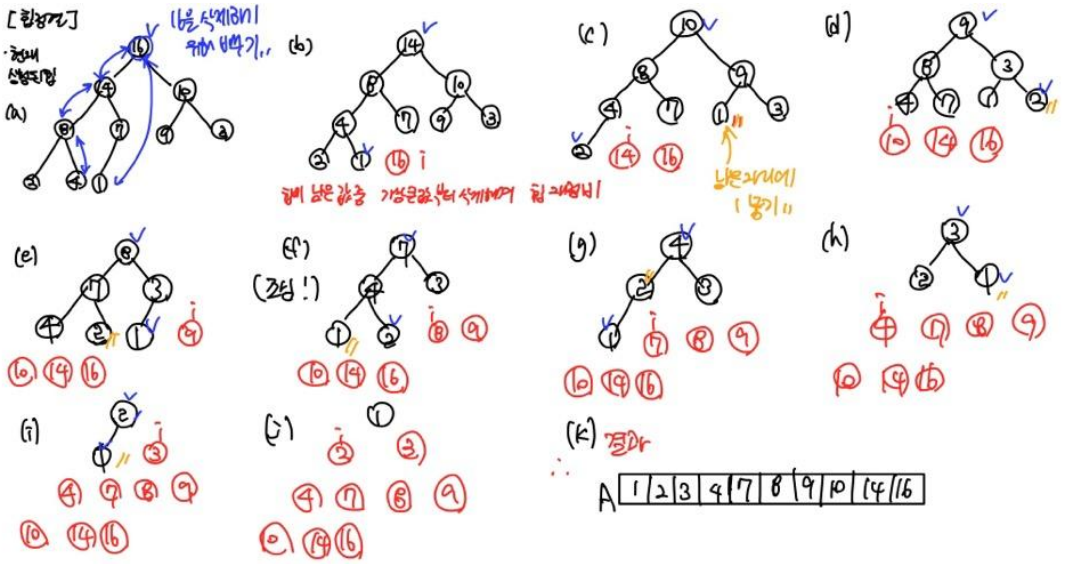
배열의 크기가 n이라고 할 때, n/2-1부터 시작해서 0까지 i를 감소하며, makeMaxHeap(배열a, i(i를 서브트리의 루트노드로 설정하기 위함), n-1(끝)) 반복 호출을 통해 서브트리의 맨 위(root)노드 값이 max값이 되도록 설정했습니다. i를 n/2-1부터 시작한 이유는 n/2-1은 가장 마지막 리프 노드의 부모 인덱스이기 때문에 여기서부터 시작해서 위로 올라가며 각각의 서브트리를 maxHeap으로 만들 수 있기 때문입니다. (참고로 makeMaxHeap(힙생성)함수에서 1)(if문1)은 어떤 값이 큰지 자식끼리 값을 비교한 것이고(더 큰 값이 son에 저장), 2)(if문2)는 어떤 값이 큰지 부모노드와 son값을 비교한 것입니다.) 자식이 부모보다 크면, son이 parent가 됩니다. (자식이 부모보다 크지 않을 때 반복은 종료(break)됩니다.)

알고리즘 과제 보고서 3

2022110151 이주연

2. 힙 정렬 (25점)

A. 단계별 이진트리 그리기



최대힙이 생성된 후, 최대 힙에서 max값을 배열 끝으로 보내 제거하고, 남은 원소로 다시 힙을 재정비하는 과정을 반복해 정렬된 결과를 출력했습니다.

B. 코드 구현 (정렬된 일차원 배열 출력)

```
초기 배열: 4 1 3 2 16 9 10 14 8 7
생성된 힙: 16 14 10 8 7 9 3 2 4 1
정렬된 배열: 1 2 3 4 7 8 9 10 14 16
```

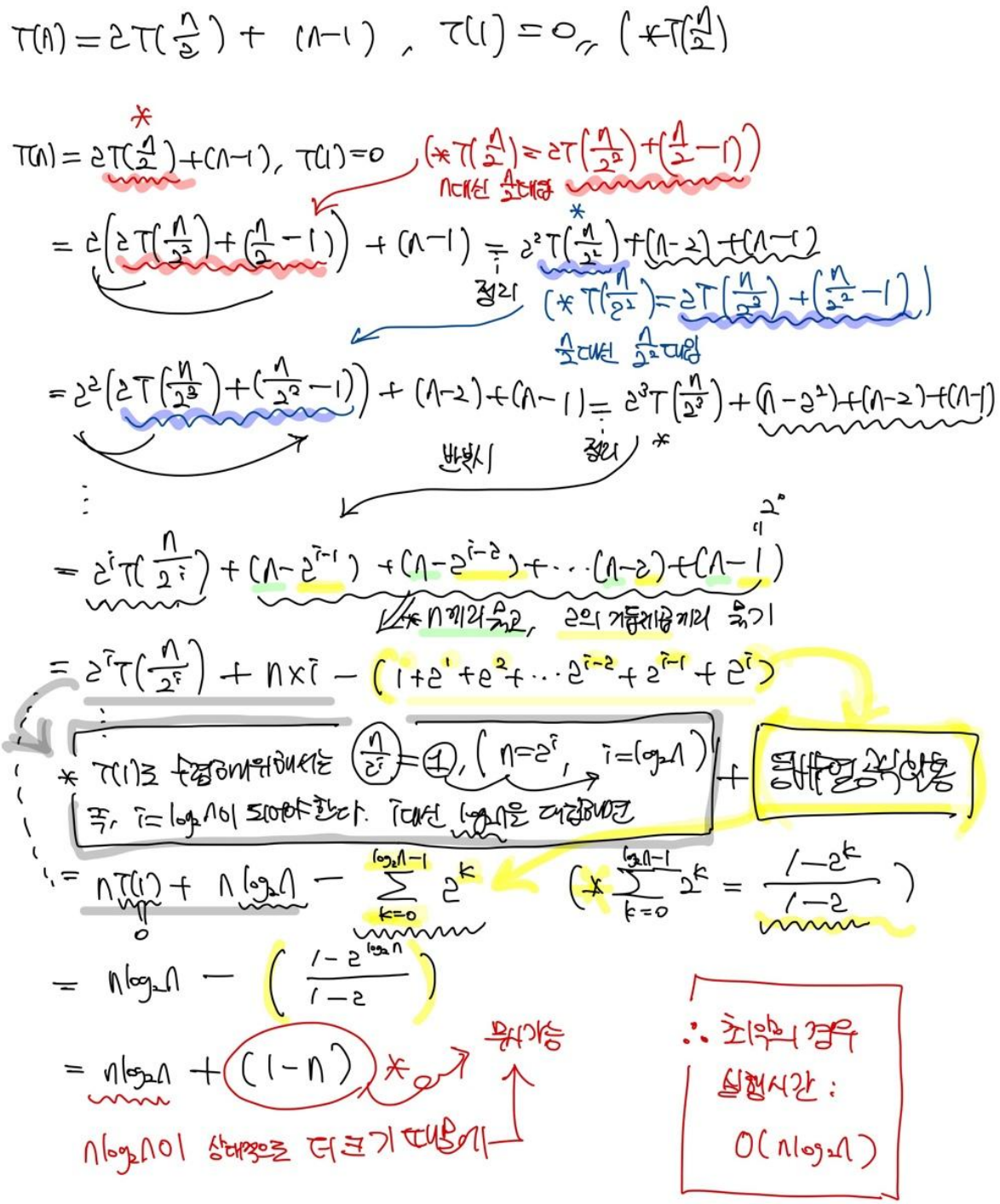
heapSort(a배열, 배열길이)를 호출하면, n/2 – 1(즉, 리프를 제외한 마지막 부모 노드)에서부터 root인 0까지 역순으로 돌며 max heap을 만든 후, 힙에서 최댓값을 꺼내 맨뒤로 보내주면서 하나씩 정렬했습니다. (맨뒤부터 시작해 앞으로 가며 정렬되도록 반복문을 사용했습니다.) 최댓값인 a[0]을 맨 뒤 값인 a[i]와 자리를 바꾼 후, 크기를 하나 줄인 상태에서 다시 힙생성 함수(makeMaxHeap)을 호출해 최대힙을 만들고 이를 계속 반복하며 오름차순으로 정렬될 수 있도록 설정했습니다. (가장 큰 값이 맨 뒤로 가며 뒤에서부터 정렬되었습니다.)

알고리즘 과제 보고서 3

2022110151 이주연

문제 3. 합병 정렬

1. 최악의 실행시간 구하기



T(n/2)를 없애기 위해 대입하며 T(1)으로 수렴할 수 있도록, 더욱이 등비수열을 활용해 nlogn이라는 최악의 경우 실행시간 결과를 도출했습니다.

알고리즘 과제 보고서 3

2022110151 이주연

문제 4. 합병 정렬

1. 순환적 합병 정렬

A. 유사코드

```
MergeSort(A, Left, Right)
    만약 left right 둘이 다르면
        Mid ← (Left + Right) / 2 (중간을 설정해서 중간을 기준으로)
        MergeSort(A, Left, Mid)  왼쪽 절반을 다시 재귀 호출하고
        MergeSort(A, Mid + 1, Right) 오른쪽 절반을 재귀 호출하고
        Merge(A, Left, Mid, Right) 정렬된 두 부분을 합치기
Merge(A, Left, Mid, Right)
        임시 배열 B (buffer) 생성하고
        left pointer와 right 포인터를 설정해서
      // 포인터를 옮겨 가며 왼쪽과 오른쪽 부분 배열을 비교하여 병합
      while LeftPtr는 Mid까지 RightPtr는 Right까지 움직이는 동안 do
        if A[LeftPtr] < A[RightPtr] then
            B에 A[LeftPtr] 저장
            LeftPtr ← LeftPtr + 1
        else
            B에 A[RightPtr] 저장
            RightPtr ← RightPtr +
    while LeftPtr ≤ Mid do // 남은 왼쪽 값들-> buffer에 저장
        B에 A[LeftPtr] 추가
        LeftPtr ← LeftPtr + 1
    while RightPtr ≤ Right do // 남은 오른쪽 값들 -> buffer에 저장
        B에 A[RightPtr] 추가
        RightPtr ← RightPtr + 1
    // B 배열을 A 배열로 복사
    for i from 0 to B 크기 - 1 do
        A[Left + i] ← B[i] //// B 배열을 A 배열로 복사
// main()  A ← {30, 20, 40, 35, 5, 50, 45, 10, 25, 15},  //초기 배열 출력  //병합 정렬 실행 //최종 정렬 결과 출력
```

알고리즘 과제 보고서 3

2022110151 이주연

B. 코드 zip 파일에 함께 첨부 (mergeSortRecur.c)

C. 결과 출력 (입력 A=[30 20 40 35 5 50 45 10 25 15])

```
Merge Sort (재귀 ver.)
초기 배열: 30 20 40 35 5 50 45 10 25 15

Merge(0,0,1)
B[0:1]: 20 30
A 배열 상태: 20 30 40 35 5 50 45 10 25 15

Merge(0,1,2)
B[0:2]: 20 30 40
A 배열 상태: 20 30 40 35 5 50 45 10 25 15

Merge(3,3,4)
B[3:4]: 5 35
A 배열 상태: 20 30 40 5 35 50 45 10 25 15

Merge(0,2,4)
B[0:4]: 5 20 30 35 40
A 배열 상태: 5 20 30 35 40 50 45 10 25 15

Merge(5,5,6)
B[5:6]: 45 50
A 배열 상태: 5 20 30 35 40 45 50 10 25 15

Merge(5,6,7)
B[5:7]: 10 45 50
A 배열 상태: 5 20 30 35 40 10 45 50 25 15

Merge(8,8,9)
B[8:9]: 15 25
A 배열 상태: 5 20 30 35 40 10 45 50 15 25

Merge(5,7,9)
B[5:9]: 10 15 25 45 50
A 배열 상태: 5 20 30 35 40 10 15 25 45 50

Merge(0,4,9)
B[0:9]: 5 10 15 20 25 30 35 40 45 50
A 배열 상태: 5 10 15 20 25 30 35 40 45 50

최종 정렬 결과: 5 10 15 20 25 30 35 40 45 50
```

main함수에서 mergeSort를 실행하면, mid를 기준으로 재귀함수 호출을 통해 왼쪽 배열을 정렬하고, 오른쪽 배열을 정렬해 merge를 통해 합병되도록 설정되어 있습니다. 이때, 정렬은 merge에서 실행되게 되는데 mid를 기준으로 left~mid까지 움직이는 leftpointer와 mid +1부터 right까지 움직이는 rightpointer를 설정해 포인터를 움직이며 값을 비교해 더 작은 값을 버퍼에 넣어주고 (포인터 1증가), 남은 값들은 버퍼에 저장해 병합한 정보를 다시 A에 복사해 A배열 상태를 볼 수 있게 설정했습니다. 이때, 단계별로 합병되는 모습을 보이기 위해, 최종적으로 A배열에 다시 복사하기 전에 병합한 정보와 B버퍼 배열을 출력해 합병되는 과정을 보이고자 했습니다. 그리고 bufPtr은 버퍼에 값을 넣어주기 위해 생성한 인덱스를 알려주는 포인터입니다.

알고리즘 과제 보고서 3

2022110151 이주연

2. 비순환적 합병 정렬

A. 유사 코드

```
입력: 배열 A[0...n-1], 정렬할 원소 수 n

1. width ← 1  // 병합 단위

2. while width < n do

    3. for i ← 0 to n-1 in steps of 2×width do

        4. left ← i

        5. mid ← i + width - 1

        6. right ← i + 2×width - 1

        7. 만약 mid ≥ n이면 병합 불가 → 건너뜀

        8. 만약 right ≥ n이면 right ← n-1로 조정

        9. Merge(A, left, mid, right) 수행

    10. width ← 2 × width
```

B. 코드 zip 파일에 함께 첨부 (mergeSortIter.c)

C. 결과 출력 (입력 A=[30 20 40 35 5 50 45 10 25 15])

```
Merge Sort (재귀X ver.)
초기 배열: 30 20 40 35 5 50 45 10 25 15

Merge(0,0,1)
B 전체: 20 30 _ _ _ _ _ _ _ _
A 배열 상태: 20 30 40 35 5 50 45 10 25 15

Merge(2,2,3)
B 전체: _ _ 35 40 _ _ _ _ _ _
A 배열 상태: 20 30 35 40 5 50 45 10 25 15

Merge(4,4,5)
B 전체: _ _ _ _ 5 50 _ _ _ _
A 배열 상태: 20 30 35 40 5 50 45 10 25 15

Merge(6,6,7)
B 전체: _ _ _ _ _ _ 10 45 _ _
A 배열 상태: 20 30 35 40 5 50 10 45 25 15

Merge(8,8,9)
B 전체: _ _ _ _ _ _ _ _ 15 25
A 배열 상태: 20 30 35 40 5 50 10 45 15 25

Merge(0,1,3)
B 전체: 20 30 35 40 _ _ _ _ _ _
A 배열 상태: 20 30 35 40 5 50 10 45 15 25

Merge(4,5,7)
B 전체: _ _ _ _ 5 10 45 50 _ _
A 배열 상태: 20 30 35 40 5 10 45 50 15 25

Merge(8,9,9)
B 전체: _ _ _ _ _ _ _ _ 15 25
A 배열 상태: 20 30 35 40 5 10 45 50 15 25

Merge(0,3,7)
B 전체: 5 10 20 30 35 40 45 50 _ _
A 배열 상태: 5 10 20 30 35 40 45 50 15 25

Merge(0,7,9)
B 전체: 5 10 15 20 25 30 35 40 45 50
A 배열 상태: 5 10 15 20 25 30 35 40 45 50

최종 정렬 결과: 5 10 15 20 25 30 35 40 45 50
```

알고리즘 과제 보고서 3

2022110151 이주연

메인함수에서 MergeSortNonRecursive을 호출하면, 비재귀 버전의 합병정렬이 시행됩니다. 이때, 재귀버전과 다르게 width(즉, 현재 합병할 블록의 크기)를 설정했습니다. a배열 전체를 2*width 간격으로 나눠 순서대로 합쳐 결론적으로 작은 블록에서 더큰 블록으로 병합되도록 설정했습니다.

특히, 이부분은 현재 어떤 두 블록을 병합할지, 범위를 지정한 부분입니다.

```
for (i = 0; i < n; i += 2 * width) {
        int Left = i; // 왼쪽 블록 시작 인덱스
        int Mid = i + width - 1; // // 왼쪽 블록의 끝 인덱스 (크기 width)
        int Right = i + 2 * width - 1; // 오른쪽 블록의 끝 인덱스 (크기 width)
```

정렬된 블록과 정렬된 블록을 합쳐 더 큰 정렬된 블록으로 만들기 위해, 병합 대상 구간의 왼쪽 시작 지점을 left로 설정하고, 왼쪽 블록의 끝 인덱스를 mid로 설정하고, 오른쪽 블록의 끝 인덱스(left + 2*width -1)를 right으로 설정해 이를 바탕으로 병합 함수(Merge)를 호출할 수 있도록 설정했습니다.