1.1 순차검색 과 이진검색 (반복적, 재귀적) 코드

< 전체 코드 >

```
import java.util.*;

public class SearchMain {
   private static final double MILLISEC = 1000000.00;
   private static final double MICROSEC = 1000.00;

   static int[] s = null;
   static int num, key;

   public static void main(String[] args) {
      num = 1000;

      Random r = new Random();
      s = new int[num];  // num크기의 int형 배열 s

      for (int i = 0; i < num; i++) {   //num만큼 반복
         s[i] = r.nextInt(num);    //s배열에 랜덤지정
      }
      key = r.nextInt(num);  //key 값도 랜덤지정

      for (int i = 0; i < num; i++) {
         System.out.print(s[i] + " ");  //정렬된 배열 s의 내용 출력
      }
      System.out.println("\nKey Value: " + key + "\n");

      long startTime = System.nanoTime( );
      int location = sequentialSearch();
      long endTime = System.nanoTime( );
      System.out.println("[Sequential Search Result]");
      System.out.println("Key value " + key + ": location " + location);
      System.out.format("Elapsed Time:  %8d nano sec.\n", endTime - startTime);
      System.out.format("Elapsed Time:  %8.3f micro sec.\n", (double)nanoSecTo-
MicroSec(endTime - startTime));
      System.out.format("Elapsed Time:  %8.3f milli sec.\n", (double)nanoSecTo-
MillSec(endTime - startTime));
      System.out.println();

      Arrays.sort(s); //배열정렬
      for (int i = 0; i < num; i++) {
         System.out.print(s[i] + " "); // 정렬된 배열 s의 내용 출력
      }
      System.out.println("\nKey Value: " + key + "\n");

      startTime = System.nanoTime();
      location = binarySearch(key,s); //위에 배열s안의 원하는 값 key
      endTime = System.nanoTime();
```

```
      System.out.println("[Binary Search Result]");
      System.out.println("Key value " + key + ": location " + location);
      System.out.format("Elapsed Time:  %8d nano sec.\n", endTime - startTime);
      System.out.format("Elapsed Time:  %8.3f micro sec.\n", (double)nanoSecTo-
MicroSec(endTime - startTime));
      System.out.format("Elapsed Time:  %8.3f milli sec.\n", (double)nanoSecTo-
MillSec(endTime - startTime));
      System.out.println();

      startTime = System.nanoTime();
      location = recursiveBinarySearch(0, num-1); //high값을 num-1의 크기로
해야하기때문  0부터시작
      endTime = System.nanoTime();
      System.out.println("[Recursive Binary Search Result]");
      System.out.println("Key value " + key + ": location " + location);
      System.out.format("Elapsed Time:  %8d nano sec.\n", endTime - startTime);
      System.out.format("Elapsed Time:  %8.3f micro sec.\n", (double)nanoSecTo-
MicroSec(endTime - startTime));
      System.out.format("Elapsed Time:  %8.3f milli sec.\n", (double)nanoSecTo-
MillSec(endTime - startTime));
   }

   public static int sequentialSearch() { //순차검색
      int location = 0;  //값을 찾았을때 반환하는 변수
      while (location < num && s[location] != key) //찾는값이 아니고, num보다
작을경우
         location++;
      if (location > num)
         location = -1;
      return location;  //값을 반환
      }

   public static int binarySearch(int key, int num[]) { //이진검색

      int low = 0;
      int high = num.length -1; //마지막 값은 배열 num크기보다 1작게
      int  mid;

      while(low <= high) { // 경우를 만족할떄 반복문
       mid = (low+high)/2;   //중간값 설정  // 정수 나눗셈 (나머지 버림)
         if( num[mid] > key )  //key값이 더 작을경우
            high = mid-1;  //최대값 즉 배열의 마지막 값을 mid-1로 범위를 점차 줄임
         else if ( num[mid] < key) // key값이 더 클경우
            low = mid+1;    //최소값을 mid+1로 범위를 점차 줄임
         else if(num[mid]== key)  //찾고자하는 값일 경우
            return mid;       //찾음
      }
      return -1;      //못찾음
   }
```

```
  public static int recursiveBinarySearch(int low, int high) {
    int mid;
    if (low > high)
      return -1;      // 못찾음
    else {
      mid = (low + high)/2;        // 정수 나눗셈 (나머지 버림)
      if (key == s[mid])
        return mid;        //  찾음
      else if (key < s[mid])  //key값이 더 작은경우
        return recursiveBinarySearch(low, mid-1);// 왼쪽 반 선택
왼쪽배열에서 탐색하기
      else
        return recursiveBinarySearch(mid+1, high);  // 오른쪽 반 선택
오른쪽배열에서 탐색하기

    }
  }

  public static double nanoSecToMicroSec(long nanoSec) {
    return nanoSec / MICROSEC;
  }

  public static double nanoSecToMillSec(long nanoSec) {
    return nanoSec / MILLISEC;
  }
  }
```

Code0. 순차검색 과 이진검색 (반복적, 재귀적) 전체코드

< 순차검색함수>

```
public static int sequentialSearch() { //순차검색
    int location = 0;  //값을 찾았을때 반환하는 변수
    while (location < num && s[location] != key) //찾는값이 아니고, num보다 작을경우
        location++;
    if (location > num)
        location = -1;
    return location;  //값을 반환
    }
```

Code1. 순차검색함수 코드

값을 찾았을 때 반환하는 변수인 location을 0으로 지정합니다. .

이후 num보다 작고 배열 중에 location값이 찾고자 하는 값 key와 다를 경우 location을 증가시키고 num보다 클 경우에는 배열 안에 존재하는 것이 아니므로 -1값을 반환합니다. 이 외의 경우는 값을 찾았을 때 이므로 location을 반환합니다.

< 이진검색 반복함수>

```
public static int binarySearch(int key, int num[]) { //이진검색

    int low = 0;
    int high = num.length -1; //마지막 값은 배열 num크기보다 1작게
    int  mid;

    while(low <= high) { // 경우를 만족할때 반복문
     mid = (low+high)/2;   //중간값 설정 // 정수 나눗셈 (나머지 버림)
        if( num[mid] > key )  //key값이 더 작을경우
            high = mid-1;   //최대값 즉 배열의 마지막 값을 mid-1로 범위를 점차 줄임
        else if ( num[mid] < key) // key값이 더 클경우
            low = mid+1;        //최소값을 mid+1로 범위를 점차 줄임
        else if(num[mid]== key)  //찾고자하는 값일 경우
            return mid;         //찾음
```

Code2. 이진검색 반복함수 코드

low=0, high을 num크기보다 1작게 값을 줍니다. 0부터 시작하므로 num크기를 다 주면 안된다는 것에 유의해야합니다.

우선 low <= high 라는 경우를 만족할 때 반복문을 돌게 합니다.

mid는 low, high 값을 더한 후 2로 나눈 정수 값을 나타냅니다. 나머지 값은 정수의 나눗셈에 의해 버려지게 됩니다. 주석으로 달은 각각의 경우에 따라 점차 범위를 좁혀가며 찾고자 하는 값을 반환합니다.

< 이진검색 재귀함수 >

```
public static int recursiveBinarySearch(int low, int high) {

    int mid;

    if (low > high)
        return -1;          // 못찾음
    else {
        mid = (low + high)/2;       // 정수 나눗셈 (나머지 버림)
        if (key == s[mid])
            return mid;         //  찾음
        else if (key < s[mid])  //key값이 더 작은경우
            return recursiveBinarySearch(low, mid-1);   // 왼쪽 반 선택  왼쪽배열에서 탐색하기
        else
            return recursiveBinarySearch(mid+1, high);  // 오른쪽 반 선택 오른쪽배열에서 탐색하기


    }
}
```

Code2. 이진검색 재귀함수 코드

low > high인 경우 조건에 맞지 않기 때문에 값을 찾지못해 -1을 반환합니다.

그 외의 경우에 mid는 low, high 값을 더한 후 2로 나눈 정수 값을 나타냅니다. 나머지 값은 정수의 나눗셈에 의해 버려지게 됩니다.

찾고자 하는 값 key가 s[mid]보다 작은 경우, 왼쪽 반을 선택하여 왼쪽 배열에서 재귀함수로 검색하고, 큰 경우 오른쪽 반을 선택하여 오른쪽 배열에서 재귀함수로 검색하여 값을 찾도록 합니다.

< num = 10,1000,10000 에 따른 수행시간결과 >

```
9 0 6 1 1 2 9 9 7 5
Key Value: 5

[Sequential Search Result]
Key value 5: location 9
Elapsed Time:     20100 nano sec.
Elapsed Time:    20.100 micro sec.
Elapsed Time:     0.020 milli sec.

0 1 1 2 5 6 7 9 9 9
Key Value: 5

[Binary Search Result]
Key value 5: location 4
Elapsed Time:      4584 nano sec.
Elapsed Time:     4.584 micro sec.
Elapsed Time:     0.005 milli sec.

[Recursive Binary Search Result]
Key value 5: location 4
Elapsed Time:      5289 nano sec.
Elapsed Time:     5.289 micro sec.
Elapsed Time:     0.005 milli sec.
```

Result1. num = 10

```
Key Value: 625

[Sequential Search Result]
Key value 625: location 915
Elapsed Time:     23978 nano sec.
Elapsed Time:    23.978 micro sec.
Elapsed Time:     0.024 milli sec.

Key Value: 625

[Binary Search Result]
Key value 625: location 619
Elapsed Time:      8815 nano sec.
Elapsed Time:     8.815 micro sec.
Elapsed Time:     0.009 milli sec.

[Recursive Binary Search Result]
Key value 625: location 619
Elapsed Time:      7758 nano sec.
Elapsed Time:     7.758 micro sec.
Elapsed Time:     0.008 milli sec.
```

Result2. num = 1000

```
Key Value: 4179

[Sequential Search Result]
Key value 4179: location 4872
Elapsed Time:    118832 nano sec.
Elapsed Time:   118.832 micro sec.
Elapsed Time:     0.119 milli sec.


Key Value: 4179

[Binary Search Result]
Key value 4179: location 4181
Elapsed Time:     23272 nano sec.
Elapsed Time:    23.272 micro sec.
Elapsed Time:     0.023 milli sec.

[Recursive Binary Search Result]
Key value 4179: location 4181
Elapsed Time:     17630 nano sec.
Elapsed Time:    17.630 micro sec.
Elapsed Time:     0.018 milli sec.
```

Result3. num = 10000

Fig. 1..num이 증가할때 그래프

순차 > 재귀 > 이진의 순으로 수행시간이 걸리는 것을 알 수 있습니다.

또한, num이 증가할수록 수행시간이 증가합니다.

반복적 이진탐색이 재귀적 이진탐색보다 수행시간이 훨씬 오래 걸린다는 것을 알 수 있습니다.

1.2 피보나치수열

```
import java.util.*;

public class FibonacciMain {
   private static final double MILLISEC = 1000000.00;
   private static final double MICROSEC = 1000.00;

   public static void main(String[] args) {
      int num = 10;

      long startTime = System.nanoTime();
      long result = iterativeFibonacci(num);
      long endTime = System.nanoTime();
      System.out.println("[Iterative Fibonacci]");
      System.out.println("Num " + num + " : Fibonacci Number " + result);
      System.out.format("Elapsed Time:  %8d nano sec.\n", endTime - startTime);
      System.out.format("Elapsed Time:  %8.3f micro sec.\n", (double)nanoSecTo-
MicroSec(endTime - startTime));
      System.out.format("Elapsed Time:  %8.3f milli sec.\n", (double)nanoSecTo-
MillSec(endTime - startTime));

      startTime = System.nanoTime();
      result = recursiveFibonacci(num);
      endTime = System.nanoTime();
      System.out.println("[Recursive Fibonacci]");
      System.out.println("Num " + num + " : Fibonacci Number " + result);
      System.out.format("Elapsed Time:  %8d nano sec.\n", endTime - startTime);
      System.out.format("Elapsed Time:  %8.3f micro sec.\n", (double)nanoSecTo-
MicroSec(endTime - startTime));
      System.out.format("Elapsed Time:  %8.3f milli sec.\n", (double)nanoSecTo-
MillSec(endTime - startTime));
   }

   public static long iterativeFibonacci(int num) {

      int before=0, curfib=1;
      int i, temp;

       if (num <= 1)
           return num;          return num;
       else{
           for (i = 1; i < num; i++){
               temp = curfib;    //현재값
```

```
            curfib = before + curfib;  //앞의 두항의 값을 합한게 현재값
            before = temp;   }
        return curfib;
        }
}


   public static long recursiveFibonacci(int num) {
   if ( num <= 1 )
      return num; //num 반환
   else
      return recursiveFibonacci( num - 1 ) +  recursiveFibonacci( num - 2 );
}// 재귀함수를 이용하여 앞의 두항의 합 구하기

public static double nanoSecToMicroSec(long nanoSec) {
   return nanoSec / MICROSEC;
}

public static double nanoSecToMillSec(long nanoSec) {
   return nanoSec / MILLISEC;
}
}
```

Code3. 피보나치수열 전체코드

```
  public static long iterativeFibonacci(int num) {

      int before=0, curfib=1;
      int i, temp;

      if (num <= 1)
          return num;
      else{
          for (i = 1; i < num; i++){
              temp = curfib;    //현재값
              curfib = before + curfib;   //앞의 두항의 값을 합한게 현재값
              before = temp;  }
          return curfib;
          }
```

Code4. 반복적 피보나치수열 코드

이전값을 0, 현재값을 1이라고 지정합니다. 먼저 빈 곳에 넣을 수 있도록 아무것도 들어있지 않은 변수 temp도 선언해 줍니다.

Num이 0일땐 0 반환, 1일땐 1반환 하고 1부터 num번의 반복을 해 줍니다.

빈곳에 현재값을 넣고, 현재값에 이전값과 현재값의 합한 값을 넣어줍니다. 이후에 이전값에 현재값을 넣어줍니다

```java
public static long recursiveFibonacci(int num) {
    if ( num <= 1 )
        return num; //num 반환
    else
        return recursiveFibonacci( num - 1 ) +  recursiveFibonacci( num - 2 );
}// 재귀함수를 이용하여 앞의 두항의 합 구하기
```

.

Code5. 재귀적 피보나치수열 코드

재귀함수를 통해 앞의 두항의 합을 구해 현재 값을 알아냅니다.

피보나치수열 재귀 함수는 하나의 함수에 두개의 함수를 호출하게 됩니다. 따라서 호출이 2배씩 늘어나게 되고, 총 N번의 수행을 하기 때문에 O(2^n-1) = O(2^n)의 시간복잡도를 나타냅니다.

피보나치수열 반복 함수는 1보다 작거나 같은 경우에는 바로 리턴이 되고 그 외의 경우는 num번의 수행을 하므로 O(n)의 시간복잡도를 나타냅니다.

```
[Iterative Fibonacci]
Num 3 : Fibonacci Number 2
Elapsed Time:      3526 nano sec.
Elapsed Time:     3.526 micro sec.
Elapsed Time:     0.004 milli sec.
[Recursive Fibonacci]
Num 3 : Fibonacci Number 2
Elapsed Time:      5642 nano sec.
Elapsed Time:     5.642 micro sec.
Elapsed Time:     0.006 milli sec.
```

Result1. num = 3

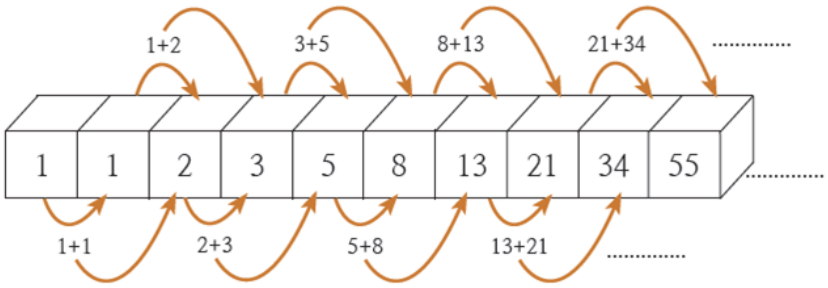
```
[Iterative Fibonacci]
Num 7 : Fibonacci Number 13
Elapsed Time:      2468 nano sec.
Elapsed Time:     2.468 micro sec.
Elapsed Time:     0.002 milli sec.
[Recursive Fibonacci]
Num 7 : Fibonacci Number 13
Elapsed Time:      5289 nano sec.
Elapsed Time:     5.289 micro sec.
Elapsed Time:     0.005 milli sec.
```

Result2. num = 5

10

```
[Iterative Fibonacci]
Num 10 : Fibonacci Number 55
Elapsed Time:      2115 nano sec.
Elapsed Time:     2.115 micro sec.
Elapsed Time:     0.002 milli sec.
[Recursive Fibonacci]
Num 10 : Fibonacci Number 55
Elapsed Time:      8462 nano sec.
Elapsed Time:     8.462 micro sec.
Elapsed Time:     0.008 milli sec.
```

Result3. num = 10



num이 3일경우 앞의 두 항의 값 1+1 = 2 의 결과값을 나타냅니다.

num이 5일경우 앞의 두 항의 값 2+3= 5 의 결과값을 나타냅니다.

num이 10일경우 앞의 두 항의 값 21+34 = 55 의 결과값을 나타냅니다.

또한 재귀적함수를 이용하여 값을 구할 때 시간이 더 오래 걸린다는 것을 알 수 있습니다.

Conclusion

최선의 경우만이 아닌 최악의 경우에도 자세히 생각을 해보는 계기가 되었습니다. 하나의 예시를 들자면 이진검색 알고리즘으로 키를 찾기 위해서 최악의 경우 S에 있는 항목을 몇 개나 검색을 해야하는 가에 대해 while문을 수행할 때마다 검색 대상의 크기가 절반으로 감소하기 때문에 $n = 2^k$라 하면 최악의 경우에 $\log n+1$번을 비교하는 최악의 경우가 나올 수 있습니다.

무조건적으로 하나의 방법만 사용하는 것이 수행시간을 줄일 수 있는 방법이 아니라, 어떠한 값을 구하냐의 경우에 따라 반복적, 재귀적을 적절히 사용하여 수행시간이 짧은 코드를 사용하는 것이 효율적이라는 것을 알게 되었습니다.

n개의 서로 다른 양의 정수 배열이 주어질 때, 그 배열을 크기가 각각 n/2가 되도록 2개의 배열로 분할하되, 두 부분 배열의 정수합의 차가 최대가 되도록 분할하라. (n은 2의 배수라고 가정)

< 의사코드 >

```
Arr : int[N];
Random()

For  1 to N;
put the values in arr[i];

sort the array;

for repeat as the length of the array;
show the arr[i]  as the length of the array;

repeat as  half of the length of the array;
Sum of small values (minSum);
```

Sum of largel values (maxSum);

To the integer sums of the two partial arrays is maximized

< 전체 코드 >

```
import java.util.Arrays;
import java.util.Random;

public class Main {
   static int N = 1000, minSum = 0, maxSum = 0;
   static int[] arr;

   public static void main(String[] args) {
      arr = new int[N];
      Random r = new Random();  //랜덤 값으로 넣어주기

      for(int i = 0; i < N; i++)  //배열에 랜덤값 넣기
         arr[i] = r.nextInt(N);

      Arrays.sort(arr); //배열 정렬  //nlogn번

      for(int i = 0; i < arr.length; i++) //배열 출력하기     //N번
         System.out.println(arr[i]);    // 값들을 보기위함
      for(int i = 0; i < arr.length/2; i++) {     // N/2번
         minSum += arr[i];       //베열 앞에서부터 넣기
         maxSum += arr[arr.length-1-i]; //배열 뒤에서부터  나머지 넣기
      }
      System.out.print(" minSum : ");
      System.out.println(minSum);
      System.out.print(" maxSum : ");
      System.out.println(maxSum);
   }
}
```

Code0. 배열분리코드

랜덤 함수를 이용해서 배열에 랜덤 값을 넣어줄 것 입니다.
우선 크기가 N인 arr배열을 선언합니다. 반복문을 통해 베열에 랜덤 값을 넣어주고 정렬해 줍니다. 정렬의 과정에서 nlogn번의 수행됩니다.
배열의 길이인 N만큼의 반복으로 각각의 배열에 들어간 값들을 출력합니다.
문제에서 배열의 크기가 N/2만큼의 두개의 배열로 분리하도록 하였으므로 N/2번 반복문을 통해 한 개의 배열에는 작은 값들을 넣고 다른 배열에는 나머지 값들을 넣어 줍니다.

2 결과

2.1 수행결과

```
964
967
968
970
970
971
973
976
976
976
978
978
979
979
981
982
982
984
986
988
988
988
990
990
991
992
994
995
997
998
998
998
998
999
 minSum : 126123
 maxSum : 380753
```

분리한 배열값의 개수가 너무 많아 캡쳐 한 화면에 다 넣을 수는 없었지만, 배열의 크기가 각각 N/2가 되도록 두개의 배열로 나누어 절반은 작은 값 배열 절반은 큰 값 배열에 minsum, maxsum에 넣어 두 배열의 두 부분 배열의 정수의 합의 차가 최대가 되도록 하였습니다.

2.2 시간복잡도를 구하기 위해 최악, 최선, 평균, 모든 경우 복잡도 분석중 어느 것을 활용해야 하는가? 선택한 복잡도 분석법에 대한 정확한 복잡도 수식을 구할 수 있다면 제시하시오.

시간복잡도를 구하기 위해서는 알고리즘의 핵심이 되는 연산의 횟수만 세야 합니다. 최악,최선,평균 세가지 경우가 있으며 평균적인 경우가 가장 이상적으로

느껴질 수 있지만, 알고리즘이 복잡할수록 평균의 경우를 구하기 어렵기 때문에 최악의 경우를 활용합니다.

정렬을 할 때 NlogN 번, 배열을 보여줄 때 N 번, 크기가 N/2인 배열로 만들기 위해 절반씩 큰값,작은값을 넣어줄 때 N/2번 입니다. 알고리즘의 NlogN + N + N/2 로 T(n) = NlogN 입니다.

➔ 알고리즘의 점근적 복잡도는 NlogN 입니다.

```
import java.util.*;

public class MergeSortMain {

   static int[] s1 = null;
   static int[] s2 = null;
   static int num;
   static int[] tempArray = null; / /inPlaceMergeSort를 위한 임시 배열
      public static void main(String[] args) {
        num = 10000;

        Random r = new Random(System.currentTimeMillis());
        s1 = new int[num]; //Initialize the array s1
        s2 = new int[num]; //Initialize the array s2
        tempArray = new int[num]; //Initialize the additional array

        for (int i = 0; i < num; i++) {
           s1[i] = s2[i] = r.nextInt(num);
        }



        long startTime = 0, endTime = 0;
        int location = -1;

        startTime = System.nanoTime();
        mergeSort(num, s1);

          System.out.println("s1");
               for (int i = 0; i < num; i++) {   //정렬된 배열
                  System.out.print(s1[i] + " ");
               }
               System.out.println();

               System.out.println("s2");
```

```
            for (int i = 0; i < num; i++) {  //정렬되지 않은 배열
               System.out.print(s2[i] + " ");
            }
            System.out.println();
            System.out.println();


        endTime = System.nanoTime();
        System.out.println("[MergeSort Result]");
        System.out.println("Elapsed Time: " + (endTime - startTime) + "nano
sec.");
        System.out.println();


        startTime = System.nanoTime();
        inPlaceMergeSort(0, num-1);
        endTime = System.nanoTime();
        System.out.println("[In-place MergeSort Result]");
        System.out.println("Elapsed Time: " + (endTime - startTime) + "nano
sec.");
        System.out.println();
      for (int i = 0; i < num; i++) {
         System.out.print(s1[i] + " ");
      }

      System.out.println();
}


   public static void mergeSort(int num, int[] s) { //mergeSort [Algorithm
2.2]
      //merge(h, m, u, v, s);
      if (num > 1) {

         int k = 0;
         int h = (num/2), m = num - h;
         int [] U = new int [h];
         int [] V = new int [m];

         //copy S[1] through S[h] to U[1] through U[h];
         //copy S[h+1] through S[n] to V[1] through V[m];

         for(int i = 0; i < h; i++) {
            U[i] = s[i];
         }
         for(int j = h; j < num; j++) {
            V[k] = s[j];
            k++;
         }
         mergeSort(h,U);
         mergeSort(m,V);
         merge(h,m,U,V,s);

      }
   }
```

```
   public static void merge(int h, int m, int[] u, int[] v, int[] s) { //merge
[Algorithm 2.3]
      int i=0, j=0, k=0;

      int t=0;

      while ( i < h && j < m ) { // i와 j가 배열 u,v의 길이를 넘지 않을때 반복
         if (u[i] < v[j]) {//각 포인터가 가리키는 데이터 두 개를 비교하여 더 작은
것을 S 배열에 복사
            s[k] = u[i];
            i++;  //복사한것에 대해 다음으로 옮김

         }
      else { //반대의 경우
         s[k] = v[j];
         j++;
         }
      k++;
      }

      //냠은 데이터들을 순서대로 복사
      if(i >= h) {  // S[h+m]  v의 인덱스가 4까지
         for (t = j; t<m; t++) {
            s[k] = v[t];
            k++;
         }
      }

      else {
         for( t = i; t<h; t++ ) {
            s[k] = u[t];    // U[i] ~ U[h]를 S[k] ~ S[h+m]
            k++;
         }
      }

   }
   public static void inPlaceMergeSort(int low, int high) { //mergesort2 [Al-
gorithm 2.4]
         //배열 분할함수

         if (low < high) {
            int mid = ((low + high) / 2);
            inPlaceMergeSort(low, mid);  //low부터 mid까지 분할
            inPlaceMergeSort(mid+1, high); //mid+1부터 high까지 분할
            inPlaceMerge(low, mid, high);//분할된 배열을 합병하여 정렬
         }
      }

      //두 개의 정렬된 부분배열을 하나의 정렬된 배열로 합병 함수
      public static void inPlaceMerge(int low, int mid, int high) { //merge2
[Algorithm 2.5]
```

```
        int i, j, k;
        i = low; j = mid + 1; k = low;
        //low~mid, mid+1~high 두 부분으로 분할된 배열을 병합하기 위해 선언
        while (i <= mid && j <= high) {
           if (s1[i] < s1[j]) {//분할후 s[i]가 더 작다면
              tempArray[k] = s1[i];  //임시배열에 저장해주고
              i++;             // 인덱스 증가
           } else {//분할한 부분 중 작은 원소가 s[j]일때
              tempArray[k] = s1[j];
              j++;
           }
           k++;
        }


        //나머지 데이터들 순서대로 복사
        if(i > mid) {
           for( int a =j; a<=high; a++) {
              tempArray[k] = s1[a];
              k++;
           }
        }
        else {
           for( int b = i; b<=mid; b++) {
              tempArray[k] = s1[b];
              k++;
           }
        }
        for( int c=low; c<=high; c++) {
            s1[c] = tempArray[c];

         }
     }


}
```

Code1. 머지소트와 머지함수를 이용한 코드

i와 j를 num/2만큼 즉 배열의 크기가 num/2만큼인 두개의 배열 u[i] 와 v[j]로 나눕니다. 반복문을 통해 u[i]의 각각을 가리키는 데이터 두개를 비교하여 더 작은 것을 S 배열에 복사 합니다. u[i]와 v[j] 에 또한 복사합니다.
나머지 남은 데이터를 순서대로 복사합니다. 만약 i가 num/2 보다 클때 v의 인덱스가 4까지를 v[t]에 넣고 5부터 나머지 즉 뒷부분에 있는 값들을 u[t]에 넣습니다.

inplaceMergesort 함수는 배열을 분할합니다. 먼저 mid로 가운데 값을 정하고 이를 기준으로 작은 것과 큰 것으로 두개의 배열로 분할 후 inplaceMerge로 분할된 배열을 합칩니다.

inplaceMerge 함수는 분할된 배열을 합치는 것으로 두 개로 나뉜 배열을 합치기 위해 기준점을 잡고 이보다 작을 경우 임시 배열에 넣어 저장 후 인덱스를 증가시킵니다. 나머지 데이터들도 순서대로 복사시켜 넣습니다.

2.2 결과

Num= 10

```
[MergeSort Result]
Elapsed Time: 512352nano sec.

[In-place MergeSort Result]
Elapsed Time: 7757nano sec.
```

Num= 100

```
[MergeSort Result]
Elapsed Time: 3374546nano sec.

[In-place MergeSort Result]
Elapsed Time: 58887nano sec.
```

Num= 1000

```
[MergeSort Result]
Elapsed Time: 15213666nano sec.

[In-place MergeSort Result]
Elapsed Time: 575824nano sec.
```

Num= 10000

```
[MergeSort Result]
Elapsed Time: 130226447nano sec.

[In-place MergeSort Result]
Elapsed Time: 2498293nano sec.
```

Num의 크기가 작을 때는 수행 시간의 차이가 눈에 띄게 나지 않았는데 크기가 점점 클수록 InplaceMergeSort의 수행시간이 짧은 결과가 나타났습니다.
배열을 새로 만드는 과정을 거치지 않고 원래의 배열만 사용하기 때문에 더 수행시간이 빠르게 나타난 것 같습니다.

2.3 실습4

```
import java.util.*;

public class QuickAndMergeSortMain {

   static int[] s1 = null; //정렬할 배열 선언
   static int[] s2 = null;
   static int[] tempArray = null; // inPlaceMergeSort를 위해 임시 정적 배열
객체를 선언

   static int num = 0;  //배열 크기
   static int TRIALS = 10;  //수행시간을 비교할 횟수 저장
   static long[] elapsedTimeOfMergeSort = null; //합병정렬 수행시간 저장
   static long[] elapsedTimeOfQuickSort = null; //퀵정렬 수행시간 저장
   static long totalElapsedTimeOfMergeSort = 0; //합병정렬 총 수행시간
   static long totalElapsedTimeOfQuickSort = 0; //퀵정렬 총 수행시간

   public static void main(String[] args) {
      num = 8;
      elapsedTimeOfMergeSort = new long[TRIALS];
      elapsedTimeOfQuickSort = new long[TRIALS];

      Random r = new Random(System.currentTimeMillis());
      long startTime = 0, endTime = 0;

      for (int trial = 0; trial < TRIALS; trial++) {
         System.out.println("*** Trial #" + trial + ": Starts with " + num + "
integers");
         s1 = new int[num];
         s2 = new int[num];
         tempArray = new int[num];
         for (int i = 0; i < num; i++) { //랜덤값으로 넣기
            s1[i] = s2[i] = r.nextInt(num);
```

```
        }

        System.out.println("Before Sorting");
        printArrays(s1, s2);

        startTime = System.nanoTime();
        inPlaceMergeSort(0, num-1); //NOTE: s1에 대해서 합병정렬
        endTime = System.nanoTime();
        elapsedTimeOfMergeSort[trial] = endTime - startTime;
        totalElapsedTimeOfMergeSort += elapsedTimeOfMergeSort[trial];

        startTime = System.nanoTime();
        quickSort(0, num-1);   //NOTE: s2에 대해서 퀵정렬
        endTime = System.nanoTime();
        elapsedTimeOfQuickSort[trial] = endTime - startTime;
        totalElapsedTimeOfQuickSort += elapsedTimeOfQuickSort[trial];

        System.out.println("After Sorting");
        printArrays(s1, s2); //정렬 후 배열 요소들 출력

        num = num * 2;  //배열길이 2배
     }

     System.out.println("[MergeSort vs. QuickSort]"); //비교
     num = 8;
     for (int i = 0; i < TRIALS; i++) {
        System.out.println(num + ": MergeSort - " + elapsedTimeOfMergeSort[i]
+ ", QuickSort - " + elapsedTimeOfQuickSort[i]);
        num = num * 2;
     }
     System.out.printf("Average Elapsed Time of Merge Sort: %.2f, Average
Elapsed Time of Quick Sort: %.2f", (double)totalElapsedTimeOfMergeSort/(dou-
ble)TRIALS, (double)totalElapsedTimeOfQuickSort/(double)TRIALS);
   }


     public static void inPlaceMergeSort(int low, int high) { //mergesort2
[Algorithm 2.4]
        //배열 분할함수

        if (low < high) {
           int mid = ((low + high) / 2);
           inPlaceMergeSort(low, mid);  //low부터 mid까지 분할
           inPlaceMergeSort(mid+1, high); //mid+1부터 high까지 분할
           inPlaceMerge(low, mid, high);//분할된 배열을 합병하여 정렬
        }
     }

     //두 개의 정렬된 부분배열을 하나의 정렬된 배열로 합병 함수
     public static void inPlaceMerge(int low, int mid, int high) { //merge2
[Algorithm 2.5]
        int i, j, k;
        i = low; j = mid + 1; k = low;
```

```
        //low~mid, mid+1~high 두 부분으로 분할된 배열을 병합하기 위해 선언
        while (i <= mid && j <= high) {
           if (s1[i] < s1[j]) {//분할후 s[i]가 더 작다면
              tempArray[k] = s1[i];  //임시배열에 저장해주고
              i++;             // 인덱스 증가
           } else {//분할한 부분 중 작은 원소가 s[j]일때
              tempArray[k] = s1[j];
              j++;
           }
           k++;
        }


        //나머지 데이터들 순서대로 복사
        if(i > mid) {
           for( int a =j; a<=high; a++) {
              tempArray[k] = s1[a];
              k++;
           }
        }
        else {
           for( int b = i; b<=mid; b++) {
              tempArray[k] = s1[b];
              k++;
           }
        }
        for( int c=low; c<=high; c++) {
            s1[c] = tempArray[c];

         }
     }


   public static void quickSort(int low, int high) { //quicksort [Algorithm
2.6]
      int pivotpoint;  //기준점 피벗위치 저장
      if (high > low) {
         pivotpoint = partition(low, high); // low와 high 사이애 피벗의 위치 결정
         quickSort(low, pivotpoint-1); //기준점을 중심으로 작은거 왼쪽
         quickSort(pivotpoint+1, high); //큰거 오른쪽
      }

   }

   public static int partition(int low, int high) { //partition [Algorithm 2.7]
            int i, j, pivotpoint;
            int pivotitem;
            int temp;
            pivotitem = s2[low]; // 피벗의 위치는 배열의 맨 앞에서부터 시작
            j = low; //j의 역할: for루프가 끝난 후 pivotitem이 있을 위치 기억

            for(i = low + 1; i <= high; i++) {// i를 high까지 반복시키며 피벗보다
```

```
작은 값들의 위치를 교환
              if (s2[i] < pivotitem) { //피벗보다 작으면
                 j++;   //j는 배열 아이템 중 pivotitem보다 작은 것의 개수 만큼
증가함
                 temp = s2[i];  // 배열의 i위치와 j위치의 원소를 교환
                 s2[i] = s2[j];
                 s2[j] = temp;
                 }
           }
           pivotpoint = j; // 반복문이 끝난 뒤 가장 마지막의 피벗위치는 j가 됨

           temp = s2[low];
           s2[low] = s2[pivotpoint];
           s2[pivotpoint] = temp;
        // low와 피벗 위치의 배열원소를 바꾸어 피벗이 제자리를 찾게 됨

           return pivotpoint;  //현재 피벗위치 반환
           }
  public static void printArrays(int[] s1, int[] s2) {
     System.out.println("Array s1:");
     for (int i = 0; i < num; i++) {
        System.out.print(s1[i] + " ");
     }
     System.out.println();
     System.out.println("Array s2:");
     for (int i = 0; i < num; i++) {
        System.out.print(s2[i] + " ");
     }
     System.out.println();
     System.out.println();
  }
  }
```

실습3과 다른게 실습4에서는 quicksort함수도 사용합니다. 이 함수에서는 먼저 피벗포인트로 피벗의 위치를 저장하여 조건문에 해당할 시 피벗포인트를 low에서 high사이의 위치로 지정해줍니다. 이후 low에서 지정한 위치-1까지의 배열로 나누고 정렬하고 지정한 위치+1 부터 high까지로 배열을 나누고 정렬합니다.

Partition 에서는 퀵정렬에서 배열을 피벗기준으로 정렬하고 현재 피벗위치를 반환해줍니다. 먼저 j의 역할은 for루프가 끝난 후 pivotitem이 있을 위치 기억해줍니다. 피벗의 위치를 배열의 맨앞에부터 시작하게 지정해줍니다. 그렇기 때문에 반복문을 돌릴 때 low 가아닌 low+1부터 돌려야합니다.

조건문을 통해 배열 i.와 j를 바꾸며 low와 피벗 위치의 배열 원소들끼리 교환하여 피벗이 제자리를 찾게 해줍니다.

2.4 결과

Num = 8 ~4096

```
[MergeSort vs. QuickSort]
8: MergeSort - 11284, QuickSort - 5642
16: MergeSort - 9520, QuickSort - 4584
32: MergeSort - 16573, QuickSort - 8463
64: MergeSort - 42667, QuickSort - 21863
128: MergeSort - 92738, QuickSort - 42314
256: MergeSort - 209102, QuickSort - 115658
512: MergeSort - 441829, QuickSort - 269400
1024: MergeSort - 177718, QuickSort - 79691
2048: MergeSort - 199582, QuickSort - 161499
4096: MergeSort - 477796, QuickSort - 339218
Average Elapsed Time of Merge Sort: 167880.90, Average Elapsed Time of Quick Sort: 104833.20
```

배열의 길이를 나타내는 num값을 8 , 16, 32, 64, …4096으로 두 배씩 증가시켜가며 수행 시간을 비교하였습니다. num값이 증가할수록 두 배열의 수행 시간 차이는 더 심해졌습니다.

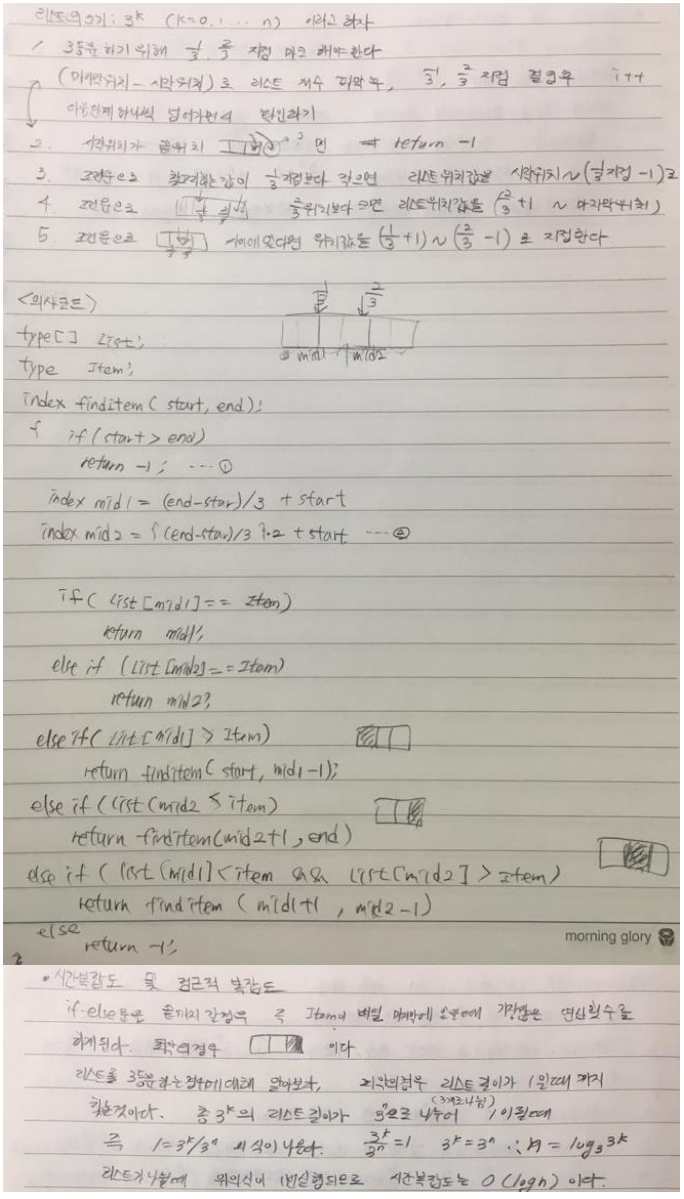
합병정렬은 모든 경우에 nlogn이고, 퀵 정렬은 최선의 경우에 n이고 최악의 경우에 n*n , 이 둘을 제외한 경우엔 nlogn, 입니다.

최악의 경우 n*n임에도 수행 시간은 퀵 정렬이 빠른 것을 결과화면으로 볼 수 있습니다. 최선의 경우와 최악의 경우가 극히 드물 것이며 같은 시간복잡도임에도 수행시간에는 퀵정렬의 평균수행시간이 더 짧은 것을 볼 수 있습니다.

3 6번

n개의 아이템을 가진 정렬된 리스트를 거의 n/3개 아이템을 가진 3개의 부분리스트로 분할하여 검색하는 알고리즘을 작성하라. 이 알고리즘은 찾는 아이템이 있을 만한 부분리스트에서 아이템을 검색하는데, 이 부분리스트를 다시 거의 같은 크기의 3개 부분리스트로 분할한다. 아이템을 찾거나, 아이템이

리스트에 없다고 결정할 때까지 이 과정을 되풀이한다. 이 알고리즘을 분석하고, 결과를 차수표기법으로 표시하라.

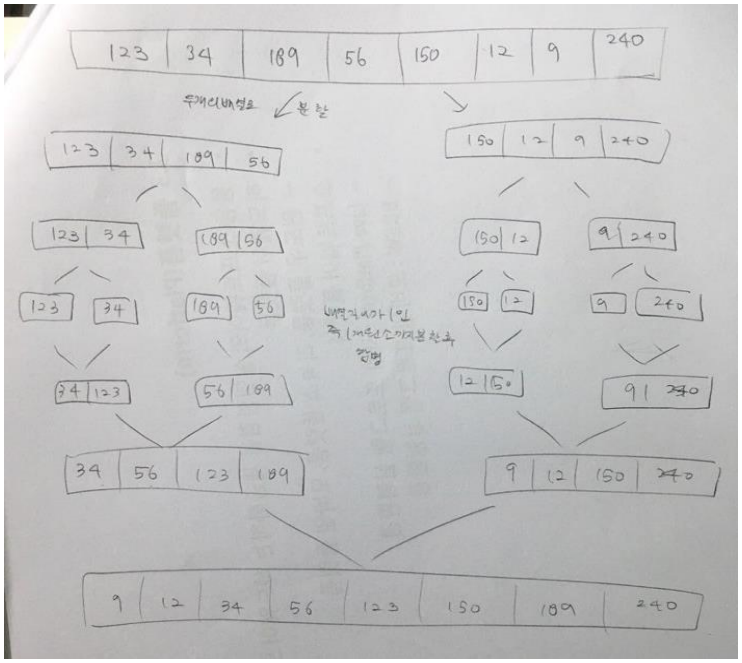


4 8번, 9번

8. 합병정렬 알고리즘 2.2와 2.4를 사용하여 다음 리스트를 정렬하라. 알고리즘이 진행되는 과정을 단계별로 보여라.

123 34 186 56 150 12 9 240

9. 연습문제 8에서 재귀 호출 트리를 구축하라.



5 15번

```
Output solve(input I) {
  if (size(I) == 1)
    해답 O를 바로 찾는다.;
  else {
    I를 3개의 입력 I1,I2,I3로 분할한다.;
    여기서 j=1,2,3에 대하여 size(Ij)=size(I)/3;
   for(j=1; j<=3; j++)
     Oj=solve(Ij);
    입력 I로 P를 푼 해답 O를 구하기 위하여 O1,O2,O3를 합병한다.;
   return O
   }
}
```

단위연산: 분할하고 합병하기

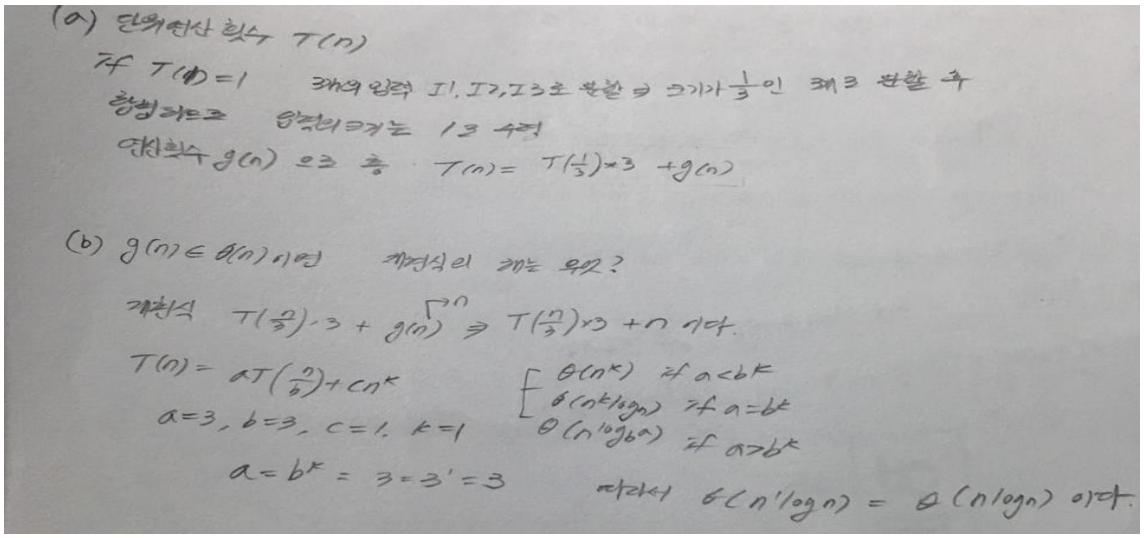
g(n)은 분할하고 합병하는 단위연산 시간복잡도 함수

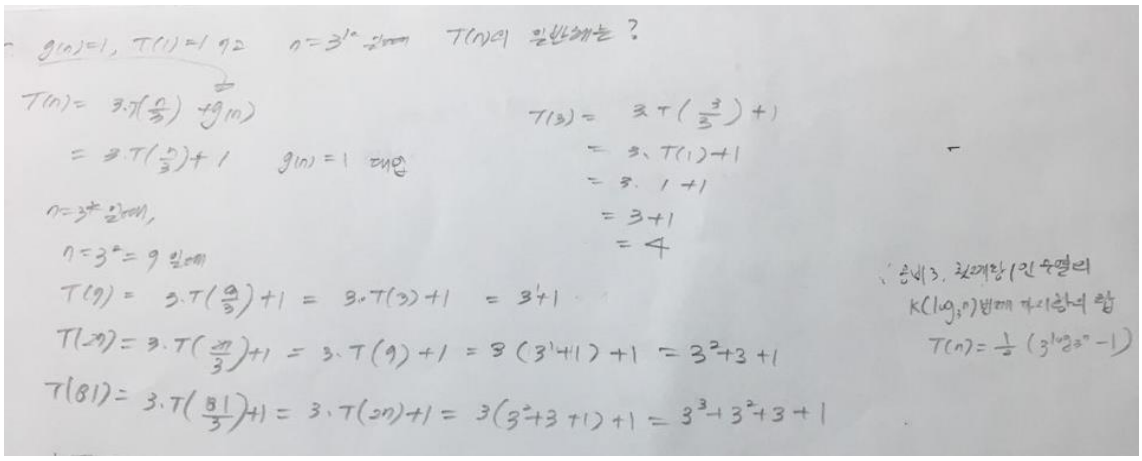
크기가 1인 사례는 단위연산 시간복잡도가 1이라고 가정

(문제 a) 입력크기가 n인 경우, 문제를 푸는 데 필요한 단위연산의 횟수를 표시하는 재현식 T(n)은?

(문제b) g(n) $\in\theta(n)$ 일 때, T(n)의 정확한 재현식은? $n=3^k$일 때 Master Theorem을 사용한 T(n)의 점근적 복잡도는?

(문제c) g(n)=1, T(1)=1 이고 $n=3^k$ 일 때의 T(n)의 일반해 (Closed Form)는?


(a) 단위연산 횟수 T(n)
if T(1)=1 3개의 입력 I1, I2, I3로 분할 ⇒ 크기가 1/3인 3개 부분 추
항상 리프로 입력의 크기는 1로 수렴
연산횟수 g(n) 으로 총 T(n)= T(1/3)*3 +g(n)
(b) g(n) ∈ Θ(n) 이면 재귀식의 해는 무엇?
재귀식 T(n/3)·3 + g(n) ⇒ T(n/3)×3 +n 이다.
T(n)= aT(n/b)+cn^k
a=3, b=3, c=1, k=1
Θ(n^k) if a<b^k
Θ(n^k log n) if a=b^k
Θ(n^log_b a) if a>b^k
a=b^k = 3=3^1=3
따라서 Θ(n^1 log n) = Θ(n log n) 이다.

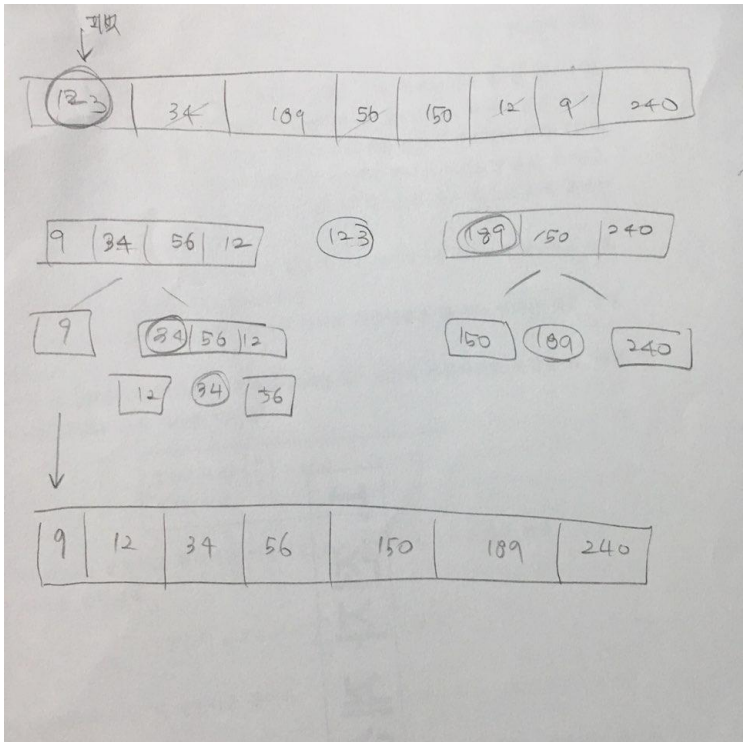

g(n)=1, T(1)=1 일 때 n=3^k 일때 T(n)의 일반해는 ?
T(n)= 3·T(n/3) +g(n)
= 3·T(n/3)+1 g(n)=1 대입
n=3^k 일때,
n=3^2=9 일때
T(9)= 3·T(9/3)+1 = 3·T(3)+1 = 3+1
T(27)= 3·T(27/3)+1 = 3·T(9)+1 = 3(3+1)+1 = 3^2+3+1
T(81)= 3·T(81/3)+1 = 3·T(27)+1 = 3(3^2+3+1)+1 = 3^3+3^2+3+1
T(3)= 3T(3/3)+1
= 3·T(1)+1
= 3·1+1
= 4
∴ 문제3. 초기항이 1인 수열의
K(log_3 n)번째 까지항의 합
T(n)= 1/2 (3^(log_3 n) -1)

6 19번 ,20번

19. 빠른정렬(알고리즘 2.6)을 사용하여 다음 리스트를 정렬하라. 알고리즘이 진행되는 과정을 단계별로 보여라.

123 34 189 56 150 12 9 240

20. 연습문제 19에서 재귀 호출 트리를 구축하라.



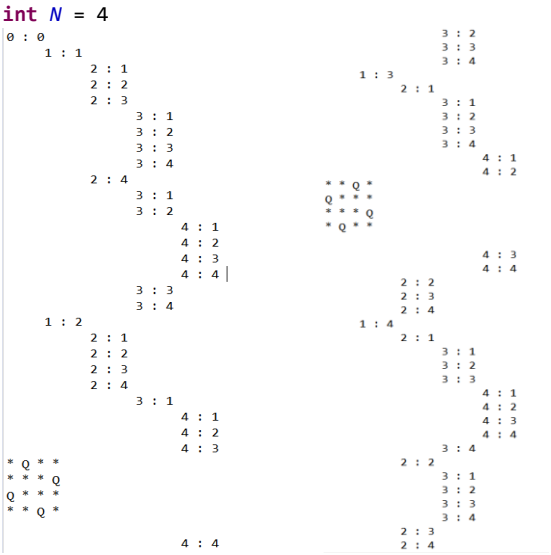
```
public class Nqueens {
    public static int[] col = null;// 퀸이 놓여있는 열
    public static int N = 4;// 총 열의 개수

    public static void main(String[] args) {
        col = new int[N + 1];// N개의 열을 생성
        queens(0);// 0을 매개변수로 전달하여 되추적을 시작
    }
    public static void queens(int i) {// i번째 열에 퀸을 놓는것을 되추적
        if (i == 0)  // 유망
            System.out.format("%d : %d %n", i, col[i]);
        else// i가 0이 아니면서 퀸을 놓을 수 있는 위치라면 i,col[i]를 출력
            System.out.format("%" + (i * 6) + "d : %d %n", i, col[i]);

        if (isPromising(i)) {// 현재의 행이 유망한 경우
            if (i == N) // 행을 의미하는 i가 N과 같다면 해답을 구한 것이다.
                printQueens();
            else {// 유망하지만 끝까지 도달하지 않은 경우
                for (int j = 1; j <= N; j++){// i+1번째 행의 j번째 열에 퀸
                  col[i + 1] = j;
                  queens(i + 1); // i+1번째 행에 대해 재귀호출하여 위 과정을
반복하여 유망성을 추적
                }
            }
        }
    }
    public static boolean isPromising(int n) {//n번째 열의 유망성을 검사
        int k = 1; //i번째 열의 k번째 행을 의미
        boolean isPromising = true; //유망하다고 가정하에 진행
        while (k < n && isPromising) {
            if(col[n] == col[k] || Math.abs(col[n] - col[k]) == n - k)//차가
같으면 같은 열에 있는것\
              isPromising = false;
            k++;
        }
        return isPromising;
    }
    public static void printQueens() {
        for (int i = 1; i <= N; i++) {// 1부터 N번째 행
            for (int j = 1; j <= N; j++) {// i번째 행의 1부터 N번째 열
                if (col[i] == j) //해당 행의 열에 퀸이 있다
                    System.out.print("Q ");
                else
                    System.out.print("* ");
            }
            System.out.println();
        }
        System.out.println();
```

```
    }
}
```

Code1. Nqueens 코드

i번째 열에 퀸을 놓는 것을 되추적하는지 확인한다. 만약 유망하다면 그곳에 퀸을 놓고 i가 0이 아니면서 퀸을 놓을 수 있는위치라면 I,col[i]를 출력한다.
만약 현재의 행이 유망한 경우 프린트하고 유망하지만 끝까지 도달하지 않는 경우 i+1번째 행에 대해 재귀호출하여 반복해서 유망성을 찾는다.
n번째의 열을 유망성을 검사하여 col[n]과 col[k]의 차가 같으면 같은 열에 있는 것으로 확인하여 퀸을 놓지 못한다.

6.1 결과

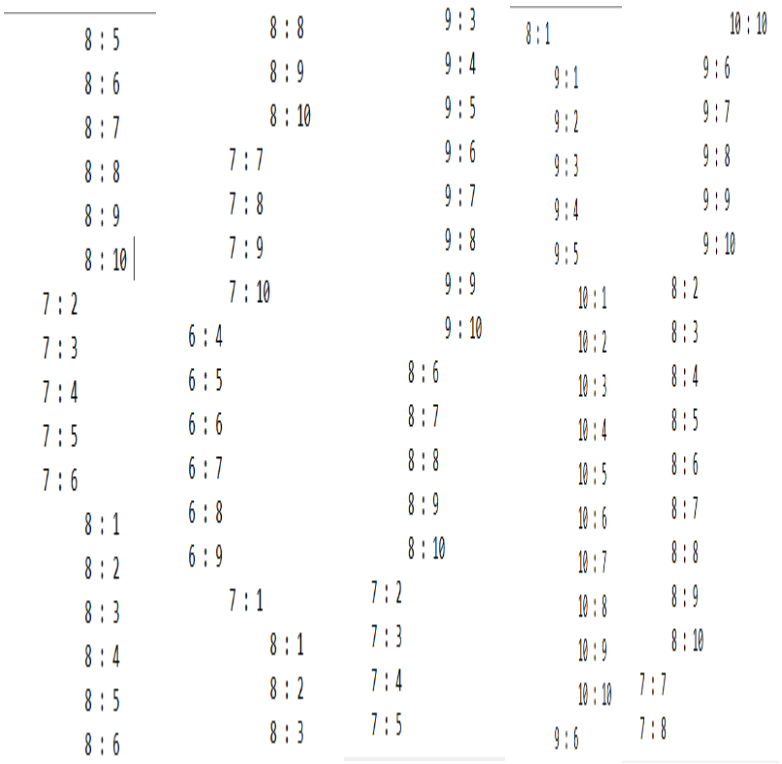


퀸을 놓을 수 있는 위치를 확인 할 수 있다.

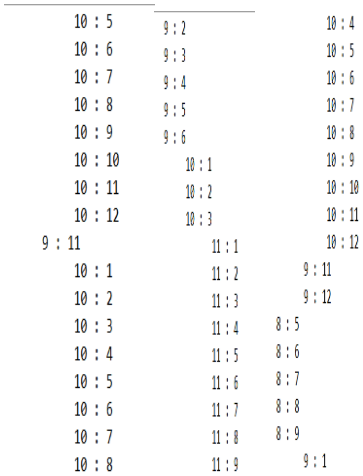
```
int N = 8
```

너무 길어 밑에는 캡쳐하지 못했습니다.

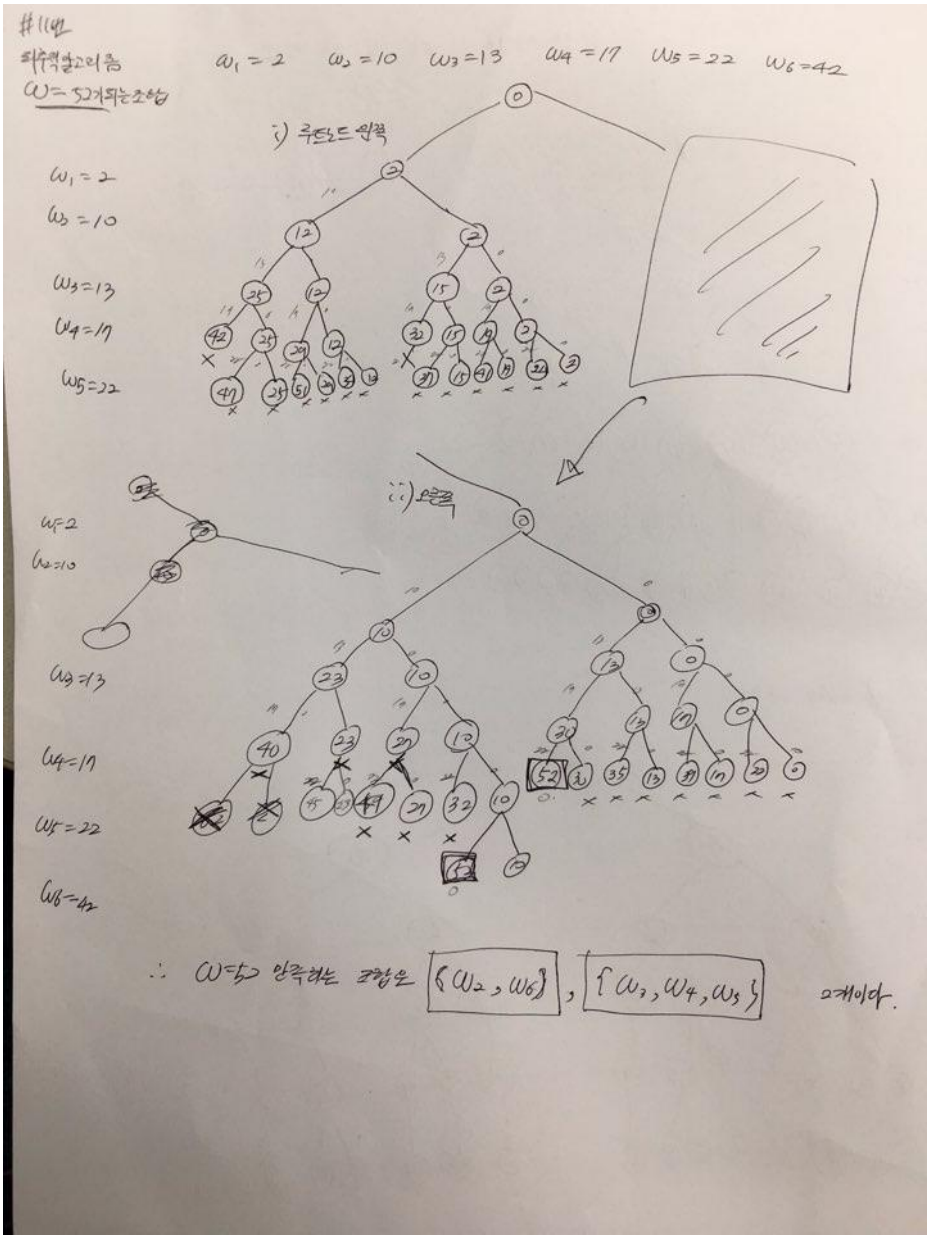
`int N = 10` 너무 길어 밑에는 캡쳐하지 못했습니다.



`int N = 12` 너무 길어 밑에는 캡쳐하지 못했습니다. 끝이 나지않습니다.



7 11번


#11번
최적적 알고리즘
W=52가되는조합
$w_1=2$ $w_2=10$ $w_3=13$ $w_4=17$ $w_5=22$ $w_6=42$
i) 루트노드 왼쪽
ii) 오른쪽
∴ W=52 만족하는 조합은 {w2, w6}, {w3, w4, w5} 이다.