데이터구조입문
정렬

김영민

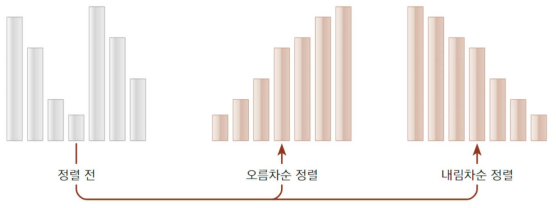
2019. 5. 2

목차

정렬

정렬

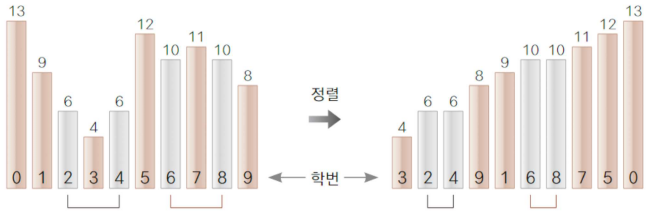
- Sorting
- 이름, 학번, 키 등 학생 항목(key)의 대소 관계에 따라 데이터 집합을 일정한 순서로 나열하는 작업
- 정렬 후에는 검색을 더 쉽게 할 수 있음
- 오름차순(ascending order)과 내림차순(descending order)



[그림 6-1] 오름차순 정렬과 내림차순 정렬

정렬 알고리즘의 안정성

- 안정된(stable) 알고리즘과 그렇지 않은 알고리즘 존재
- 같은 키 값을 갖는 요소들이 정렬 후에도 전후 관계가 유지되는 경우



같은 키 값인 요소의 순서가 정렬 전후에도 유지됩니다.

[그림 6-2] 안정된 정렬

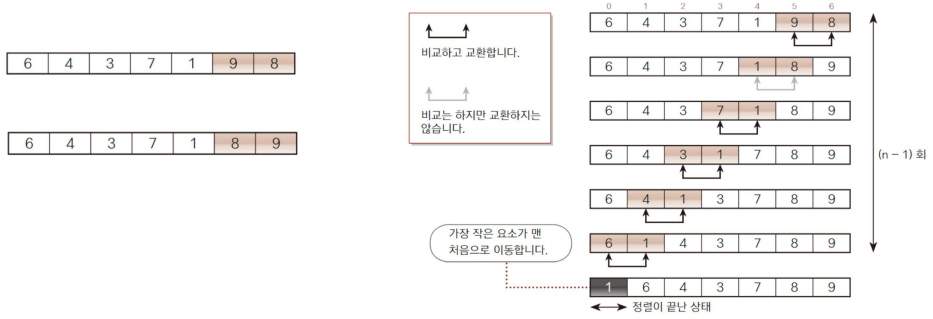
내부 정렬과 외부 정렬

- 내부 정렬: 정렬할 모든 데이터를 하나의 배열에 저장할 수 있는 경우에 사용하는 알고리즘
- 외부 정렬: 정렬할 데이터가 너무 많아서 하나의 배열에 저장할 수 없는 경우에 사용하는 알고리즘
- 정렬 알고리즘의 핵심 요소
  - 교환
  - 선택
  - 삽입

버블 정렬

버블 정렬

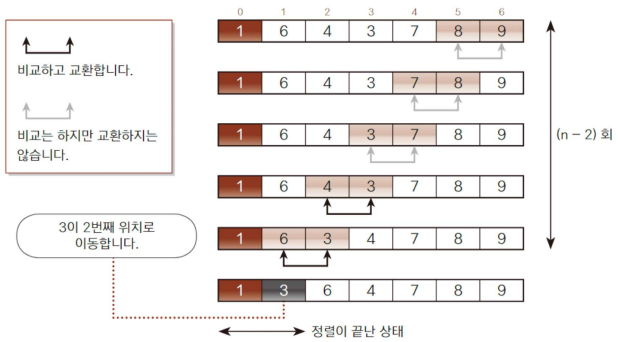
- Bubble sort
  - 이웃한 두 요소의 대소 관계를 비교하여 교환을 반복하는 정렬 알고리즘



[그림 6-3] 버블 정렬의 첫 번째 패스

버블 정렬

- 두번째 패스



[그림 6-4] 버블 정렬의 두 번째 패스 *몇 번째 패스까지?*

버블 정렬 프로그램

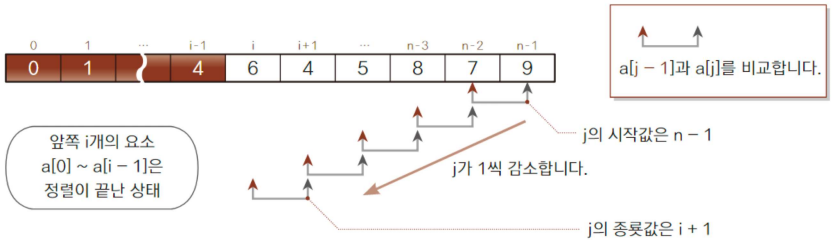
- 변수 i를 0에서 n-2까지 하나씩 증가하면서 n-1회 패스 수행

```
for(int i = 0; i < n – 1; i++) {
  // a[i], a[i + 1], ···, a[n - 1]에 대해
  // 끝에서부터 앞쪽으로 스캔하면서 이웃하는 두 요소를 비교하고 교환합니다.
}
```

- 비교하는 두 요소의 인덱스가 j-1, j 라면 어떻게 변화시켜야 할까?

버블 정렬 인덱스

- 버블 정렬의 i번째 패스



[그림 6-5] 버블 정렬의 i번째 패스

j-1은 i가 되고 j는 i+1이 될 때까지 비교, 교환 가능

비교 횟수

- 비교 횟수

$$(n-1)+(n-2)+\cdots+1 = n(n-1)/2$$

- 실제 교환 횟수는 비교 횟수의 절반인 n(n-1)/4
- 교환을 하는 메서드 안에서 값의 이동이 세 번 일어나므로 3*n(n-1)/4
- 서로 이웃한 요소만 교환하므로 안정적임

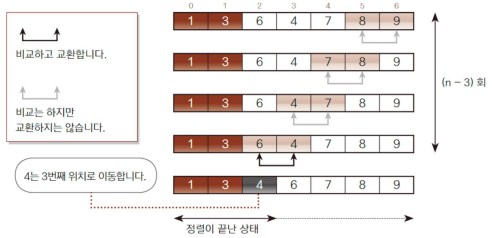
버블 정렬 코드

```
01  package chap06;
02  import java.util.Scanner;
03  // 버블 정렬(버전 1)
04
06    // a[idx1]와 a[idx2]의 값을 바꿉니다.
07    static void swap(int[] a, int idx1, int idx2) {
08      int t = a[idx1];
09      a[idx1] = a[idx2];
10      a[idx2] = t;
11    }
12
13    // 버블 정렬
14    static void bubbleSort(int[] a, int n) {
15      for (int i = 0; i < n - 1; i++)
16        for (int j = n - 1; j > i; j--)
17          if (a[j - 1] > a[j])
18            swap(a, j - 1, j);
19    }
20
21    public static void main(String[] args) {
22      Scanner stdIn = new Scanner(System.in);
23
24      System.out.println("버블 정렬(버전 1)");
25      System.out.print("요솟수 : ");
26      int nx = stdIn.nextInt();
27      int[] x = new int[nx];
28
29      for (int i = 0; i < nx; i++) {
30        System.out.print("x[" + i + "] : ");
31        x[i] = stdIn.nextInt();
32      }
33
34      bubbleSort(x, nx);          // 배열 x를 버블 정렬합니다.
35
36      System.out.println("오름차순으로 정렬했습니다.");
37      for (int i = 0; i < nx; i++)
38        System.out.println("x[" + i + "]=" + x[i]);
39    }
40  }
```

패스

알고리즘 개선 1

- 교환이 더 이상 일어나지 않는 경우



[그림 6-6] 버블 정렬의 세 번째 패스

[그림 6-7] 버블 정렬의 네 번째 패스

알고리즘 개선 1

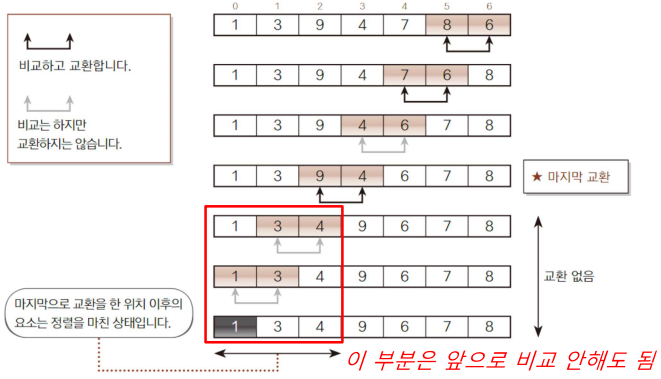
- i번째 패스에서 교환이 수행되지 않으면 그 다음에 정렬을 멈춤
- 교환이 이루어지지 않으면 정렬을 종료할 수 있도록 코드 수정

```
11  // 버블 정렬(버전 2)
12  static void bubbleSort(int[] a, int n) {
13    for (int i = 0; i < n - 1; i++) {
14      int exchg = 0;                  // 패스의 교환 횟수를 기록합니다.
15      for (int j = n - 1; j > i; j--)
16        if (a[j - 1] > a[j]) {
17          swap(a, j - 1, j);
18          exchg++;
19        }
20      if (exchg == 0) break;          // 교환이 이루어지지 않으면 종료합니다.
21    }
22  }
```

패스

알고리즘 개선 2

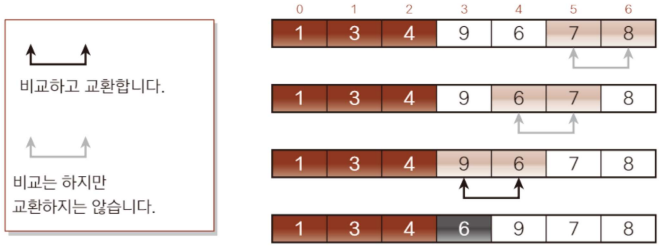
- 중간에 특정 요소부터 그 앞으로 교환이 일어나지 않는 경우



[그림 6-8] 버블 정렬의 첫 번째 패스

알고리즘 개선 2

- 두 번째 패스



[그림 6-9] 버블 정렬의 두 번째 패스

알고리즘 개선 2

- 코드 개선
- last : 마지막으로 교환한 두 요소 가운데 오른쪽 요소의 인덱스

```
11  // 버블 정렬(버전 3)
12  static void bubbleSort(int[] a, int n) {
13    int k = 0;                      // a[k]보다 앞쪽은 정렬을 마친 상태
14    while (k < n - 1) {
15      int last = n - 1;             // 마지막으로 요소를 교환한 위치
16      for (int j = n - 1; j > k; j--)
17        if (a[j - 1] > a[j]) {
18          swap(a, j - 1, j);
19          last = j;
20        }
21      k = last;
22    }
23  }
```

패스

last 값을 k에 대입

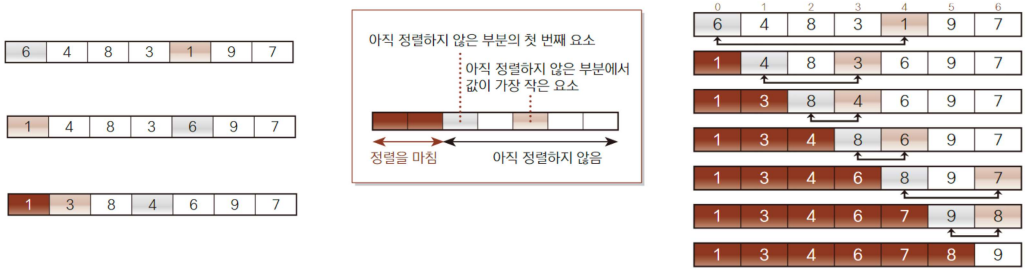
다음 번 패스에서 마지막으로 비교할 두 요소는 a[k]와 a[k-1]

단순 선택 정렬

단순 선택 정렬

- Straight selection sort
- 가장 작은 요소부터 정렬하는 알고리즘

가장 작은 요소를 아직 정렬하지 않은 부분의 첫 번째 요소와 교환



[그림 6-10] 단순 선택 정렬 과정

교환 과정

- 단순 선택 정렬 교환 과정

1. 아직 정렬하지 않은 부분에서 가장 작은 키의 값(a[min])을 선택합니다.
2. a[min]과 아직 정렬하지 않은 부분의 첫 번째 요소를 교환합니다.

```
for(int i = 0; i < n - 1; i++) {
  // min = a[i], …, a[n – 1]에서 가장 작은 값을 가지는 요소의 인덱스
  // a[i]와 a[min]의 값을 교환
}
```

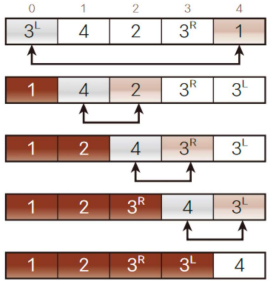

단순 선택 정렬 코드

- 코드

```
11  // 단순 선택 정렬
12  static void selectionSort(int[] a, int n) {
13    for (int i = 0; i < n - 1; i++) {
14      int min = i;          // 아직 정렬되지 않은 부분에서 가장 작은 요소의 인덱스를 기록합니다.
15      for (int j = i + 1; j < n; j++)
16        if (a[j] < a[min])
17          min = j;
18      swap(a, i, min);      // 아직 정렬되지 않은 부분의 첫 요소와 가장 작은 요소를 교환합니다.
19    }
20  }
```


비교 횟수, 안정성

- 요소값 비교 횟수는 버블 정렬과 마찬가지로 n(n-1)/2
- 서로 떨어져 있는 요소를 교환하는 것이기 때문에 안정적이지 않음



[그림 6-11] 3^L과 3^R의 위치가 뒤바뀌는 모습

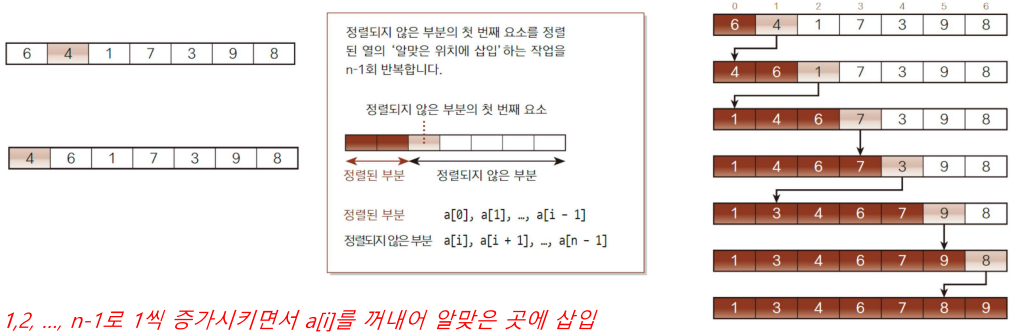
단순 삽입 정렬

단순 삽입 정렬

- Straight inserting sort
- 선택한 요소를 앞쪽에 알맞은 위치에 삽입하는 작업을 반복하여 정렬하는 알고리즘
- 단순 선택 정렬은 가장 작은 요소를 선택하기 때문에 다름

정렬 과정

- 아직 정렬되지 않은 부분의 첫번째 요소를 정렬된 부분의 알맞은 위치에 삽입



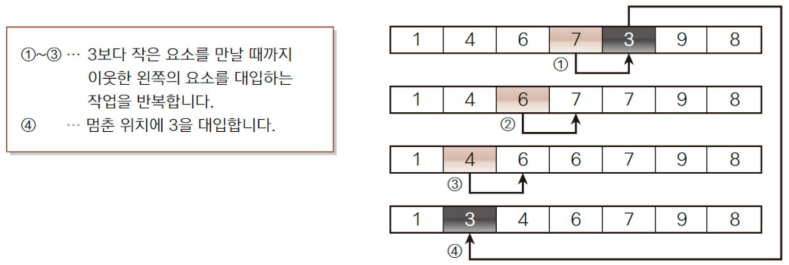
i를 1,2, …, n-1로 1씩 증가시키면서 a[i]를 꺼내어 알맞은 곳에 삽입

[그림 6-12] 단순 삽입 정렬 과정

삽입을 위한 알맞은 위치 찾기

- 알맞은 위치 찾기

3이 선택된 경우



[그림 6-13] 단순 삽입 정렬에서 요소 3의 삽입 과정

tmp에 a[i] 대입, 반복 제어용 변수 j에 i-1 대입

반복 조건

- 아래 조건 중 하나를 만족할 때까지 j를 1씩 감소시키면서 대입하는 작업 반복

1. 정렬된 열의 왼쪽 끝에 도달합니다.
2. tmp보다 작거나 같은 key를 갖는 항목 a[j]를 발견합니다.

- 위의 조건을 다시 써서 반복하기 위한 조건으로 바꾸면,

1. j가 0보다 큽니다.
2. a[j – 1] 값이 tmp보다 큽니다.

단순 삽입 정렬 코드

```
01  package chap06;
02  import java.util.Scanner;
03  // 단순 삽입 정렬
04
05  class InsertionSort {
06    // 단순 삽입 정렬
07    static void insertionSort(int[] a, int n) {
08      for (int i = 1; i < n; i++) {
09        int j;
10        int tmp = a[i];
11        for (j = i; j > 0 && a[j - 1] > tmp; j--)
12          a[j] = a[j - 1];
13        a[j] = tmp;
14      }
15    }
16
17    public static void main(String[] args) {
18      Scanner stdIn = new Scanner(System.in);
20      System.out.println("단순 삽입 정렬");
21      System.out.print("요솟수 : ");
22      int nx = stdIn.nextInt();
23      int[] x = new int[nx];
24
25      for (int i = 0; i < nx; i++) {
26        System.out.print("x[" + i + "] : ");
27        x[i] = stdIn.nextInt();
28      }
29
30      insertionSort(x, nx);        // 배열 x를 단순 삽입 정렬
31
32      System.out.println("오름차순으로 정렬했습니다.");
33      for (int i = 0; i < nx; i++)
34        System.out.println("x[" + i + "]=" + x[i]);
35    }
36  }
```


- 이렇게 구현한 단순 삽입 정렬 알고리즘은 떨어져 있는 요소들이 서로 뒤바뀌지 않은
- 요소 비교 횟수 및 교환 횟수는 n^2/2
- 단순 삽입 정렬을 셔틀 정렬(shuttle sort)이라고도 함
- 지금까지의 세 가지 단순 정렬(버블, 선택, 삽입)의 시간 복잡도는 모두 O(n^2)

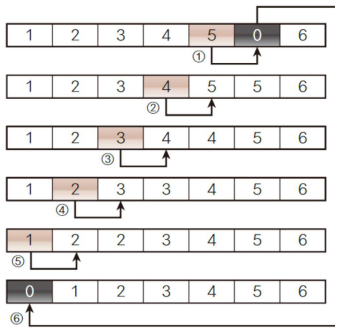
셸 정렬

셸 정렬

- 단순 삽입 정렬의 장점을 살리고 단점은 보완한 방법
- 단순 삽입 정렬 예

①~⑤ … 0보다 작은 요소를 만날 때까지 왼쪽 요소를 하나씩 대입하는 작업을 반복합니다.
⑥ … 멈춘 위치에서 0을 대입합니다.

이미 정렬이 된 5까지는 빠르게 지나가나
0을 삽입하려면 6회에 걸쳐 요소를 이동해야함



[그림 6-14] 단순 삽입 정렬에서 요소 0의 이동 과정

셸 정렬

- 단순 삽입 정렬의 특징

  ⓐ 정렬을 마쳤거나 정렬을 마친 상태에 가까우면 정렬 속도가 매우 빨라집니다(장점).

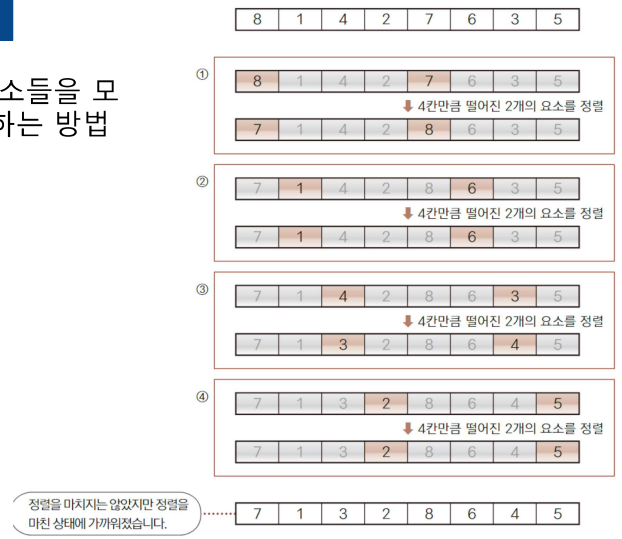
  ⓑ 삽입할 위치가 멀리 떨어져 있으면 이동(대입)해야 하는 횟수가 많아집니다(단점).

- 셸 정렬(shell sort) 위의 장점을 살리고 단점은 보완한 정렬 알고리즘
  - 배열의 요소를 그룹으로 나누어 그룹별로 단순 삽입 정렬 수행
  - 그 그룹을 합치면서 정렬을 반복

4-정렬

- 4-정렬 : 네 칸 만큼 떨어진 요소들을 모아 그룹을 4개로 나누어 정렬하는 방법

네 개의 그룹
{8,7}
{1,6}
{4,3}
{2,5}



[그림 6-15] 셸 정렬의 4-정렬

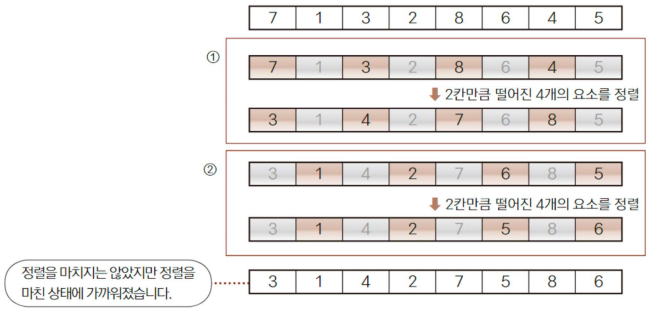
2-정렬

- 2-정렬: 2칸 만큼 떨어진 요소를 모아 두 그룹으로 나누어 정렬

두 개의 그룹
{7,3,8,4}
{1,2,6,5}

⬇

{3,4,7,8}
{1,2,5,6}



[그림 6-16] 셸 정렬의 2-정렬

셸 정렬 전체

- 마지막에 1-정렬 수행

1. 2개 요소에 대해 '4-정렬'을 합니다(4개의 그룹).
2. 4개 요소에 대해 '2-정렬'을 합니다(2개의 그룹).
3. 8개 요소에 대해 '1-정렬'을 합니다(1개의 그룹).

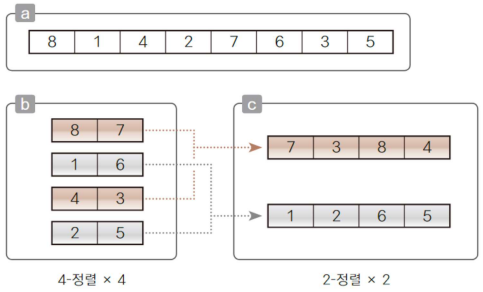
[그림 6-17] 셸 정렬

셸 정렬 코드

```
01  package chap06;
02  import java.util.Scanner;
03  // 셸 정렬(버전 1)
04
05  class ShellSort {
06    // 셸 정렬
07    static void shellSort(int[] a, int n) {
08      for (int h = n / 2; h > 0; h /= 2)
09        for (int i = h; i < n; i++) {
10          int j;
11          int tmp = a[i];
12          for (j = i - h; j >= 0 && a[j] > tmp; j -= h)
13            a[j + h] = a[j];
14          a[j + h] = tmp;
15        }
16    }
17
18    public static void main(String[] args) {
19      Scanner stdIn = new Scanner(System.in);
20
21      System.out.println("셸 정렬(버전 1)");
22      System.out.print("요솟수 : ");
23      int nx = stdIn.nextInt();
24      int[] x = new int[nx];
25
26      for (int i = 0; i < nx; i++) {
27        System.out.print("x[" + i + "] : ");
28        x[i] = stdIn.nextInt();
29      }
30
31      shellSort(x, nx);        // 배열 x를 셸 정렬
32
33      System.out.println("오름차순으로 정렬했습니다.");
34      for (int i = 0; i < nx; i++)
35        System.out.println("x[" + i + "]=" + x[i]);
36    }
37  }
```


증분값(h값) 선택

- h값을 어떤 수열로 감소시키는 것이 좋을까?



[그림 6-18] 셸 정렬의 그룹 분할 과정(h = 4, 2, 1)

이 상태에서는 처음 나눌 때의 갈색 그룹과 회색 그룹이 서로 섞이지 않음

h값이 배수가 되지 않도록 하여 요소가 충분히 섞일 수 있도록 해준다.

증분값(h값) 선택

- 다음 수열을 이용하면 셸 정렬을 간단히 수행할 수 있음

  h = ⋯, 121, 40, 13, 4, 1

- (거꾸로 봤을 때) 1부터 시작, 3배 한 값에 1을 더하여 다음 숫자를 생성

- h의 초기값은 배열의 요소수 n을 9로 나눈 값을 넘지 않도록 설정

셸 정렬 코드 수정


```
01  package chap06;
02  import java.util.Scanner;
03  // 셸 정렬(버전 2, h의 값은 ⋯, 40, 13, 4, 1)
04
05  class ShellSort2 {
06    // 셸 정렬
07    static void shellSort(int[] a, int n) {
08      int h;
09      for (h = 1; h < n / 9; h = h * 3 + 1)
10        ;                                          ← 1
11
12      for ( ; h > 0; h /= 3)
13        for (int i = h; i < n; i++) {
14          int j;
15          int tmp = a[i];
16          for (j = i - h; j >= 0 && a[j] > tmp; j -= h)   ← 2
17            a[j + h] = a[j];
18          a[j + h] = tmp;
19        }
20    }
21
22    public static void main(String[] args) {
23      Scanner stdIn = new Scanner(System.in);
24
25      System.out.println("셸 정렬(버전 2)");
26      System.out.print("요솟수 : ");
27      int nx = stdIn.nextInt();
28      int[] x = new int[nx];
29
30      for (int i = 0; i < nx; i++) {
31        System.out.print("x[" + i + "] : ");
32        x[i] = stdIn.nextInt();
33      }
34
35      shellSort(x, nx);        // 배열 x를 셸 정렬
36
37      System.out.println("오름차순으로 정렬했습니다.");
38      for (int i = 0; i < nx; i++)
39        System.out.println("x[" + i + "]=" + x[i]);
40    }
41  }
```