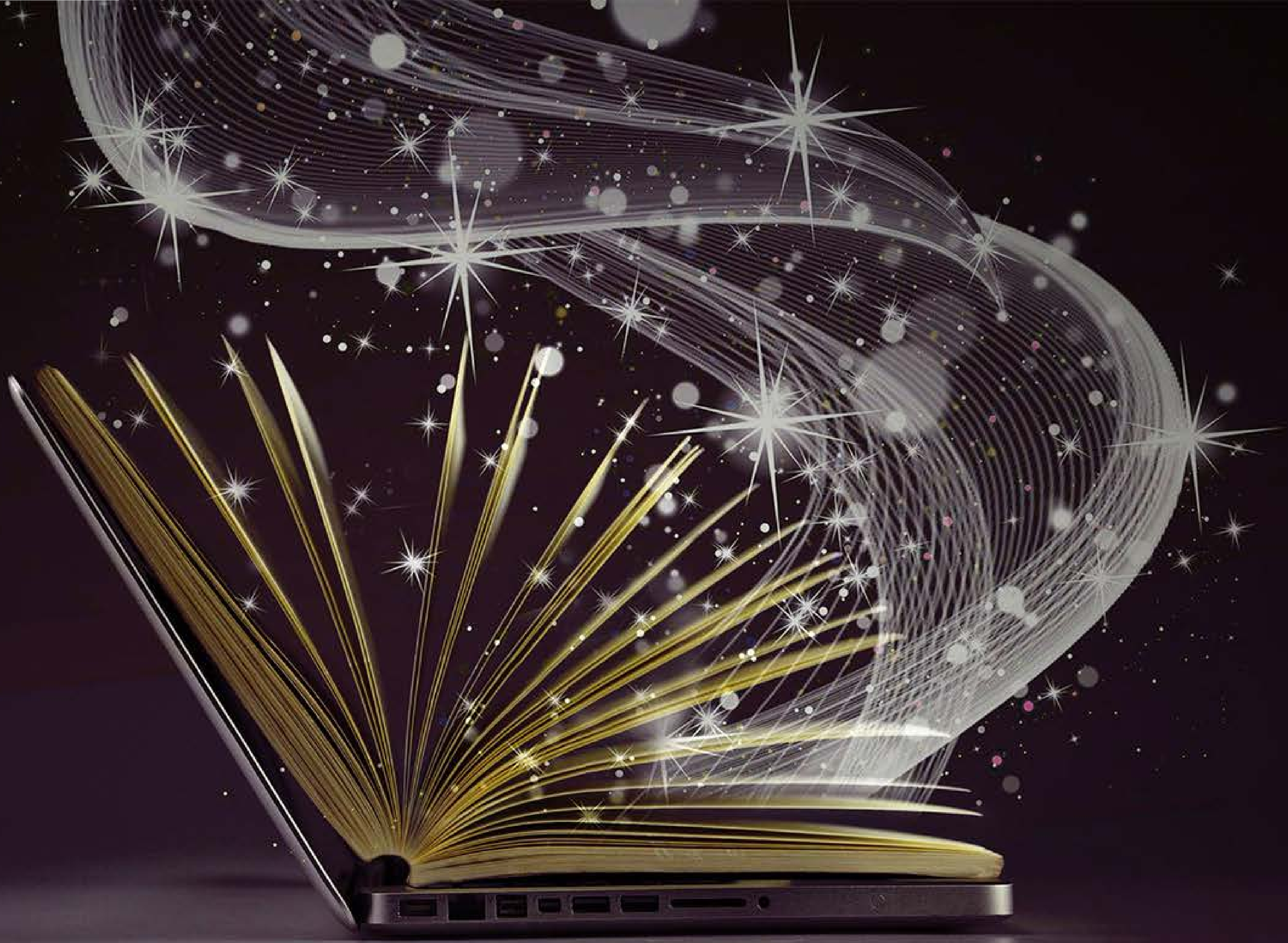


Sorting

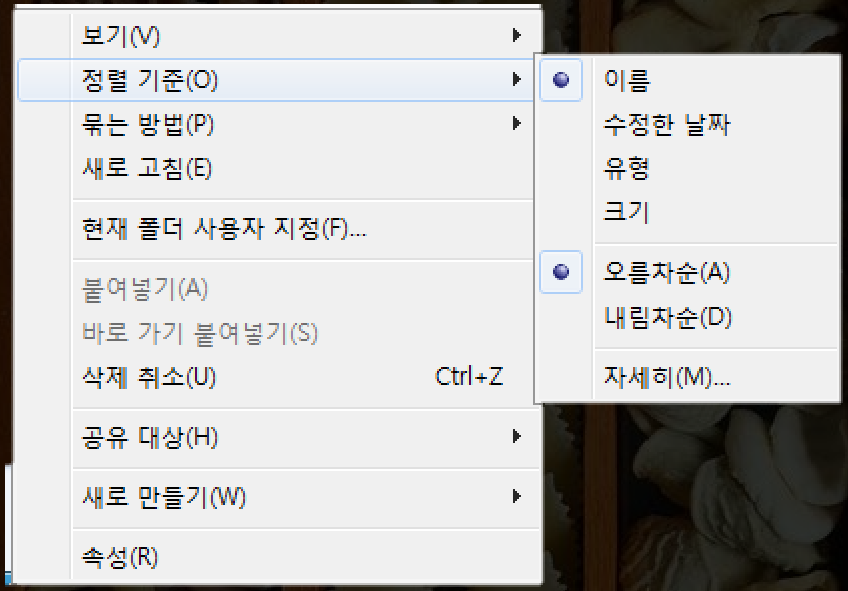
Ja-Hee Kim

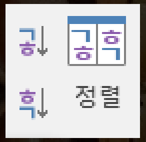
Agenda

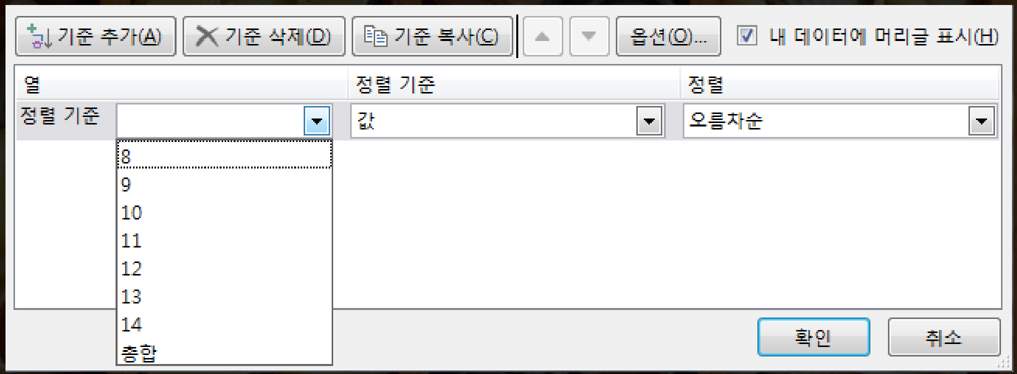
Introduction of Sorting

Definition

- Sorting: any process of arranging items in sequence or in sets
  - Ascending: small → big
  - Descending: big → small
- Sorting algorithm: any algorithm for arranging elements in lists







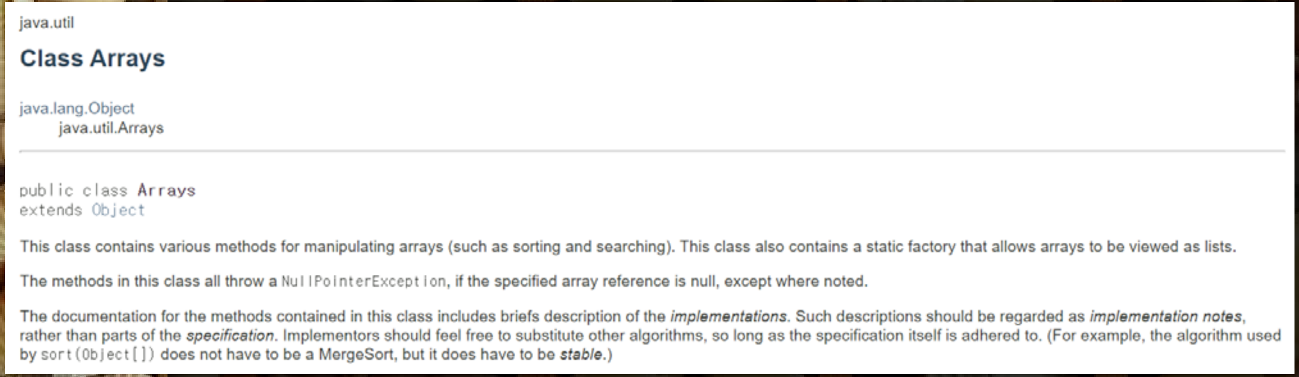
열	정렬 기준	정렬
정렬 기준	값	오름차순

Classification of Sorting Algorithm

- Pairwise (comparison) vs Distribute
- Internal Sort vs External Sort

Sorting in java

- The class "Java.util.Arrays" contains various methods for manipulating arrays (such as s orting and searching).
- This class includes several methods for arrays: binearysearch, sort, etc.
- It provides sort algorithms for various types into ascending order.
- The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, an d Joshua Bloch.



java.util

Class Arrays

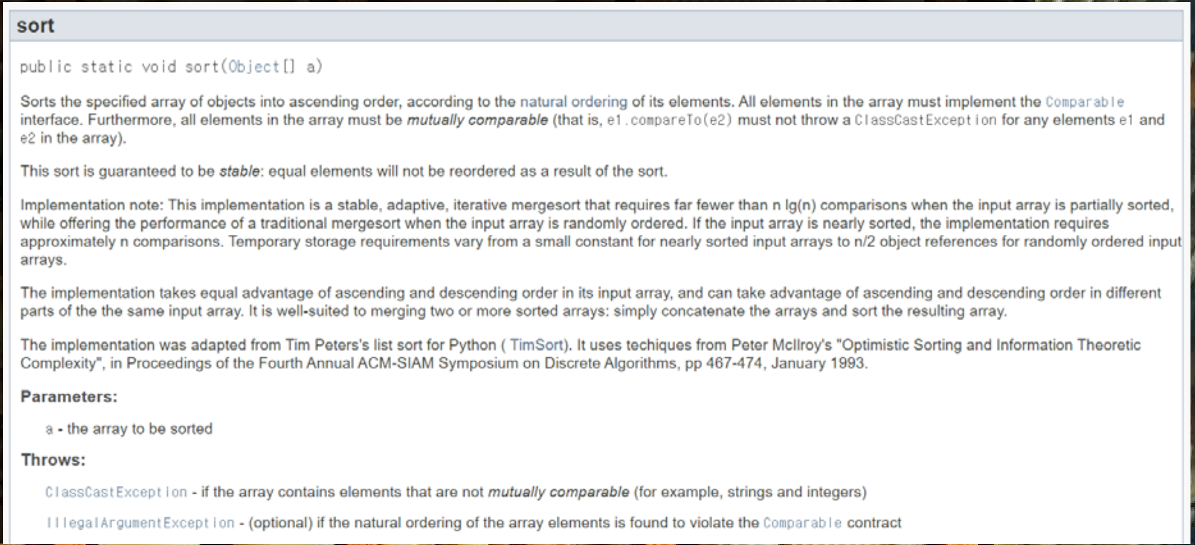
java.lang.Object
    java.util.Arrays

```
public class Arrays
extends Object
```

This class contains various methods for manipulating arrays (such as sorting and searching). This class also contains a static factory that allows arrays to be viewed as lists.

The methods in this class all throw a `NullPointerException`, if the specified array reference is null, except where noted.

The documentation for the methods contained in this class includes briefs description of the *implementations*. Such descriptions should be regarded as *implementation notes*, rather than parts of the *specification*. Implementors should feel free to substitute other algorithms, so long as the specification itself is adhered to. (For example, the algorithm used by `sort(Object[])` does not have to be a MergeSort, but it does have to be *stable*.)



sort

```
public static void sort(Object[] a)
```

Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array must implement the `Comparable` interface. Furthermore, all elements in the array must be ***mutually comparable*** (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the array).

This sort is guaranteed to be ***stable***: equal elements will not be reordered as a result of the sort.

Implementation note: This implementation is a stable, adaptive, iterative mergesort that requires far fewer than n lg(n) comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately n comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to n/2 object references for randomly ordered input arrays.

The implementation takes equal advantage of ascending and descending order in its input array, and can take advantage of ascending and descending order in different parts of the the same input array. It is well-suited to merging two or more sorted arrays: simply concatenate the arrays and sort the resulting array.

The implementation was adapted from Tim Peters's list sort for Python (TimSort). It uses techiques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.

Parameters:

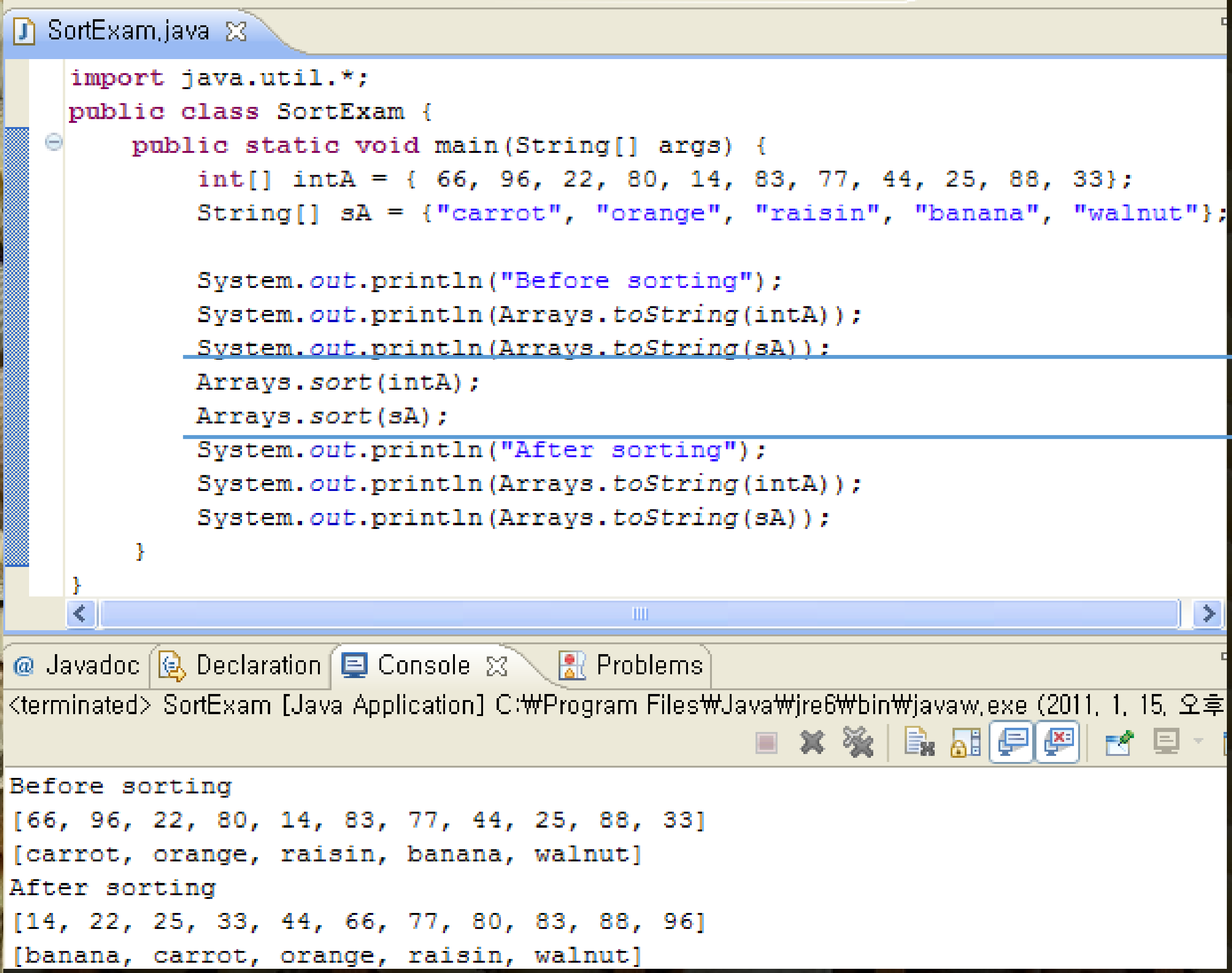
`a` - the array to be sorted

Throws:

`ClassCastException` - if the array contains elements that are not ***mutually comparable*** (for example, strings and integers)

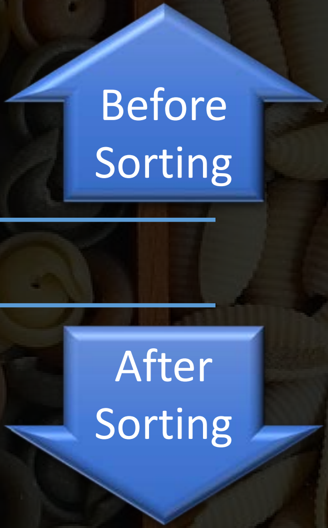
`IllegalArgumentException` - (optional) if the natural ordering of the array elements is found to violate the `Comparable` contract

Example of Arrays.sort



```java
import java.util.*;
public class SortExam {
    public static void main(String[] args) {
        int[] intA = { 66, 96, 22, 80, 14, 83, 77, 44, 25, 88, 33};
        String[] sA = {"carrot", "orange", "raisin", "banana", "walnut"};

        System.out.println("Before sorting");
        System.out.println(Arrays.toString(intA));
        System.out.println(Arrays.toString(sA));
        Arrays.sort(intA);
        Arrays.sort(sA);
        System.out.println("After sorting");
        System.out.println(Arrays.toString(intA));
        System.out.println(Arrays.toString(sA));
    }
}
```

```
<terminated> SortExam [Java Application] C:₩Program Files₩Java₩jre6₩bin₩javaw.exe (2011. 1. 15. 오후
Before sorting
[66, 96, 22, 80, 14, 83, 77, 44, 25, 88, 33]
[carrot, orange, raisin, banana, walnut]
After sorting
[14, 22, 25, 33, 44, 66, 77, 80, 83, 88, 96]
[banana, carrot, orange, raisin, walnut]
```



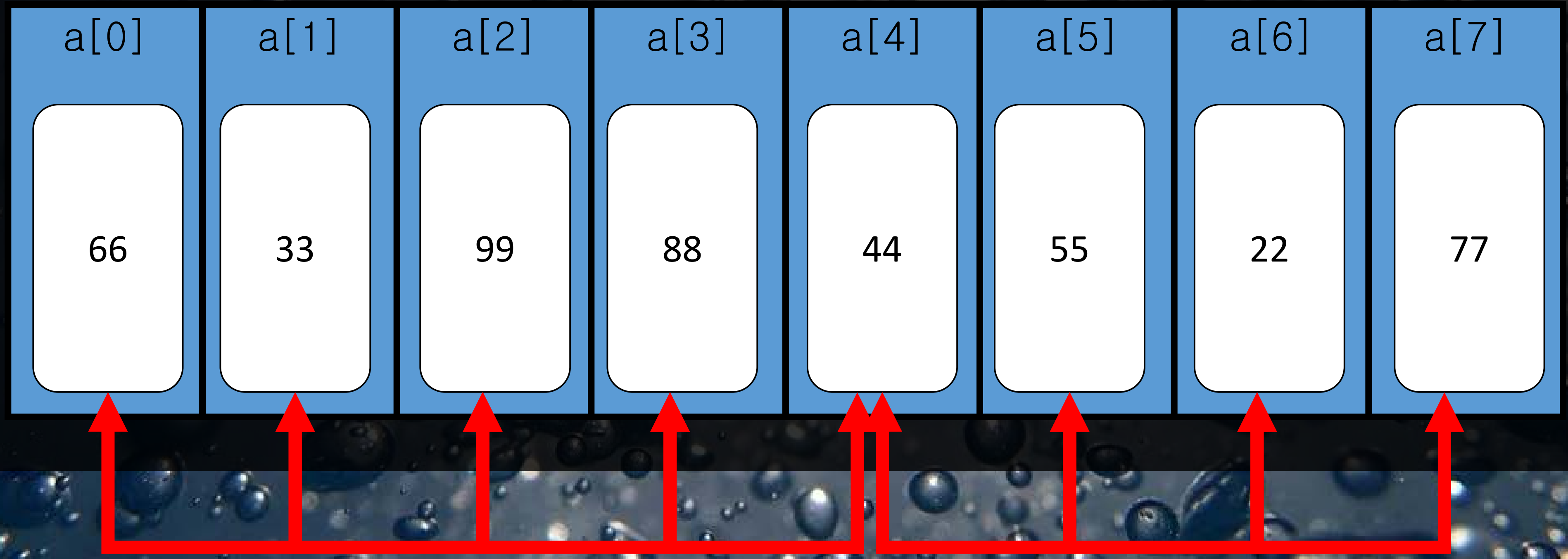
Bubble Sort

Bubble sorting

- It works by comparing adjacent elements in the array and swapping them whenever they are out of order.
- Algorithm

```
1. Repeat steps 2-3 for $i=n-1$ down to 1
2.     Repeat step 3 for j=0 up to i-1
3.         If $a_j \le a_{j+1}$, swap them
```

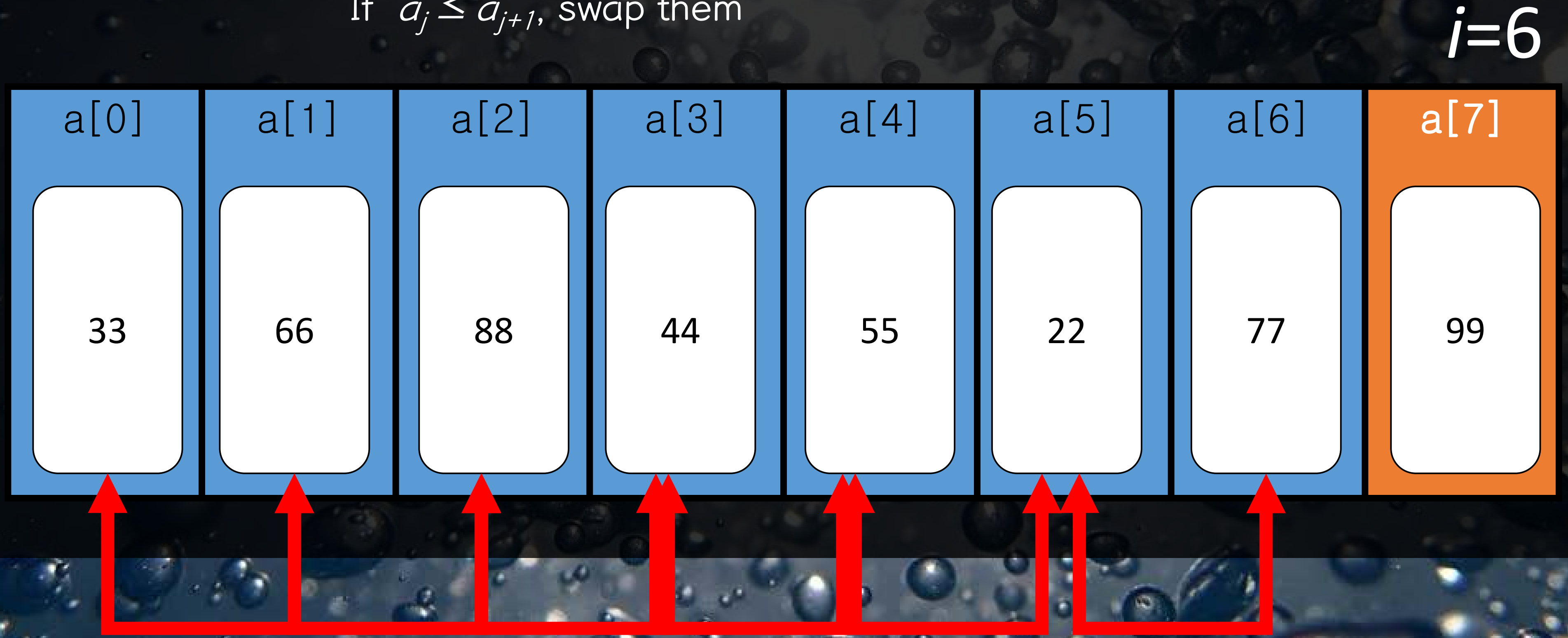
i=7



Bubble sorting

- It works by comparing adjacent elements in the array and swapping them whenever they are out of order.
- Algorithm

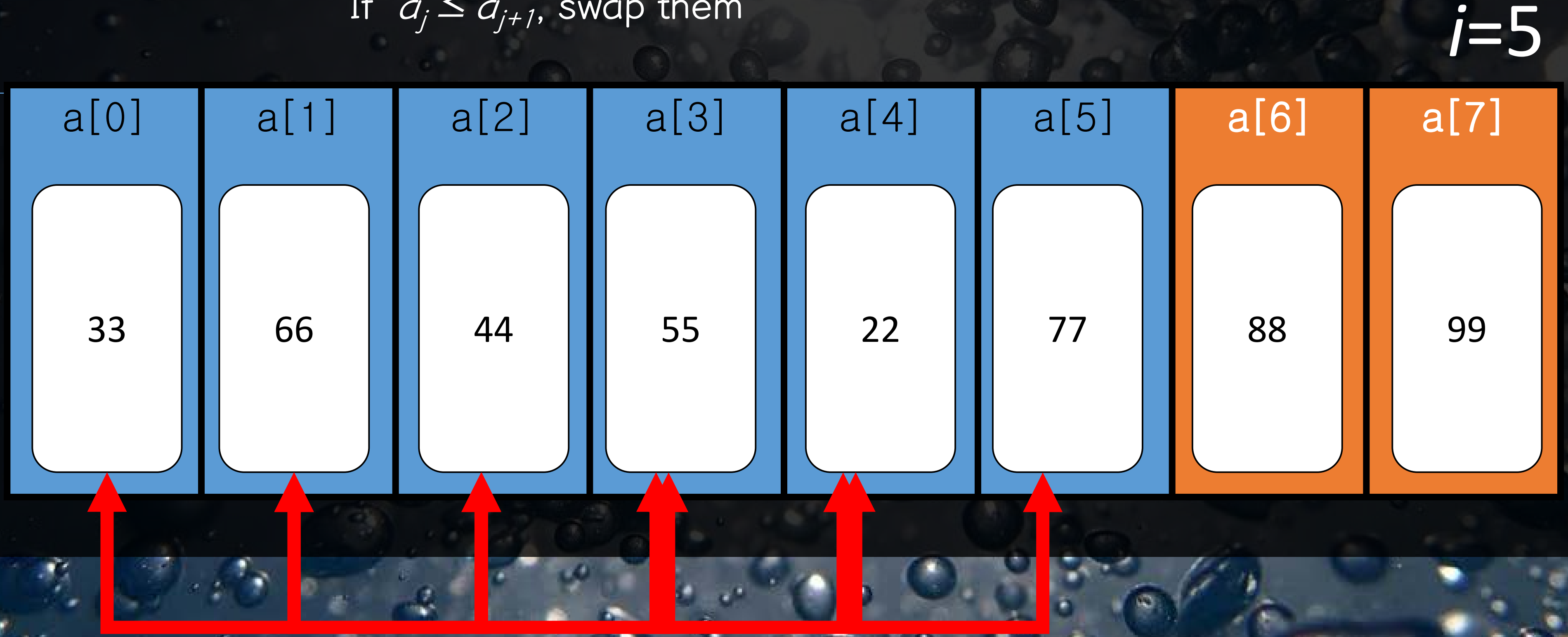
```
1. Repeat steps 2-3 for $i=n-1$ down to 1
2.     Repeat step 3 for j=0 up to i-1
3.         If $a_j \leq a_{j+1}$, swap them
```



Bubble sorting

- It works by comparing adjacent elements in the array and swapping them whenever they are out of order.
- Algorithm

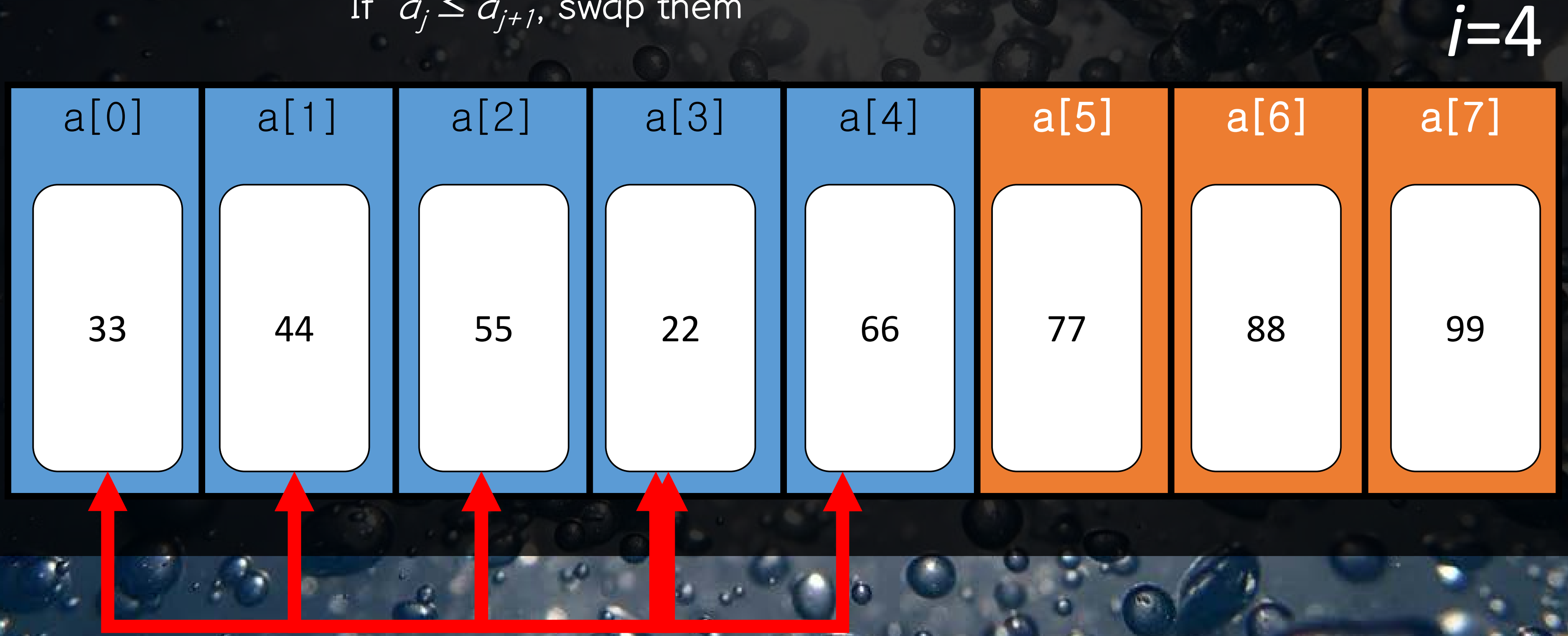
```
1. Repeat steps 2-3 for $i=n-1$ down to 1
2.         Repeat step 3 for j=0 up to i-1
3.                 If $a_j \leq a_{j+1}$, swap them
```



Bubble sorting

- It works by comparing adjacent elements in the array and swapping them whenever they are out of order.
- Algorithm

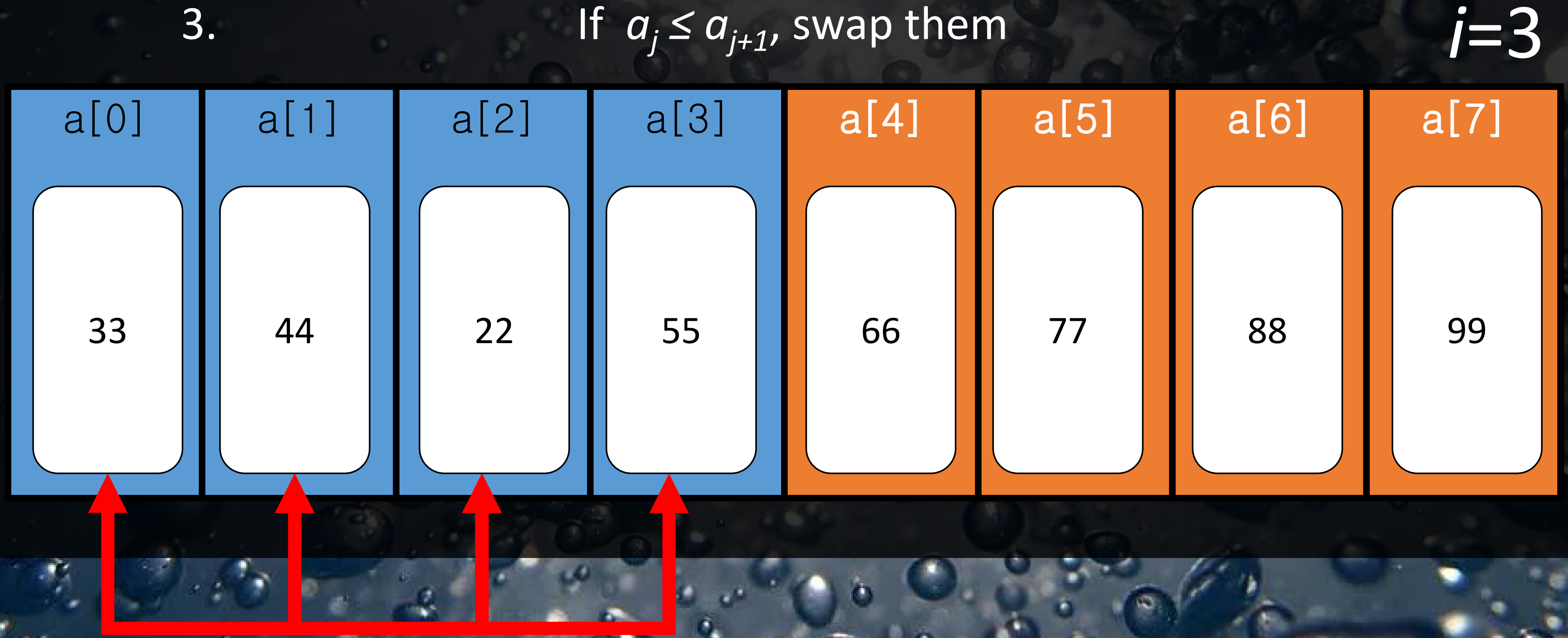
```
1. Repeat steps 2-3 for $i=n-1$ down to 1
2.     Repeat step 3 for j=0 up to i-1
3.         If $a_j \leq a_{j+1}$, swap them
```



Bubble sorting

- It works by comparing adjacent elements in the array a nd swapping them whenever they are out of order.
- Algorithm

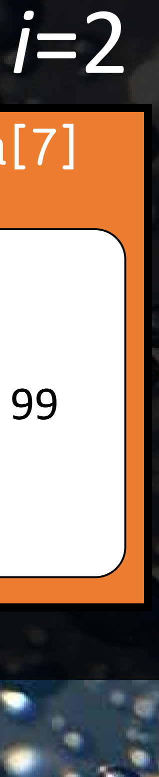
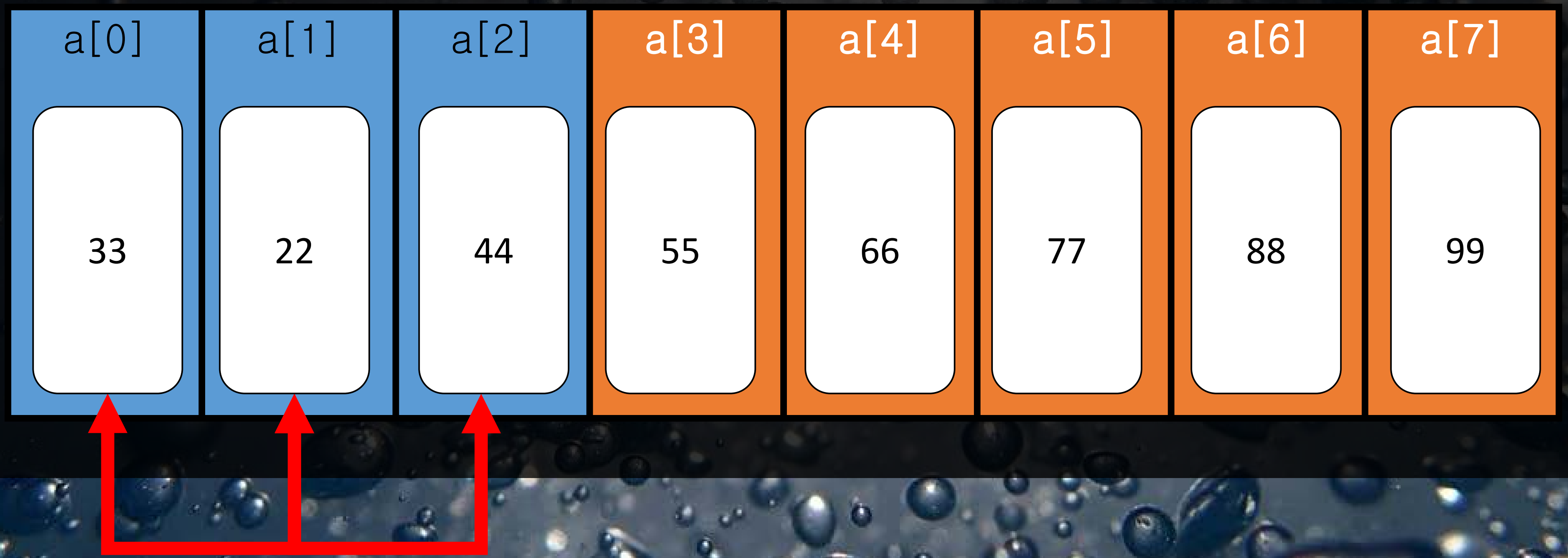
```
1.  Repeat steps 2-3 for $i=n-1$ down to 1
2.      Repeat step 3 for j=0 up to i-1
3.          If $a_j \leq a_{j+1}$, swap them
```



Bubble sorting

- It works by comparing adjacent elements in the array and swapping them whenever they are out of order.
- Algorithm

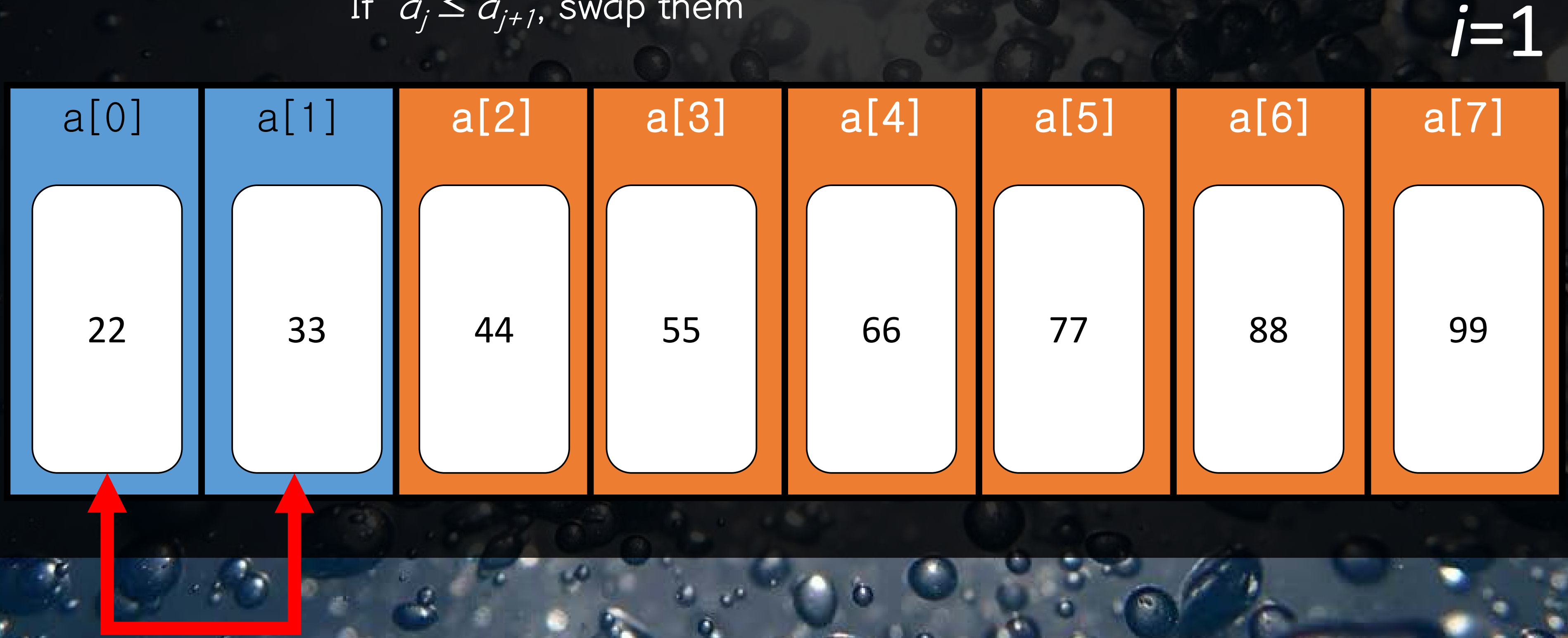
```
1. Repeat steps 2-3 for $i=n-1$ down to 1
2.     Repeat step 3 for j=0 up to i-1
3.         If $a_j \leq a_{j+1}$, swap them
```

Bubble sorting

- It works by comparing adjacent elements in the array and swapping them whenever they are out of order.
- Algorithm

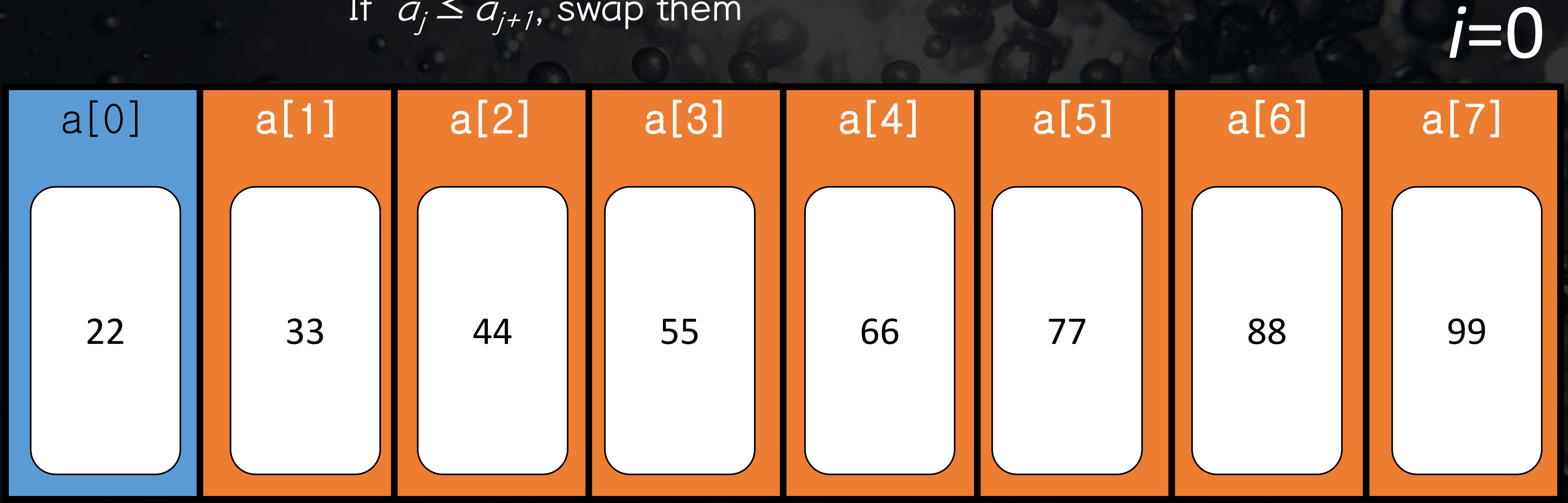
```
1. Repeat steps 2-3 for $i=n-1$ down to 1
2.     Repeat step 3 for j=0 up to i-1
3.         If $a_j \leq a_{j+1}$, swap them
```



Bubble sorting

- It works by comparing adjacent elements in the array and swapping them whenever they are out of order.
- Algorithm

```
1. Repeat steps 2-3 for $i=n-1$ down to 1
2.     Repeat step 3 for j=0 up to i-1
3.         If $a_j \leq a_{j+1}$, swap them
```



Bubble sorting

Iterative bubble sort

```java
public T[] bubbleI() {
    T[] result = array.clone();
    for(int i = result.length -1 ; i >0 ; i--) {
        for(int j=0; j < i ; j++)
            if(result[j].compareTo(result[j+1])>0) swap(result, j, j+1);
    }
    return result;
}
```

$$\sum_{i=n-1}^{1} i = \frac{n(n-1)}{2} = O(n^2)$$

Recursive bubble sort

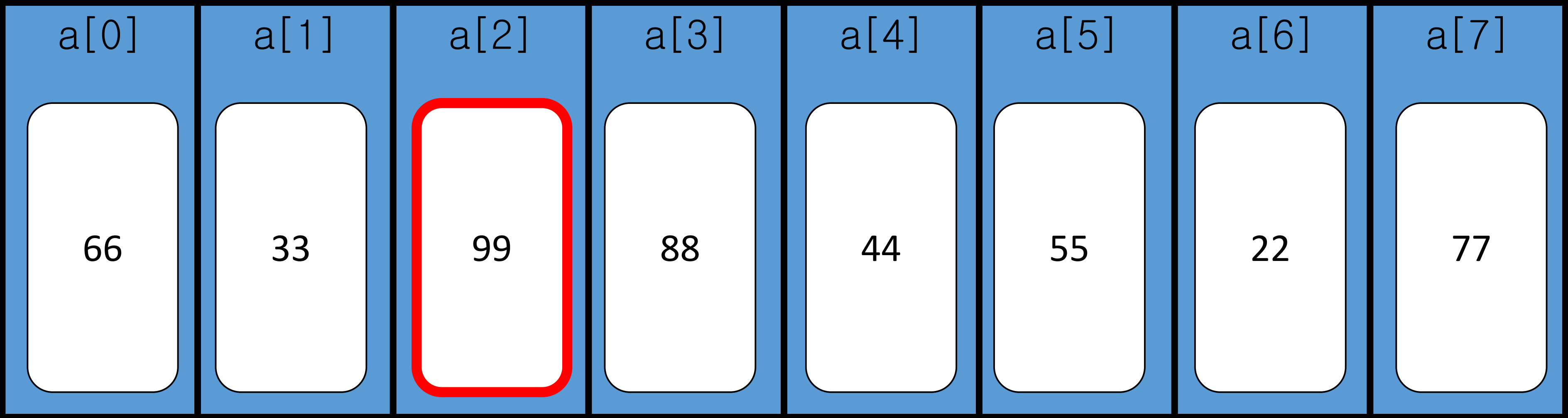
```java
public T[] bubbleR() {
    T[] result = array.clone();
    if(array.length<2) return result;
    for(int i = result.length -1 ; i > 0 ; i--)
        bubbleSort(result, 0, i);
    return result;
}
private void bubbleSort(T[] a, int n1, int n2) {
    if(n1 >= n2) return;
    if(a[n1].compareTo(a[n1+1])>0) swap(a, n1, n1+1);
    bubbleSort(a, n1+1, n2);
}
```

Selection Sort

Selection sorting

- Each pass selects the largest among the remaining unsorted elements and moves it into its correct position.
- Algorithm
  1. Repeat steps 2 for i=n-1 down to 1
  2. Swap a_i with max{a_0, ⋯, a_i}.

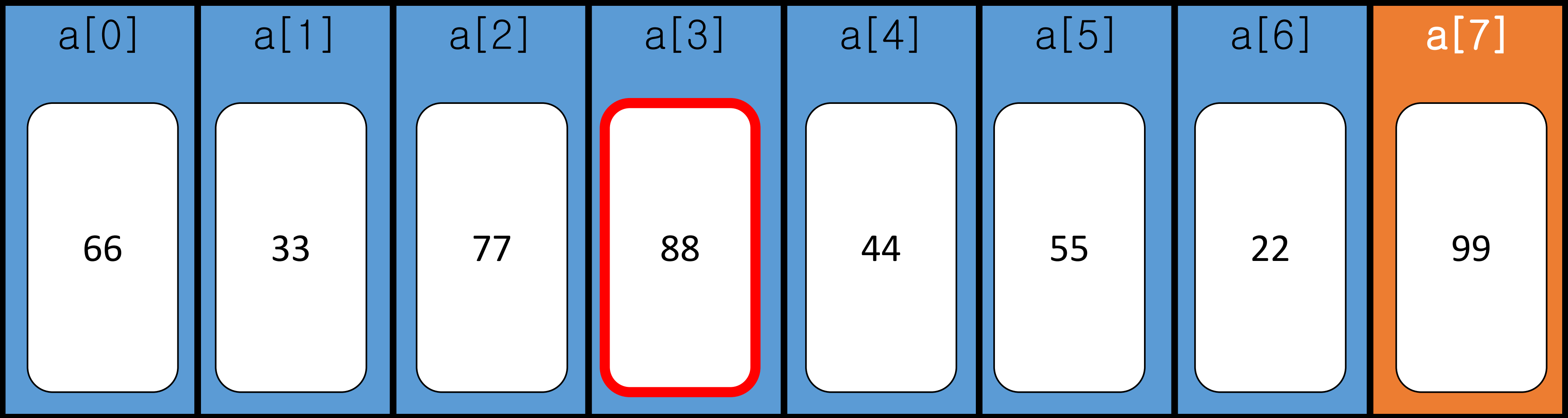
i=7



Selection sorting

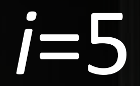
- Each pass selects the largest among the remaining unsorted elements and moves it into its correct position.
- Algorithm
  1. Repeat steps 2 for i=n-1 down to 1
  2. Swap a_i with max$\{a_0, \cdots, a_i\}$.

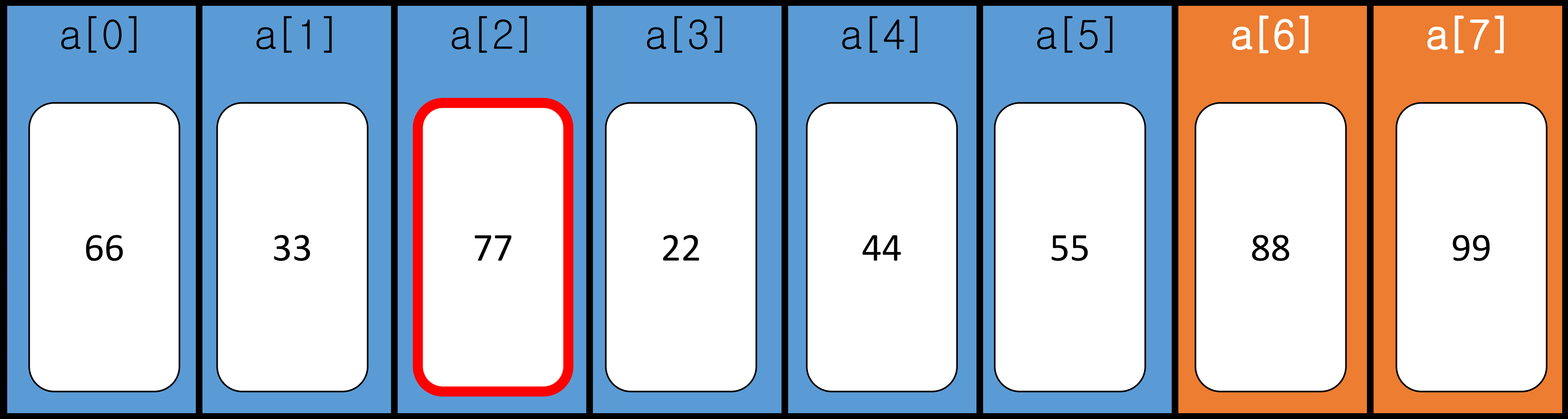
i=6



Selection sorting

- Each pass selects the largest among the remaining unsorted elements and moves it into its correct position.
- Algorithm
  1. Repeat steps 2 for i=n-1 down to 1
  2. Swap a_i with $\max\{a_0, \cdots, a_i\}$.

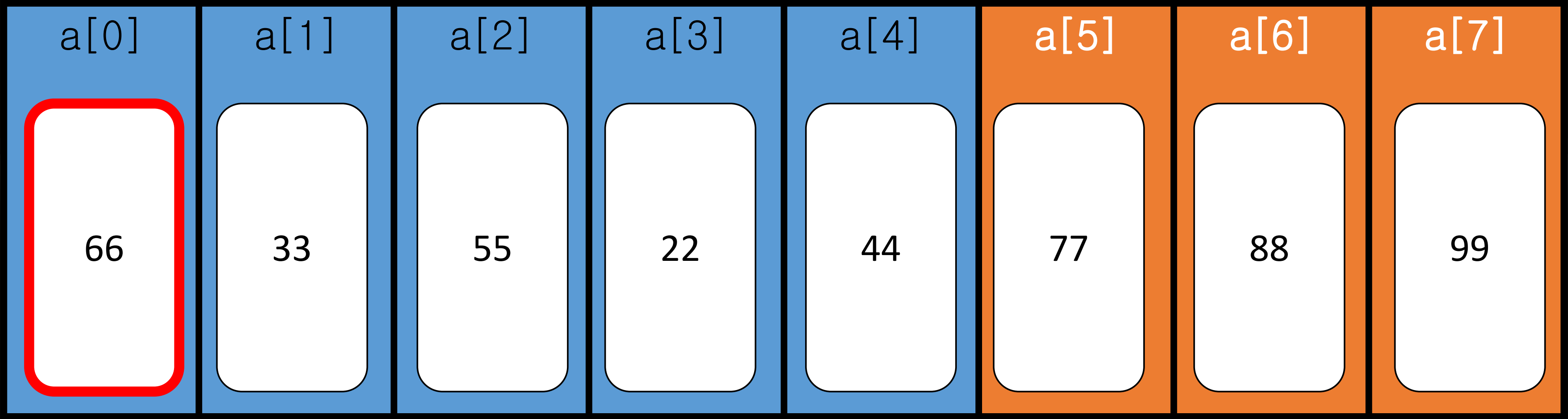




Selection sorting

- Each pass selects the largest among the remaining unsorted elements and moves it into i ts correct position.
- Algorithm
  1. Repeat steps 2 for i=n-1 down to 1
  2. Swap a_i with max{a_0, ⋯, a_i}.

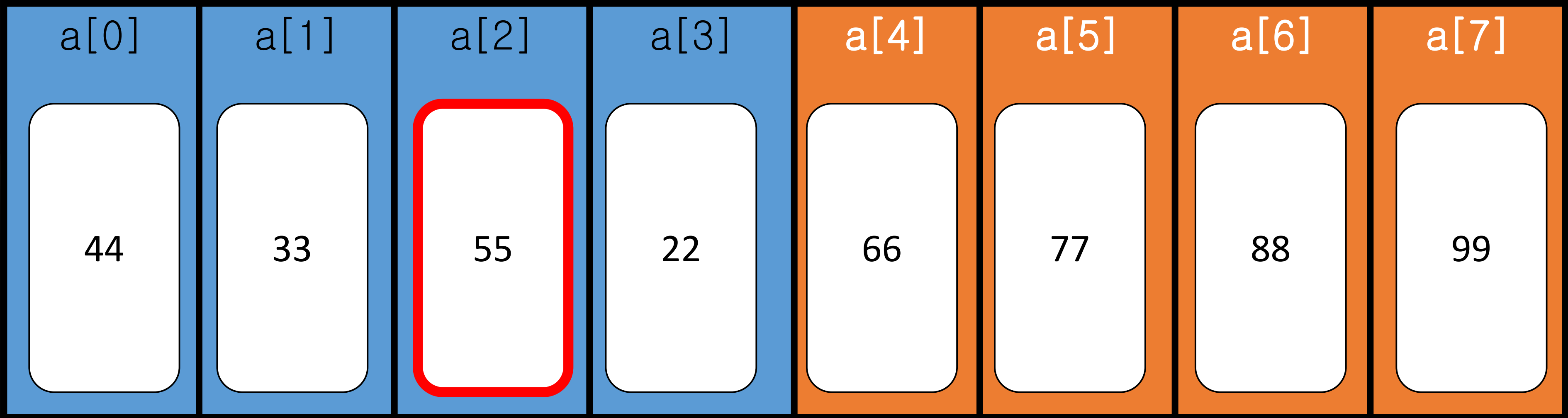
i=4



Selection sorting

- Each pass selects the largest among the remaining unsorted elements and moves it into i ts correct position.
- Algorithm
  1. Repeat steps 2 for i=n-1 down to 1
  2. Swap a_i with max{a_0, ⋯, a_i}.

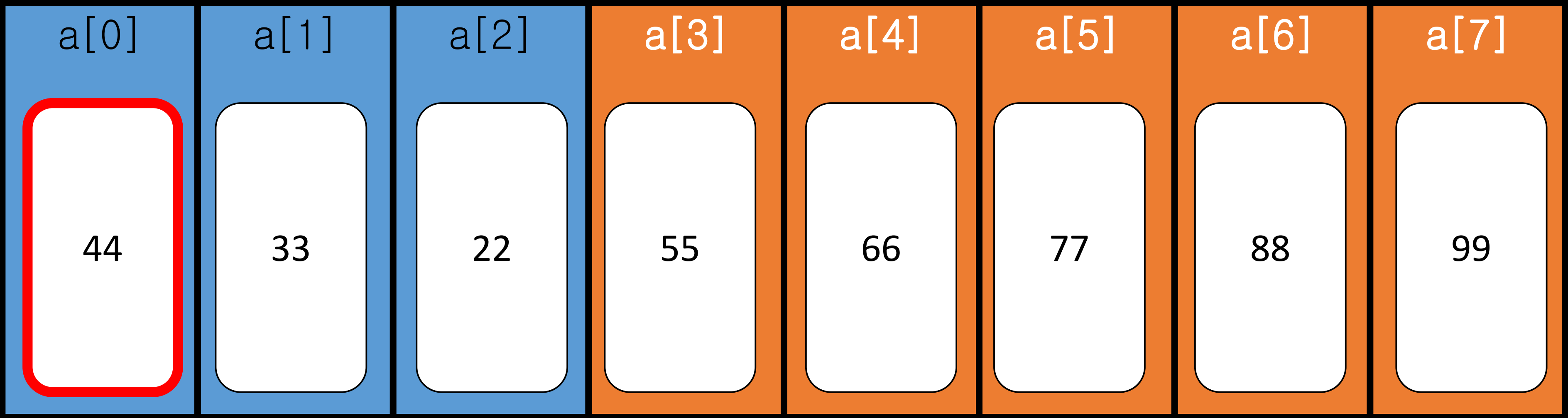
i=3



Selection sorting

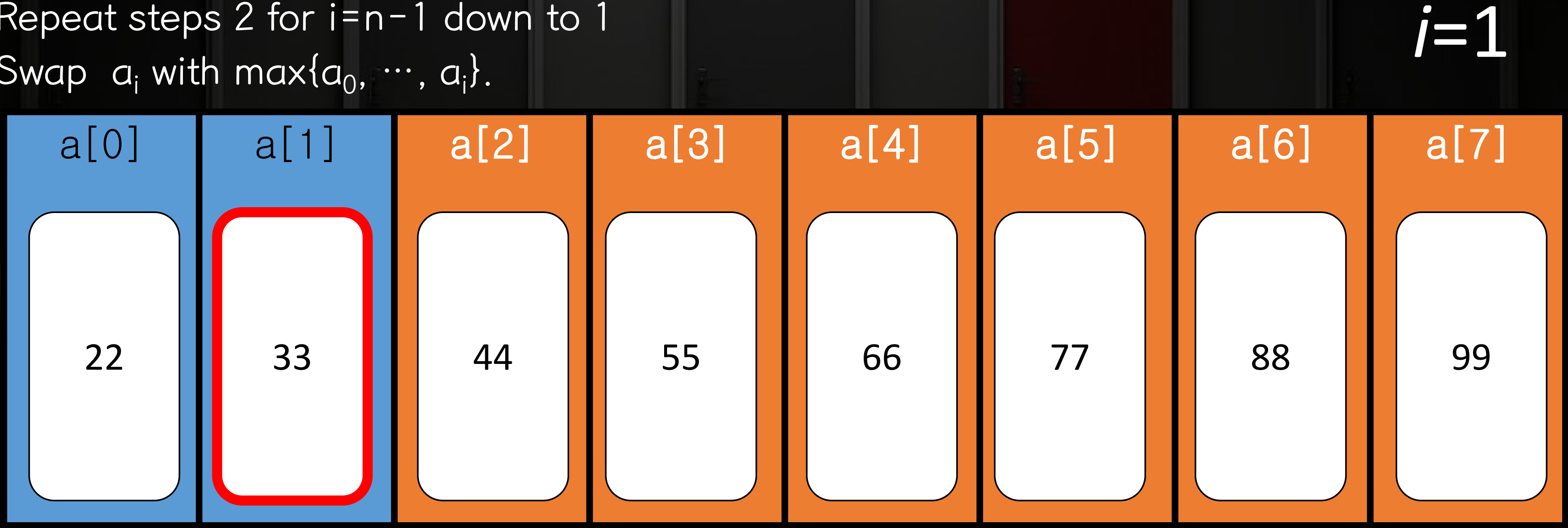
- Each pass selects the largest among the remaining unsorted elements and moves it into i ts correct position.
- Algorithm
  1. Repeat steps 2 for i=n-1 down to 1
  2. Swap a_i with max{a_0, ⋯, a_i}.

i=2



Selection sorting

- Each pass selects the largest among the remaining unsorted elements and moves it into its correct position.
- Algorithm
  1. Repeat steps 2 for i=n-1 down to 1
  2. Swap a_i with $\max\{a_0, \cdots, a_i\}$.



Selection sorting

Iterative selection sort

```
public T[] selectionI() {
    T[] result = array.clone();
    for(int i=result.length-1 ; i > 0 ; i--) {
        swap(result, i, getMaxIndex(result, i));
    }
    return result;
}
```

```
private int getMaxIndex(T[] a, int n) {
    int maxIndex = 0;
    T max=a[0];
    for(int i = 1 ; i <= n ; i++) {
        if(max.compareTo(a[i])<0) {
            max = a[i];
            maxIndex = i;
        }
    }
    return maxIndex;
}
```

Recursive selection sort

```
public T[] selectionR() {
    T[] result = array.clone();
    selectionSort(result, result.length-1);
    return result;

}
private void selectionSort(T[] a, int n) {
    if(n==0) return;
    swap(a, n, getMaxIndex(a, n));
    selectionSort(a, n-1);
}
```

$$\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} = O(n^2)$$

Insertion Sort

Insertion sorting

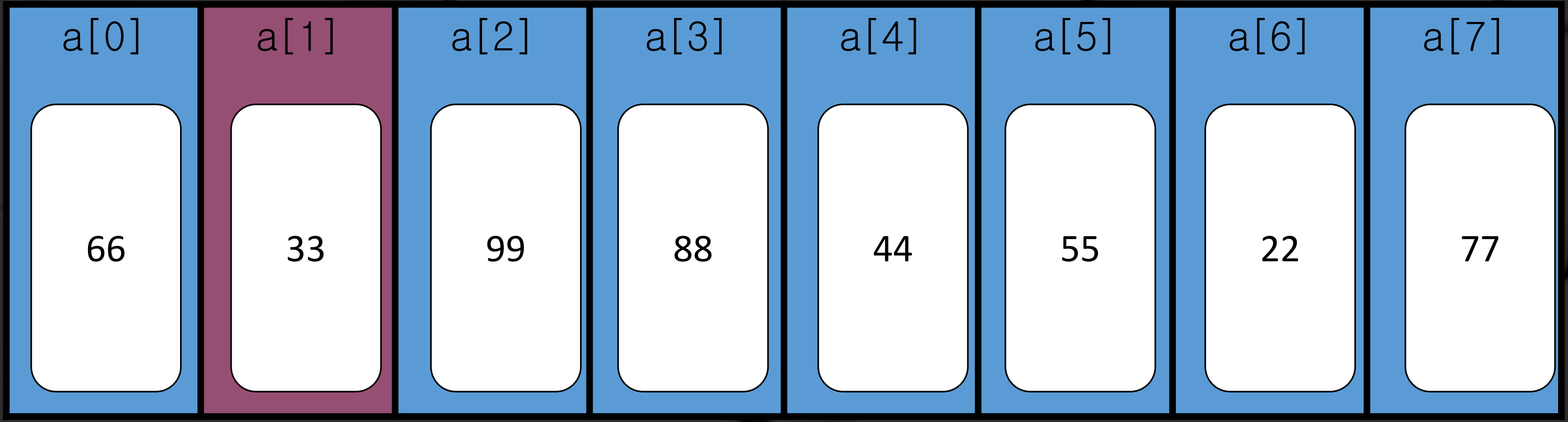
- The idea of insertion sort is to move elements from the unsorted list to the sorted list on e at a time
- As each item is moved, it is inserted into its correct position in the sorted list.
- In order to place the new item, some elements many need to be moved down to create a slot.
- Algorithm

```
1.  Do steps 2-5 for i=1 up to n-1.
    2.  Hold the element $a_i$ is a temporary space.
    3.  Locate the least index $j \le i$ for which $a_j \ge a_i$.
    4.  Shift the subsequence $a_j, \cdots, a_{i-1}$ up one position, into $a_{j+1}, \ldots, a_i$.
    5.  Copy the held value $a_i$ of into $a_j$.
```

Inserting sorting

```
1. Do steps 2-5 for i=1 up to n-1.
2.     Hold the element $a_i$ is a temporary space.
3.     Locate the least index $j \le i$ for which $a_j \ge a_i$.
4.     Shift the subsequence $a_j, \cdots, a_{i-1}$ up one position, into $a_{j+1}, \cdots, a_i$.
5.     Copy the held value $a_i$ of into $a_j$.
```

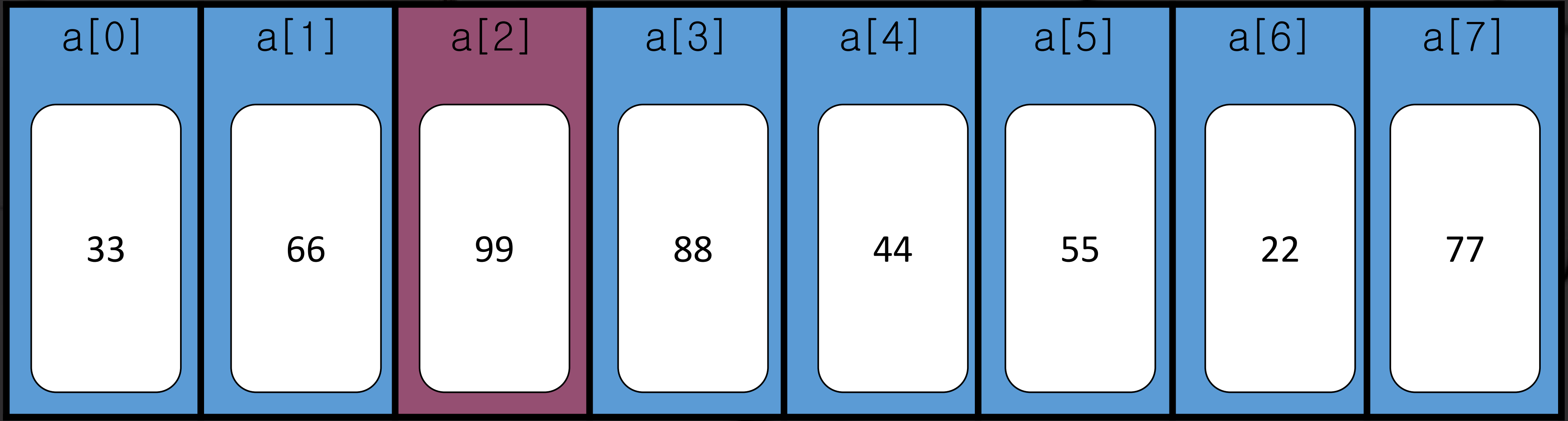
i=1, a[*i*]=33



Inserting sorting

1. Do steps 2-5 for i=1 up to n-1.
2. Hold the element a_i is a temporary space.
3. Locate the least index $j \leq i$ for which $a_j \geq a_i$.
4. Shift the subsequence $a_j, \cdots, a_{i-1}$ up one position, into $a_{j+1}, \cdots, a_i$.
5. Copy the held value a_i of into a_j.

i=2, a[*i*]=99

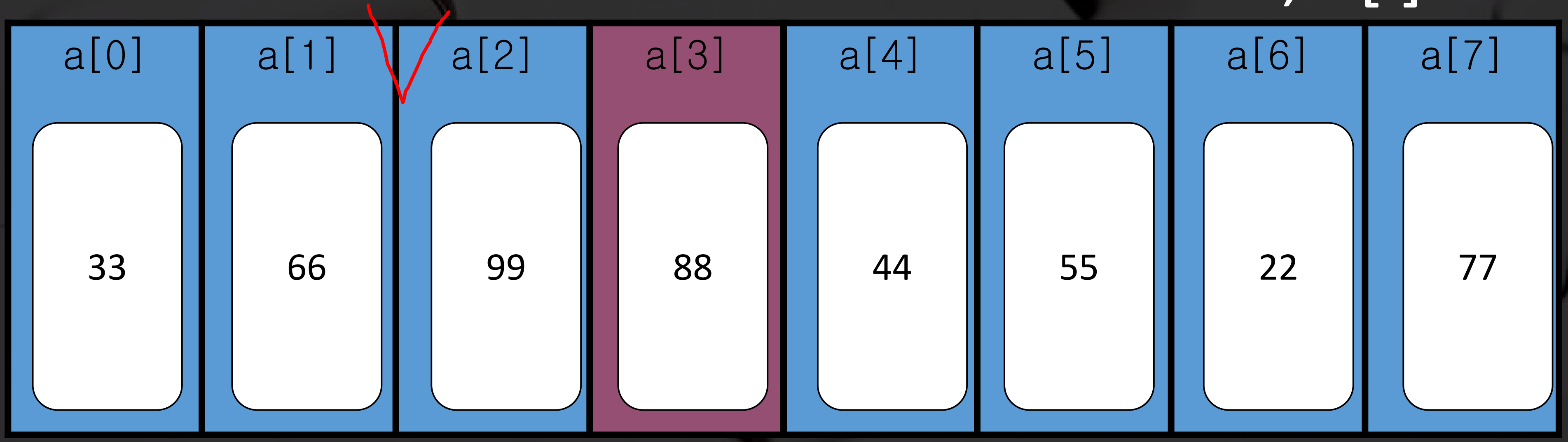


a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
33	66	99	88	44	55	22	77

Inserting sorting

1. Do steps 2-5 for i=1 up to n-1.
2. Hold the element a_i is a temporary space.
3. Locate the least index $j \leq i$ for which $a_j \geq a_i$.
4. Shift the subsequence $a_j, \cdots, a_{i-1}$ up one position, into $a_{j+1}, \cdots, a_i$.
5. Copy the held value a_i of into a_j.

i=3, a[i]=88



Inserting sorting

```
1. Do steps 2-5 for i=1 up to n-1.
2.     Hold the element $a_i$ is a temporary space.
3.     Locate the least index $j \le i$ for which $a_j \ge a_i$.
4.     Shift the subsequence $a_j, \cdots, a_{i-1}$ up one position, into $a_{j+1}, \cdots, a_i$.
5.     Copy the held value $a_i$ of into $a_j$.
```

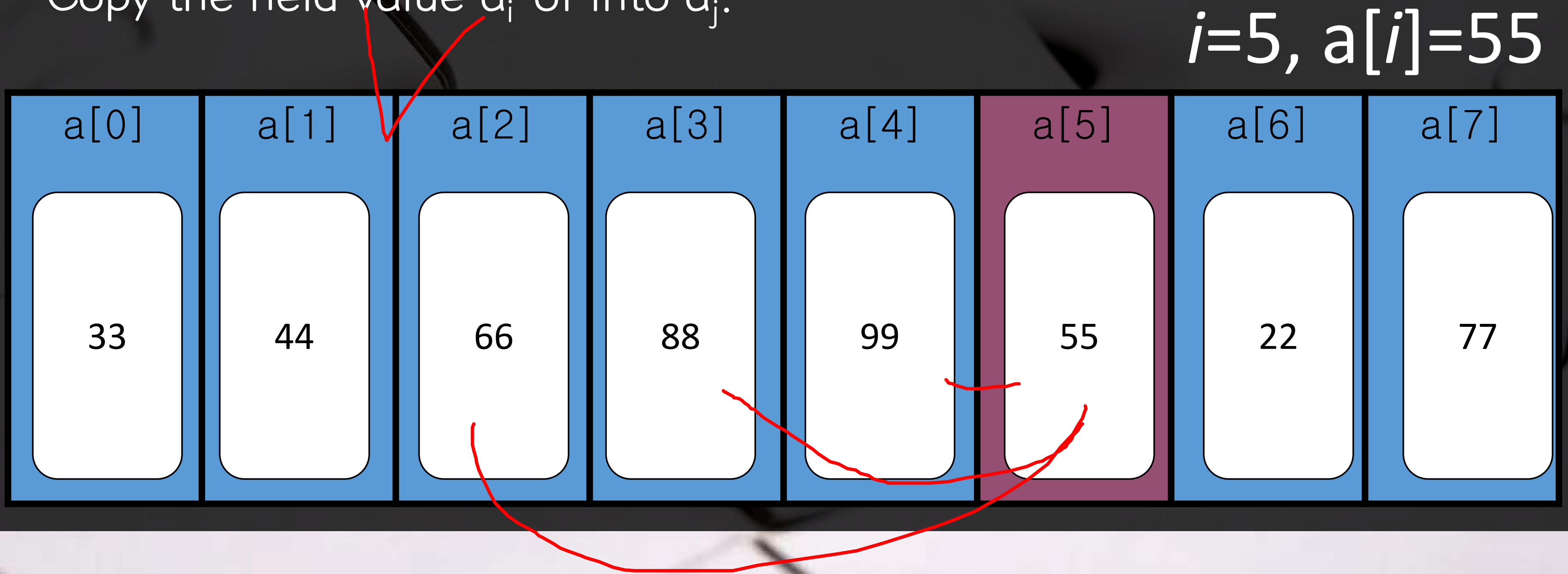
i=4, a[*i*]=44

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
33	66	88	99	44	55	22	77

Inserting sorting

1. Do steps 2-5 for i=1 up to n-1.
2. Hold the element a_i is a temporary space.
3. Locate the least index $j \leq i$ for which $a_j \geq a_i$.
4. Shift the subsequence $a_j, \cdots, a_{i-1}$ up one position, into $a_{j+1}, \cdots, a_i$.
5. Copy the held value a_i of into a_j.

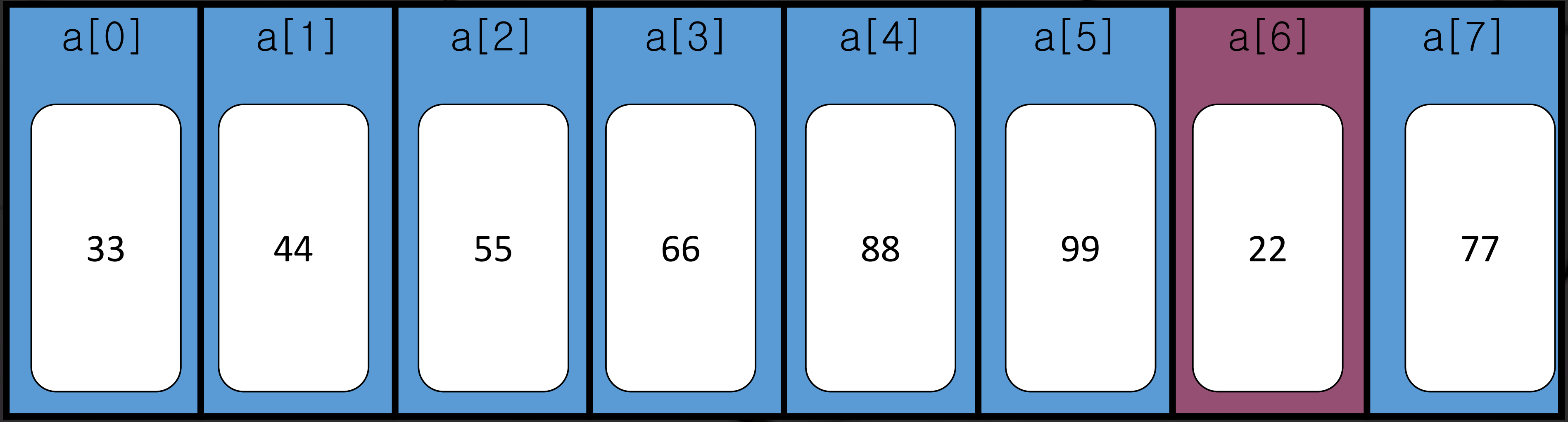
i=5, a[i]=55



Inserting sorting

```
1. Do steps 2-5 for i=1 up to n-1.
2.     Hold the element $a_i$ is a temporary space.
3.     Locate the least index $j \leq i$ for which $a_j \geq a_i$.
4.     Shift the subsequence $a_j, \cdots, a_{i-1}$ up one position, into $a_{j+1}, \cdots, a_i$.
5.     Copy the held value $a_i$ of into $a_j$.
```

i=6, a[*i*]=22



Inserting sorting

1. Do steps 2-5 for i=1 up to n-1.
2. Hold the element a_i is a temporary space.
3. Locate the least index $j \leq i$ for which $a_j \geq a_i$.
4. Shift the subsequence $a_j, \cdots, a_{i-1}$ up one position, into $a_{j+1}, \cdots, a_i$.
5. Copy the held value a_i of into a_j.

i=7, a[i]=77

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
22	33	44	55	66	88	99	77

Insertion sorting

Iterative sort

```
public T[] insertionI() {
    T[] result = array.clone();
    for(int i=1 ; i < result.length ; i++) {
        for(int j = i-1 ; j >= 0 && result[j+1].compareTo(result[j])<=0; j--){
            if(result[j+1].compareTo(result[j])<0) {
                swap(result, j, j+1);
            }
        }
    }
    return result;
}
```

Recursive sort

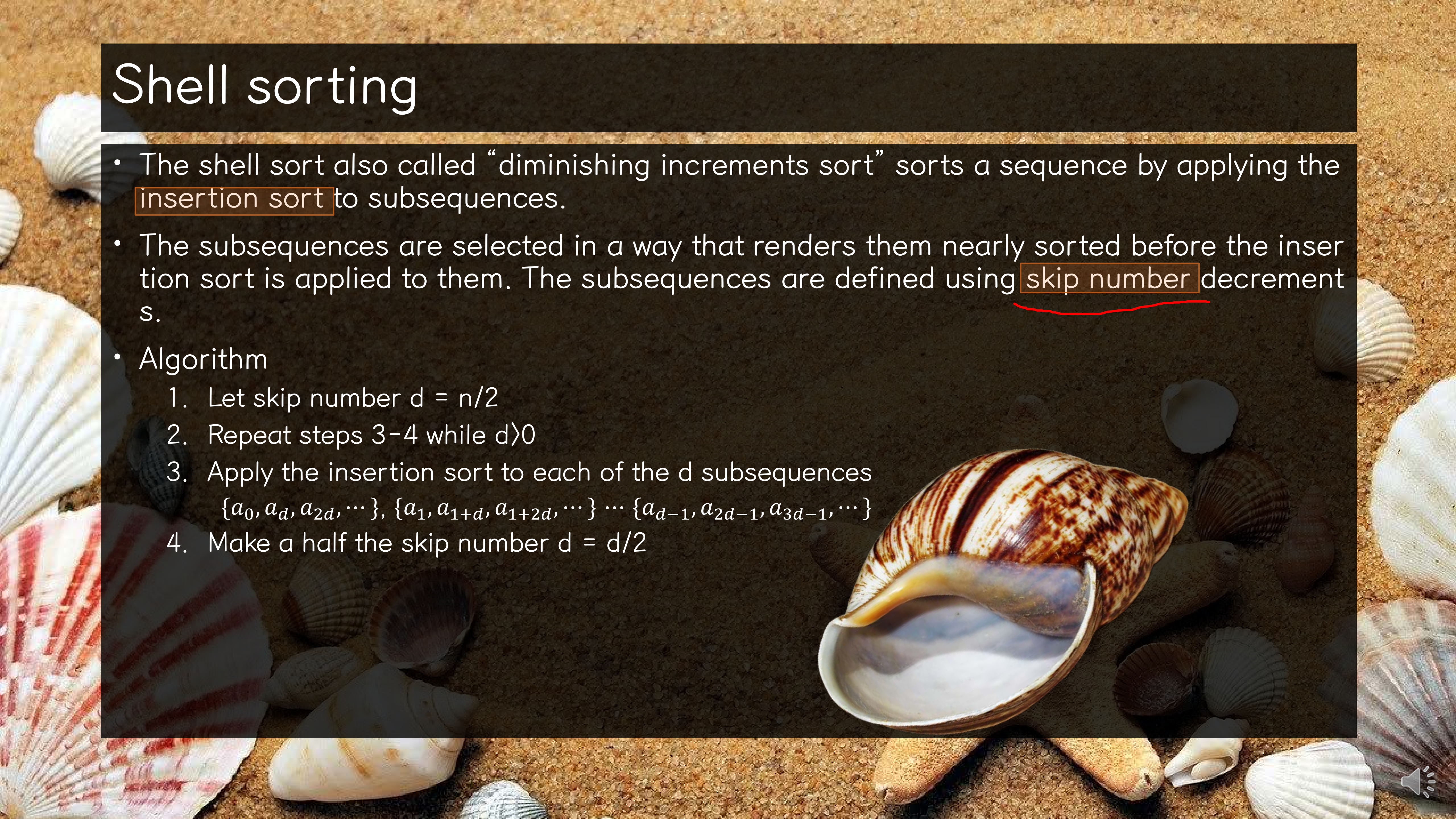
```
public T[] insertionR() {
    T[] result = array.clone();
    for(int i=1 ; i < result.length ; i++) {
        if(result[i].compareTo(result[i-1])<0)
            insertionSort(result, result[i], i-1);
    }
    return result;

}
private void insertionSort(T[] a, T value, int n) {
    if(n>=0 && value.compareTo(a[n])<0) {
        swap(a, n, n+1);
        insertionSort(a, value, n-1);
    } else  return;
}
```

- Formula:
  - Best case: $O(n)$
  - Worst case: $1+2+\cdots+(n-1) = n(n-1)/2 \sim O(n^2)$
  - Average case: $n(n-1)/4 \sim O(n^2)$

Shell Sort

Shell sorting

- The shell sort also called "diminishing increments sort" sorts a sequence by applying the insertion sort to subsequences.
- The subsequences are selected in a way that renders them nearly sorted before the inser tion sort is applied to them. The subsequences are defined using skip number decrement s.
- Algorithm
    1. Let skip number d = n/2
    2. Repeat steps 3-4 while d>0
    3. Apply the insertion sort to each of the d subsequences
       $\{a_0, a_d, a_{2d}, \cdots\}, \{a_1, a_{1+d}, a_{1+2d}, \cdots\} \cdots \{a_{d-1}, a_{2d-1}, a_{3d-1}, \cdots\}$
    4. Make a half the skip number d = d/2

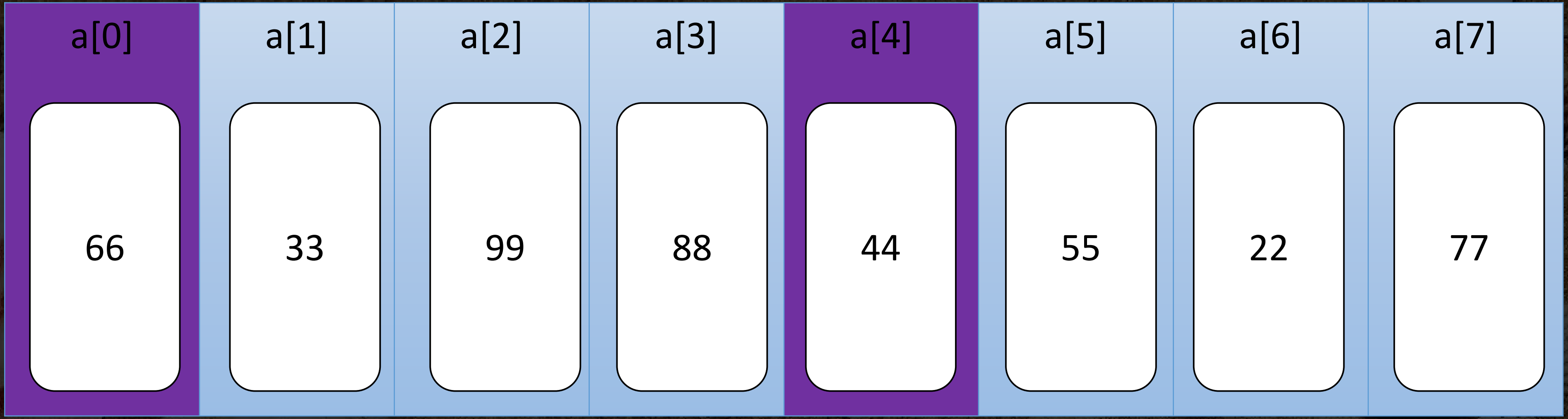


Shell sorting

1. Let skip number d = n/2
2. Repeat steps 3-4 while d>0
3. Apply the insertion sort to each of the d subsequences
   $\{a_0, a_d, a_{2d}, \cdots\}, \{a_1, a_{1+d}, a_{1+2d}, \cdots\} \cdots \{a_{d-1}, a_{2d-1}, a_{3d-1}, \cdots\}$
4. Make a half the skip number d = d/2

d=8/2=4

a*d+0



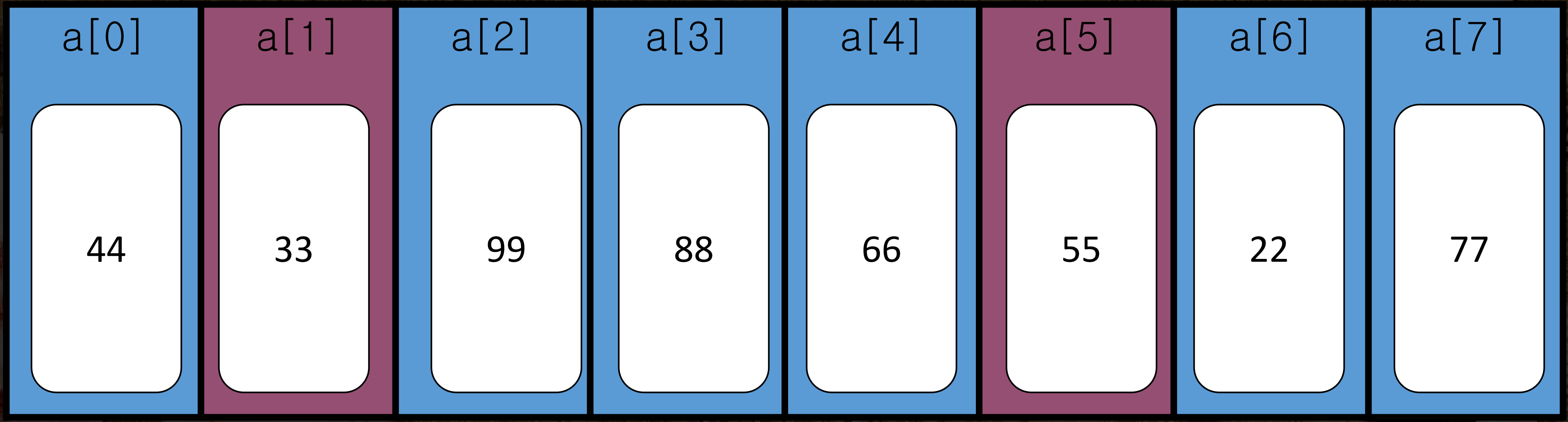
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
66	33	99	88	44	55	22	77

Shell sorting

1. Let skip number d = n/2
2. Repeat steps 3-4 while d>0
3. Apply the insertion sort to each of the d subsequences
   $\{a_0, a_d, a_{2d}, \cdots\}, \{a_1, a_{1+d}, a_{1+2d}, \cdots\} \cdots \{a_{d-1}, a_{2d-1}, a_{3d-1}, \cdots\}$
4. Make a half the skip number d = d/2

d=8/2=4

a*d+1



Shell sorting

1. Let skip number d = n/2
2. Repeat steps 3-4 while d>0
3. Apply the insertion sort to each of the d subsequences
   $\{a_0, a_d, a_{2d}, \cdots\}, \{a_1, a_{1+d}, a_{1+2d}, \cdots\} \cdots \{a_{d-1}, a_{2d-1}, a_{3d-1}, \cdots\}$
4. Make a half the skip number d = d/2

d=8/2=4

a*d+2

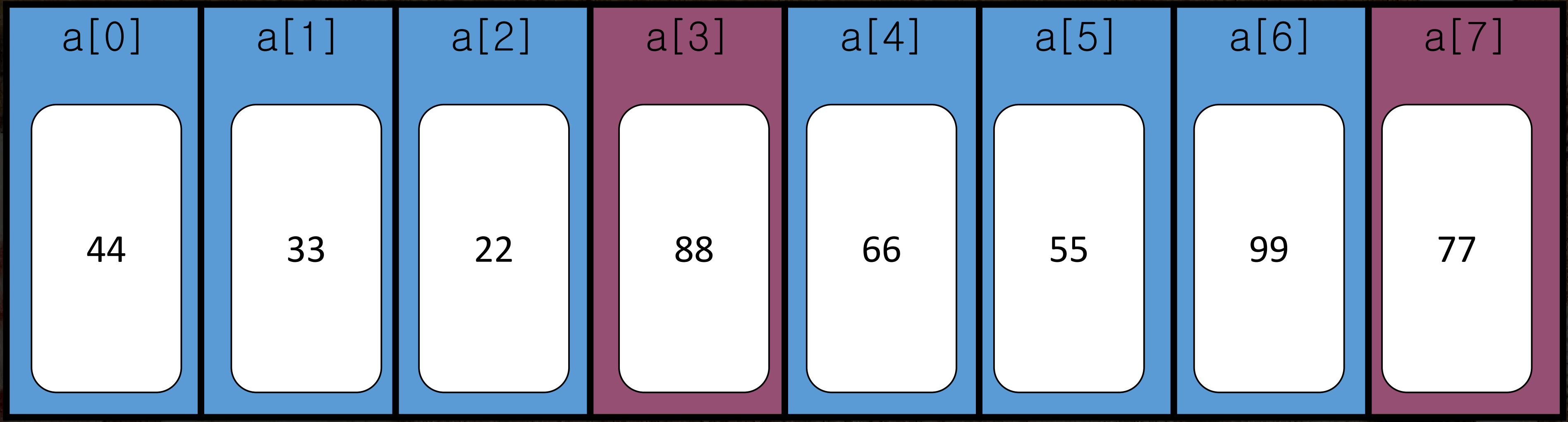
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
44	33	99	88	66	55	22	77

Shell sorting

1. Let skip number d = n/2
2. Repeat steps 3-4 while d>0
3. Apply the insertion sort to each of the d subsequences
   $\{a_0, a_d, a_{2d}, \cdots\}, \{a_1, a_{1+d}, a_{1+2d}, \cdots\} \cdots \{a_{d-1}, a_{2d-1}, a_{3d-1}, \cdots\}$
4. Make a half the skip number d = d/2

d=8/2=4

a*d+3



Shell sorting

1. Let skip number d = n/2
2. Repeat steps 3-4 while d>0
3. Apply the insertion sort to each of the d subsequences
   $\{a_0, a_d, a_{2d}, \cdots\}, \{a_1, a_{1+d}, a_{1+2d}, \cdots\} \cdots \{a_{d-1}, a_{2d-1}, a_{3d-1}, \cdots\}$
4. Make a half the skip number d = d/2

$d=4/2=\boxed{2}$

a*d+0

i=2, a[i]=22

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
44	33	22	77	66	55	99	88

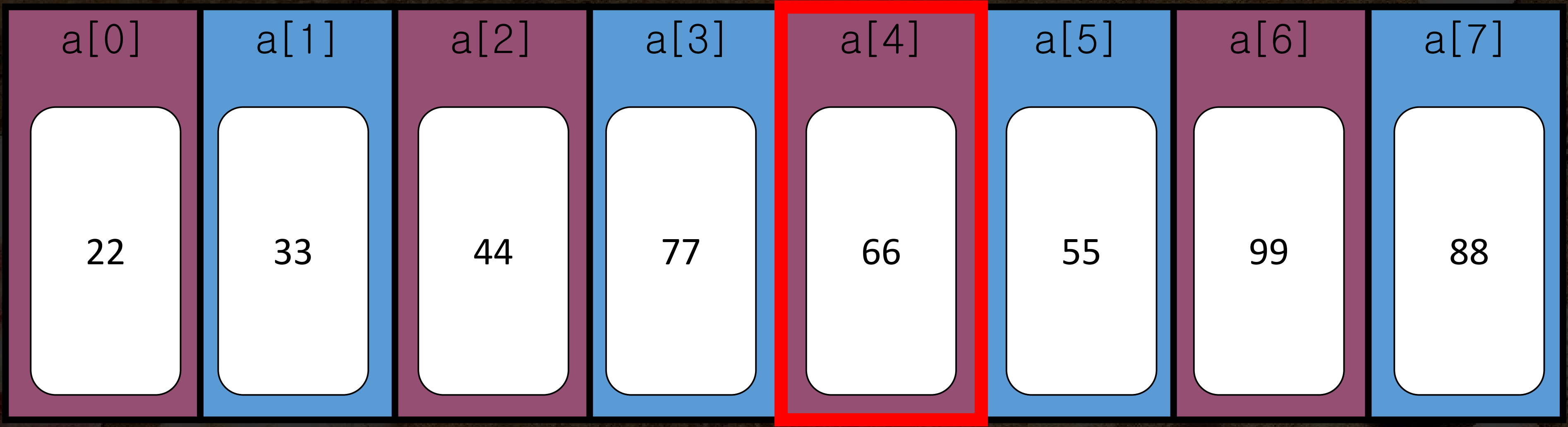
Shell sorting

1. Let skip number d = n/2
2. Repeat steps 3-4 while d>0
3. Apply the insertion sort to each of the d subsequences
   $\{a_0, a_d, a_{2d}, \cdots\}, \{a_1, a_{1+d}, a_{1+2d}, \cdots\} \cdots \{a_{d-1}, a_{2d-1}, a_{3d-1}, \cdots\}$
4. Make a half the skip number d = d/2

d=4/2=2

a*d+0

i=4, a[i]=66



a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
22	33	44	77	66	55	99	88

Shell sorting

1. Let skip number d = n/2
2. Repeat steps 3-4 while d>0
3. Apply the insertion sort to each of the d subsequences
   $\{a_0, a_d, a_{2d}, \cdots\}, \{a_1, a_{1+d}, a_{1+2d}, \cdots\} \cdots \{a_{d-1}, a_{2d-1}, a_{3d-1}, \cdots\}$
4. Make a half the skip number d = d/2

d=4/2=2

a*d+0

i=6, a[i]=99

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
22	33	44	77	66	55	99	88

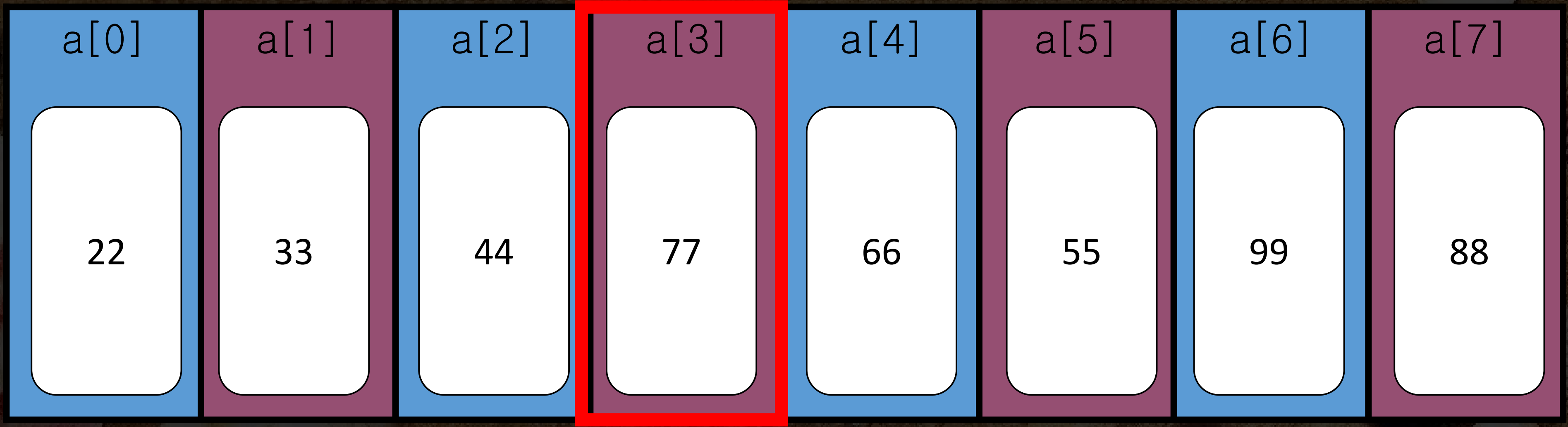
Shell sorting

1. Let skip number d = n/2
2. Repeat steps 3-4 while d>0
3. Apply the insertion sort to each of the d subsequences
   $\{a_0, a_d, a_{2d}, \cdots\}, \{a_1, a_{1+d}, a_{1+2d}, \cdots\} \cdots \{a_{d-1}, a_{2d-1}, a_{3d-1}, \cdots\}$
4. Make a half the skip number d = d/2

d=4/2=2

a*d+1

i=3, a[i]=77



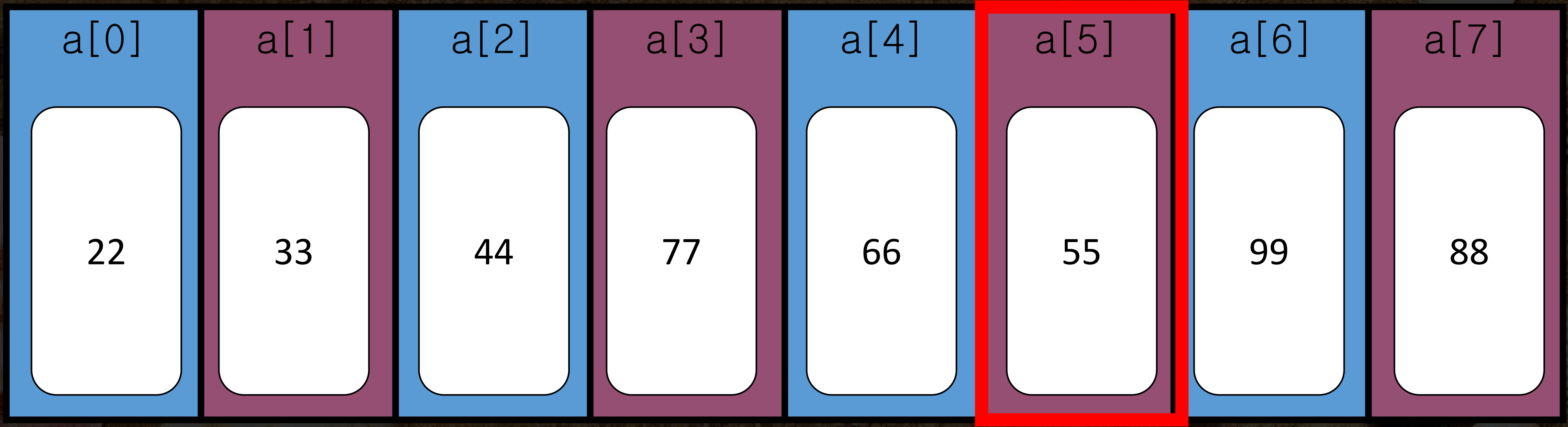
Shell sorting

1. Let skip number d = n/2
2. Repeat steps 3-4 while d>0
3. Apply the insertion sort to each of the d subsequences
   $\{a_0, a_d, a_{2d}, \cdots\}, \{a_1, a_{1+d}, a_{1+2d}, \cdots\} \cdots \{a_{d-1}, a_{2d-1}, a_{3d-1}, \cdots\}$
4. Make a half the skip number d = d/2

d=4/2=2

a*d+1

i=5, a[i]=55



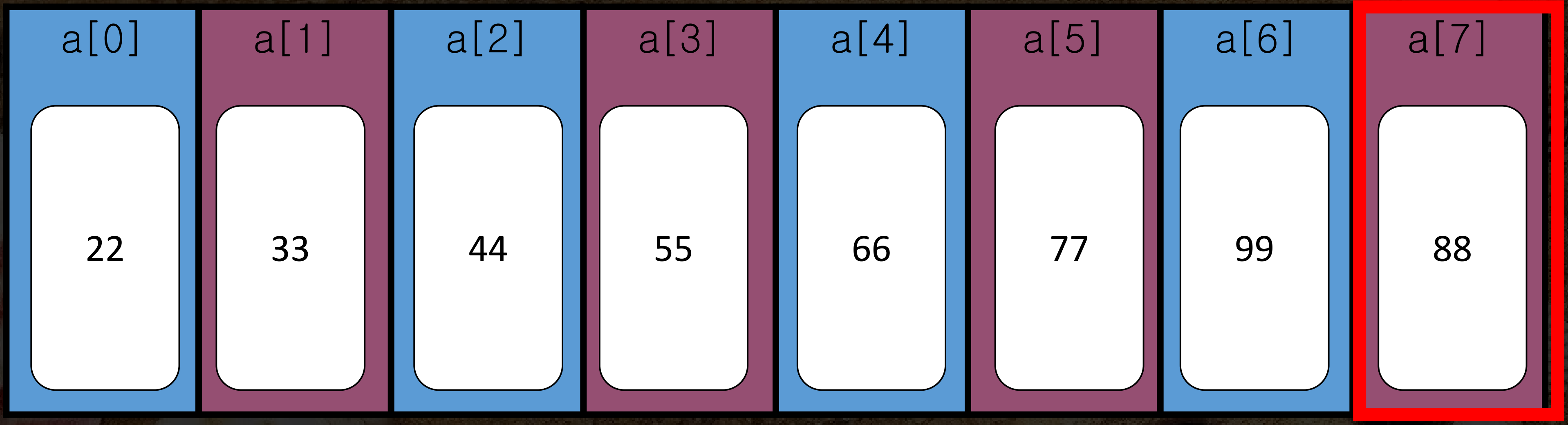
Shell sorting

1. Let skip number d = n/2
2. Repeat steps 3-4 while d>0
3. Apply the insertion sort to each of the d subsequences
   $\{a_0, a_d, a_{2d}, \cdots\}, \{a_1, a_{1+d}, a_{1+2d}, \cdots\} \cdots \{a_{d-1}, a_{2d-1}, a_{3d-1}, \cdots\}$
4. Make a half the skip number d = d/2

d=4/2=2

a*d+1

i=7, a[i]=88



Shell sorting

1. Let skip number d = n/2
2. Repeat steps 3-4 while d>0
3. Apply the insertion sort to each of the d subsequences
   $\{a_0, a_d, a_{2d}, \cdots\}, \{a_1, a_{1+d}, a_{1+2d}, \cdots\} \cdots \{a_{d-1}, a_{2d-1}, a_{3d-1}, \cdots\}$
4. Make a half the skip number d = d/2

d=2/2=1

a*d

i=1, a[i]=33

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
22	33	44	55	66	77	99	88

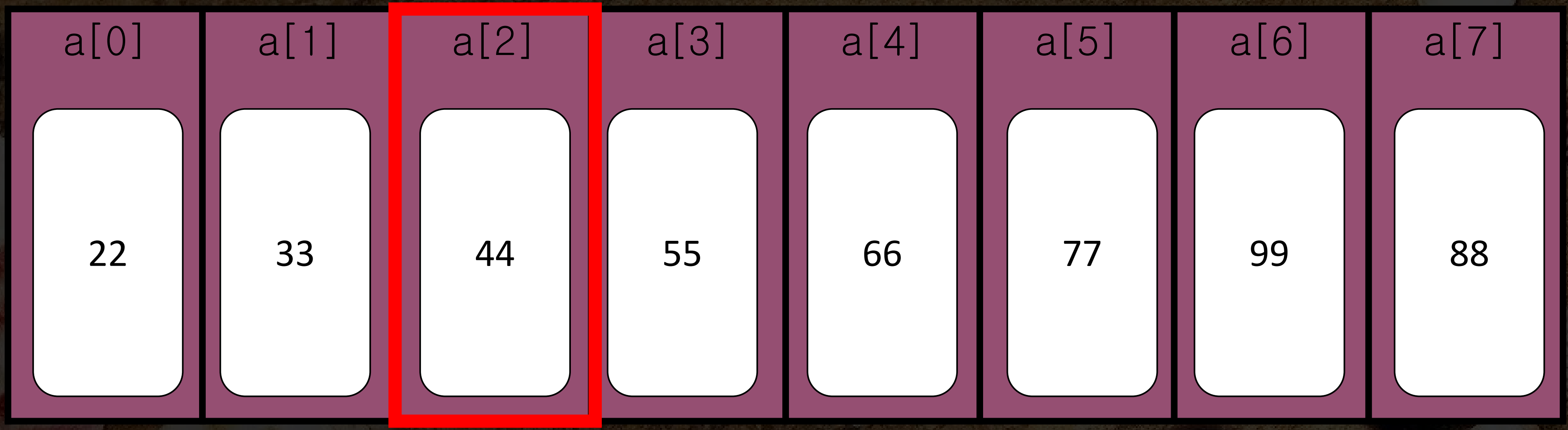
Shell sorting

1. Let skip number d = n/2
2. Repeat steps 3-4 while d>0
3. Apply the insertion sort to each of the d subsequences
   $\{a_0, a_d, a_{2d}, \cdots\}, \{a_1, a_{1+d}, a_{1+2d}, \cdots\} \cdots \{a_{d-1}, a_{2d-1}, a_{3d-1}, \cdots\}$
4. Make a half the skip number d = d/2

d=2/2=1

a*d

i=2, a[i]=22



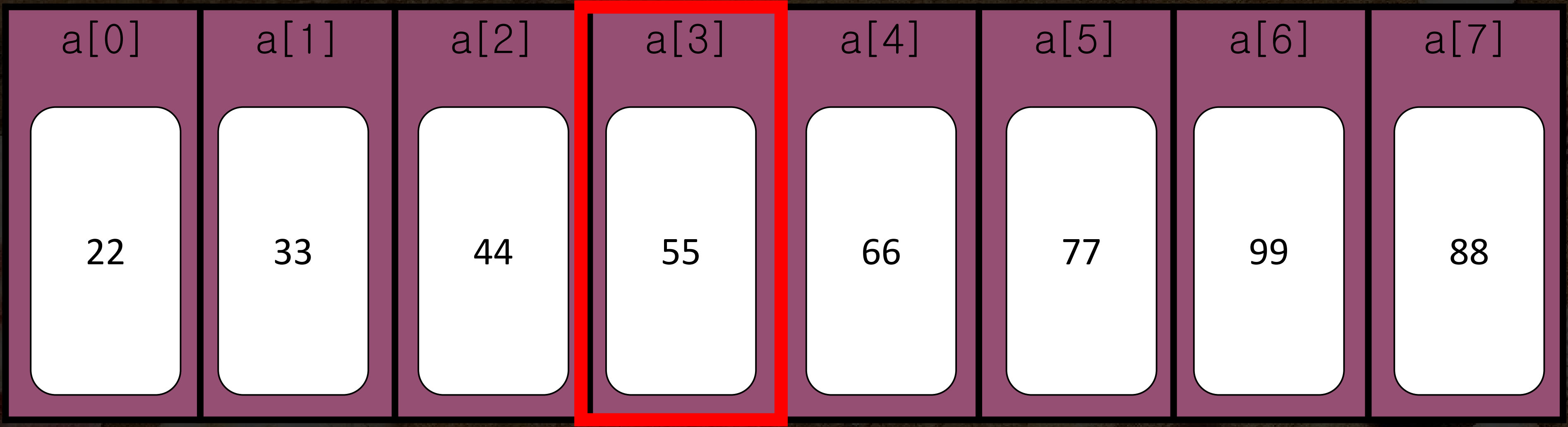
Shell sorting

1. Let skip number d = n/2
2. Repeat steps 3-4 while d>0
3. Apply the insertion sort to each of the d subsequences
   $\{a_0, a_d, a_{2d}, \cdots\}, \{a_1, a_{1+d}, a_{1+2d}, \cdots\} \cdots \{a_{d-1}, a_{2d-1}, a_{3d-1}, \cdots\}$
4. Make a half the skip number d = d/2

d=2/2=1

a*d

i=3, a[i]=55



Shell sorting

1. Let skip number d = n/2
2. Repeat steps 3-4 while d>0
3. Apply the insertion sort to each of the d subsequences
   $\{a_0, a_d, a_{2d}, \cdots\}, \{a_1, a_{1+d}, a_{1+2d}, \cdots\} \cdots \{a_{d-1}, a_{2d-1}, a_{3d-1}, \cdots\}$
4. Make a half the skip number d = d/2

d=2/2=1

a*d

i=4, a[i]=66

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
22	33	44	55	66	77	99	88

Shell sorting

1. Let skip number d = n/2
2. Repeat steps 3-4 while d>0
3. Apply the insertion sort to each of the d subsequences
   $\{a_0, a_d, a_{2d}, \cdots\}, \{a_1, a_{1+d}, a_{1+2d}, \cdots\} \cdots \{a_{d-1}, a_{2d-1}, a_{3d-1}, \cdots\}$
4. Make a half the skip number d = d/2

d=2/2=1

a*d

i=5, a[i]=77

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
22	33	44	55	66	77	99	88

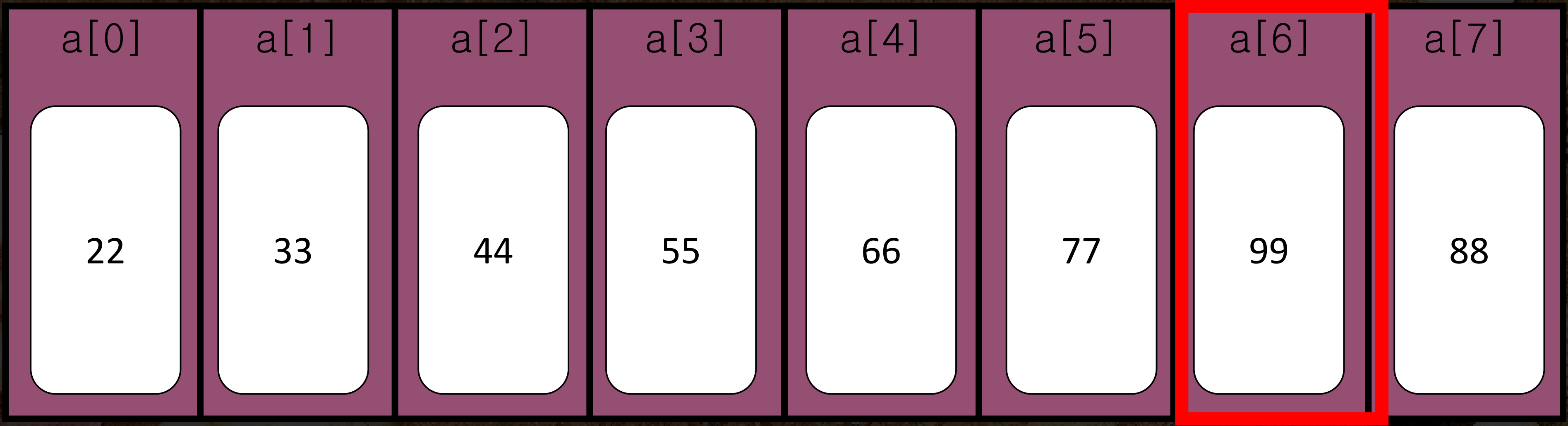
Shell sorting

1. Let skip number d = n/2
2. Repeat steps 3-4 while d>0
3. Apply the insertion sort to each of the d subsequences
   $\{a_0, a_d, a_{2d}, \cdots\}, \{a_1, a_{1+d}, a_{1+2d}, \cdots\} \cdots \{a_{d-1}, a_{2d-1}, a_{3d-1}, \cdots\}$
4. Make a half the skip number d = d/2

d=2/2=1

a*d

i=6, a[i]=99



a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
22	33	44	55	66	77	99	88

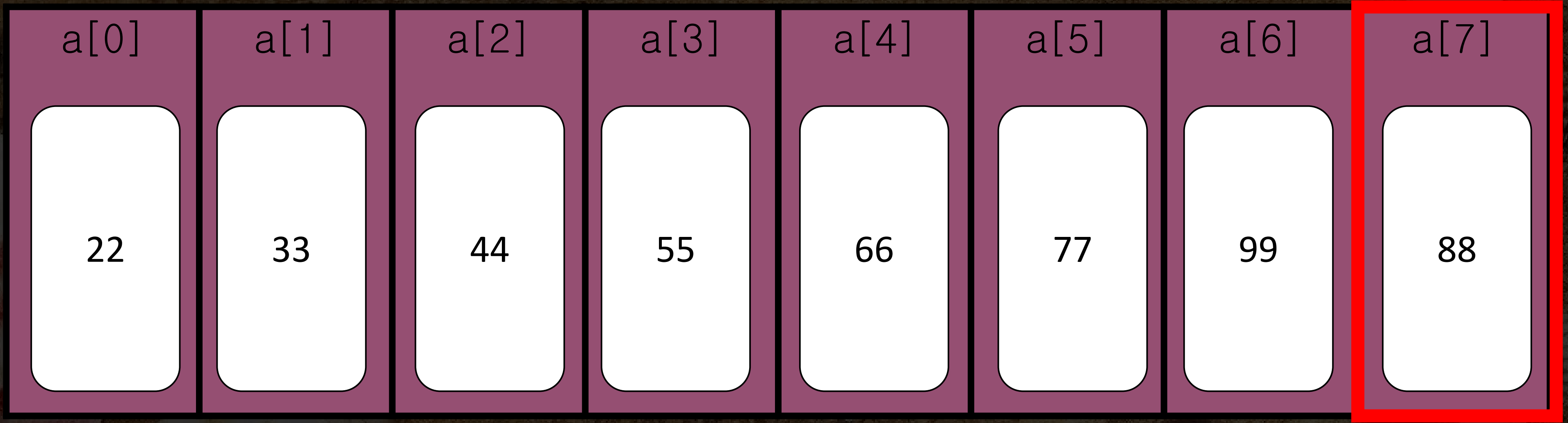
Shell sorting

1. Let skip number d = n/2
2. Repeat steps 3-4 while d>0
3. Apply the insertion sort to each of the d subsequences
   $\{a_0, a_d, a_{2d}, \cdots\}, \{a_1, a_{1+d}, a_{1+2d}, \cdots\} \cdots \{a_{d-1}, a_{2d-1}, a_{3d-1}, \cdots\}$
4. Make a half the skip number d = d/2

d=2/2=1

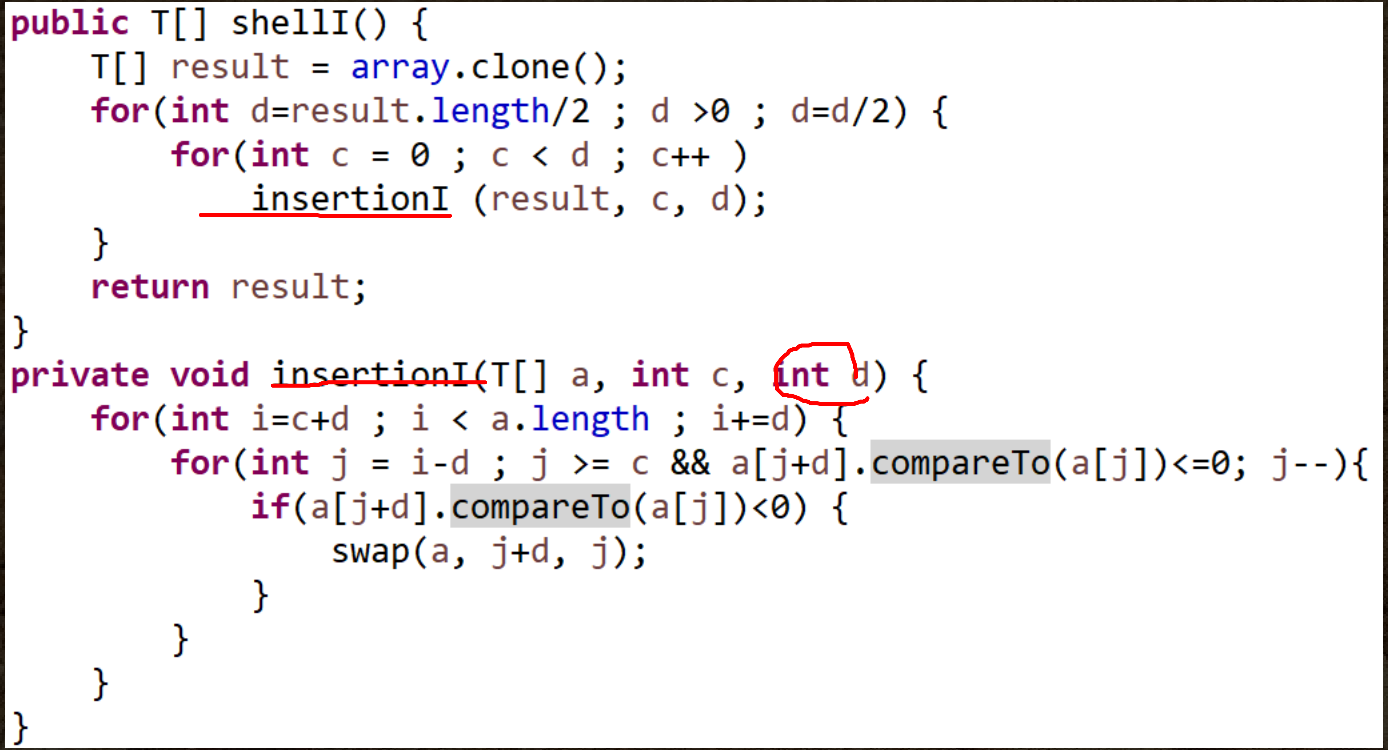
a*d

i=7, a[i]=88



a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
22	33	44	55	66	77	99	88

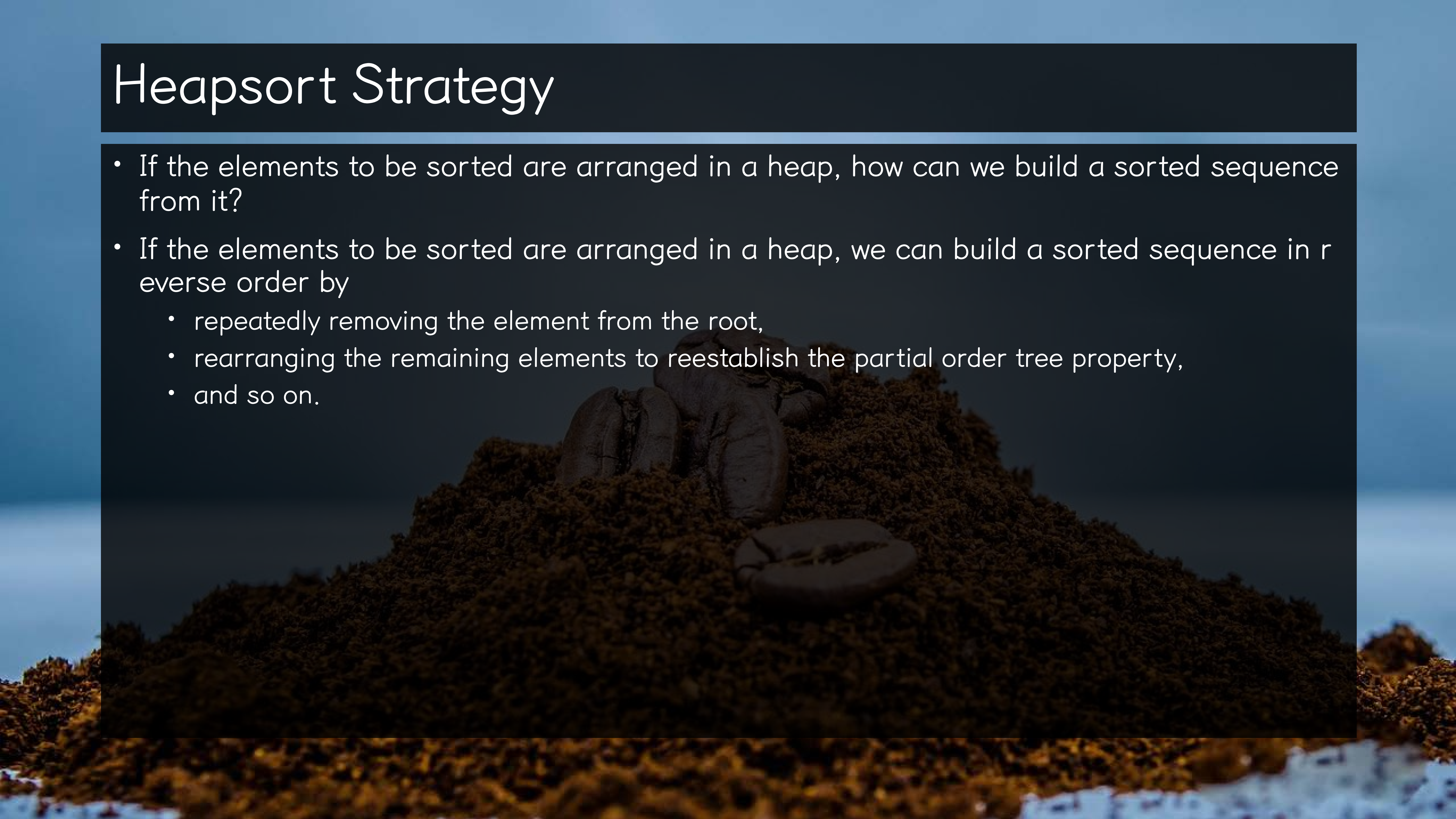
Shell sorting



```
public T[] shellI() {
    T[] result = array.clone();
    for(int d=result.length/2 ; d >0 ; d=d/2) {
        for(int c = 0 ; c < d ; c++ )
            insertionI (result, c, d);
    }
    return result;
}
private void insertionI(T[] a, int c, int d) {
    for(int i=c+d ; i < a.length ; i+=d) {
        for(int j = i-d ; j >= c && a[j+d].compareTo(a[j])<=0; j--){
            if(a[j+d].compareTo(a[j])<0) {
                swap(a, j+d, j);
            }
        }
    }
}
```

- Formula:
    - Average $\Theta(n^{1.5})$
    - Worst case: $\Theta(n^2)$

Heap Sort



Heapsort Strategy

- If the elements to be sorted are arranged in a heap, how can we build a sorted sequence from it?
- If the elements to be sorted are arranged in a heap, we can build a sorted sequence in r everse order by
    - repeatedly removing the element from the root,
    - rearranging the remaining elements to reestablish the partial order tree property,
    - and so on.

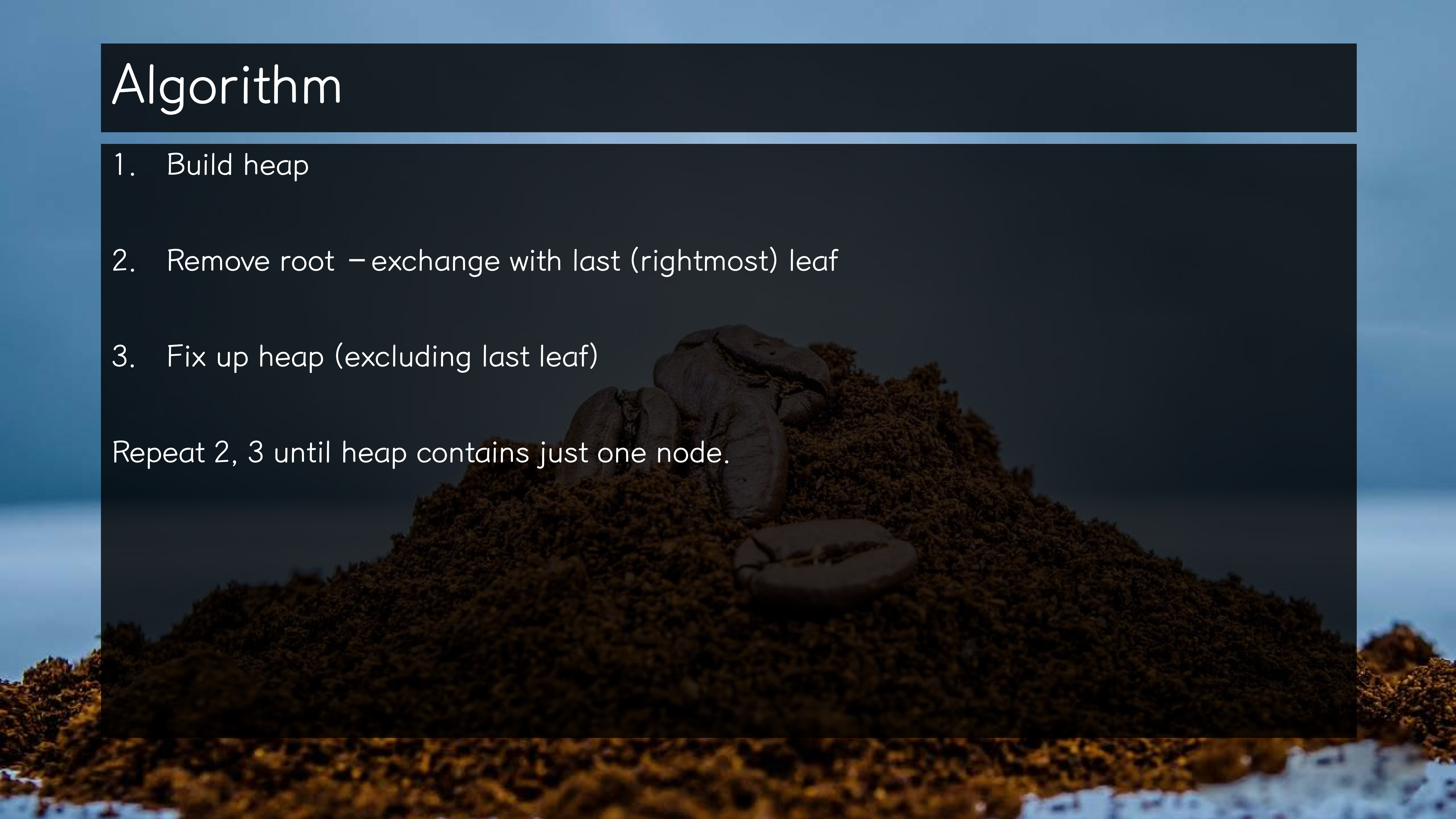
Algorithm

1. Build heap

2. Remove root – exchange with last (rightmost) leaf

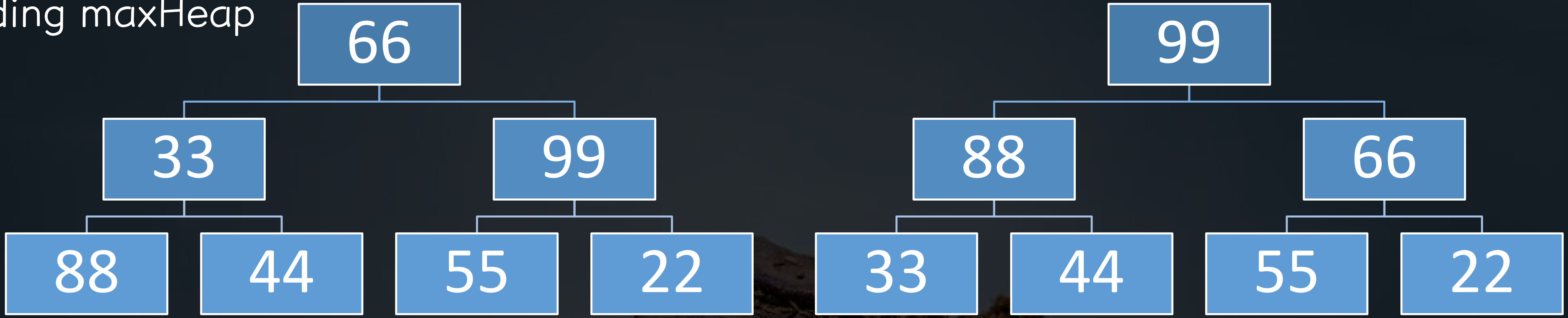
3. Fix up heap (excluding last leaf)

Repeat 2, 3 until heap contains just one node.



Building Heap

- Building maxHeap

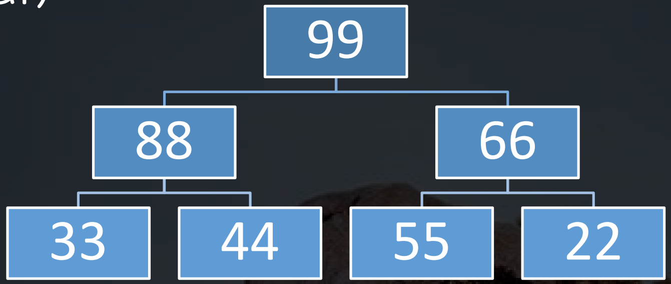


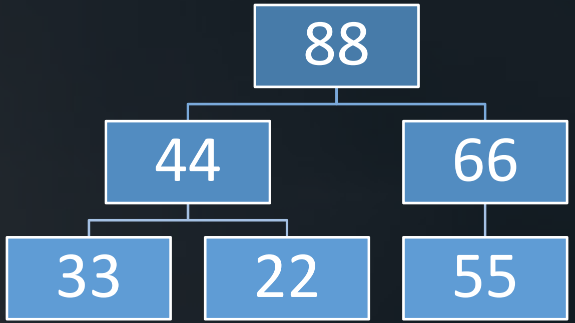
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
66	33	99	88	44	55	22

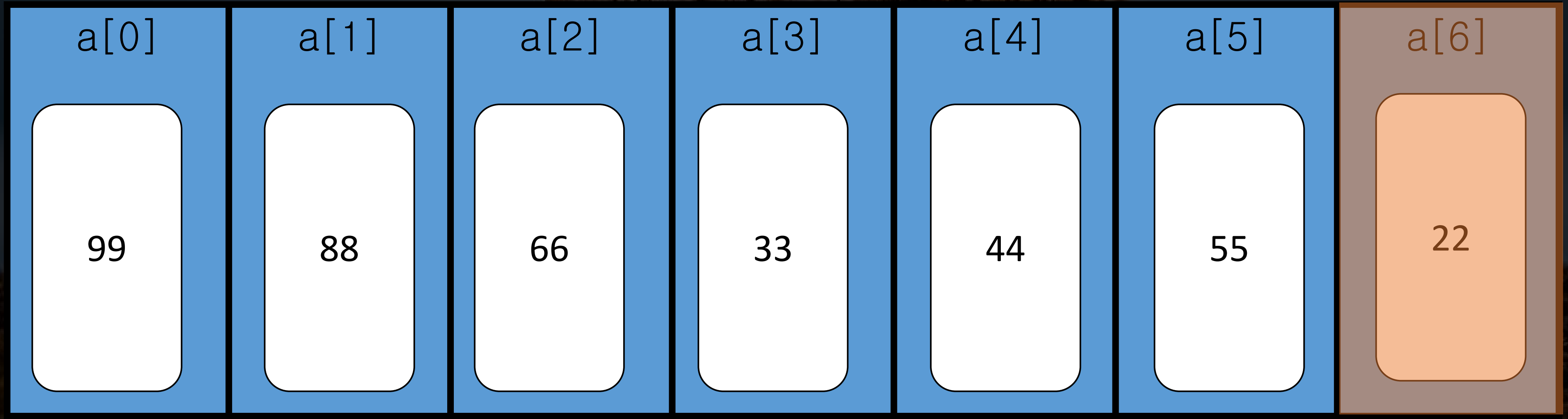
Get the Highest Priority

Remove root – exchange with last (rightmost) leaf

Fix up heap (excluding last leaf)



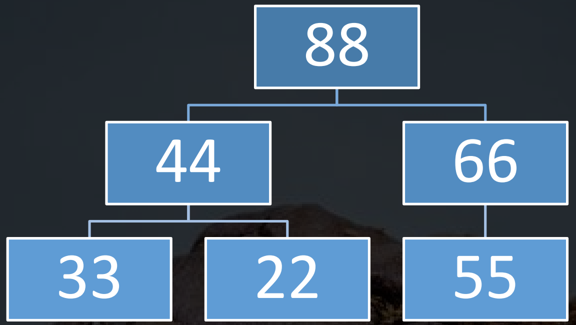


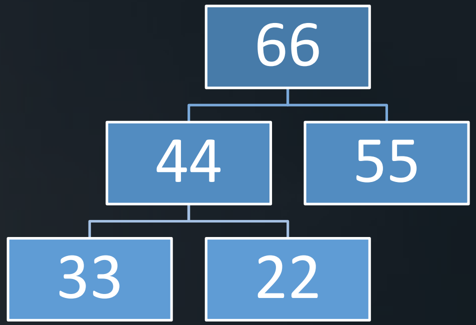


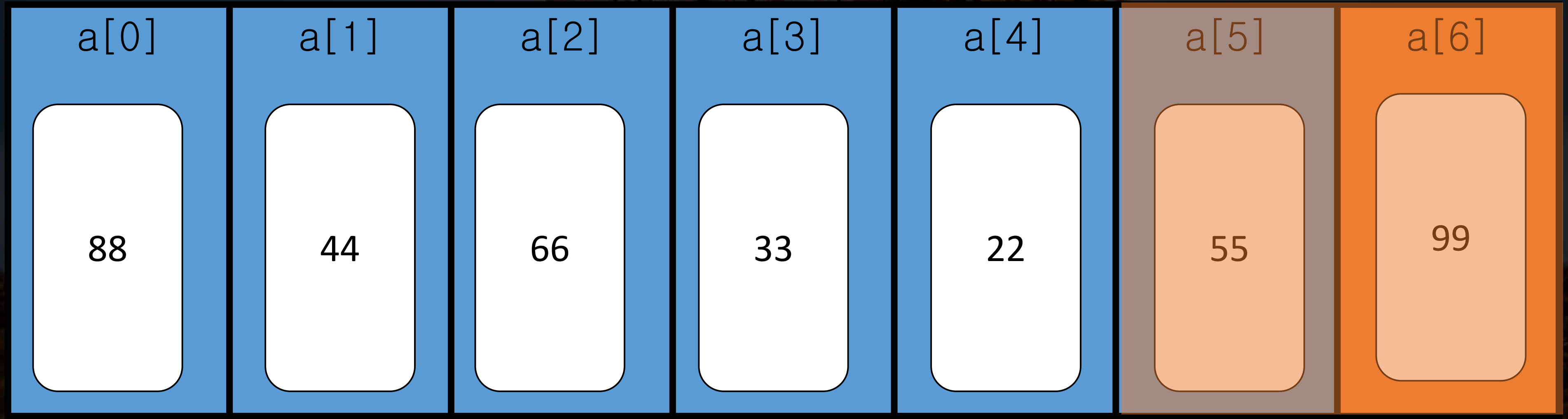
Get the Highest Priority

Remove root – exchange with last (rightmost) leaf

Fix up heap (excluding last leaf)



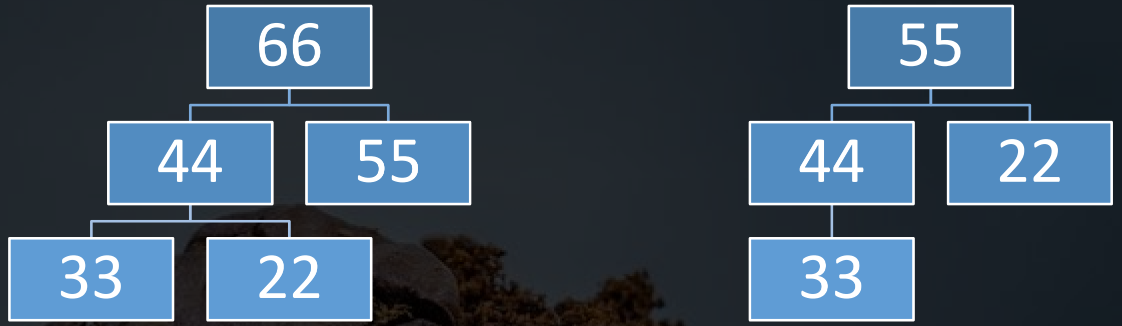


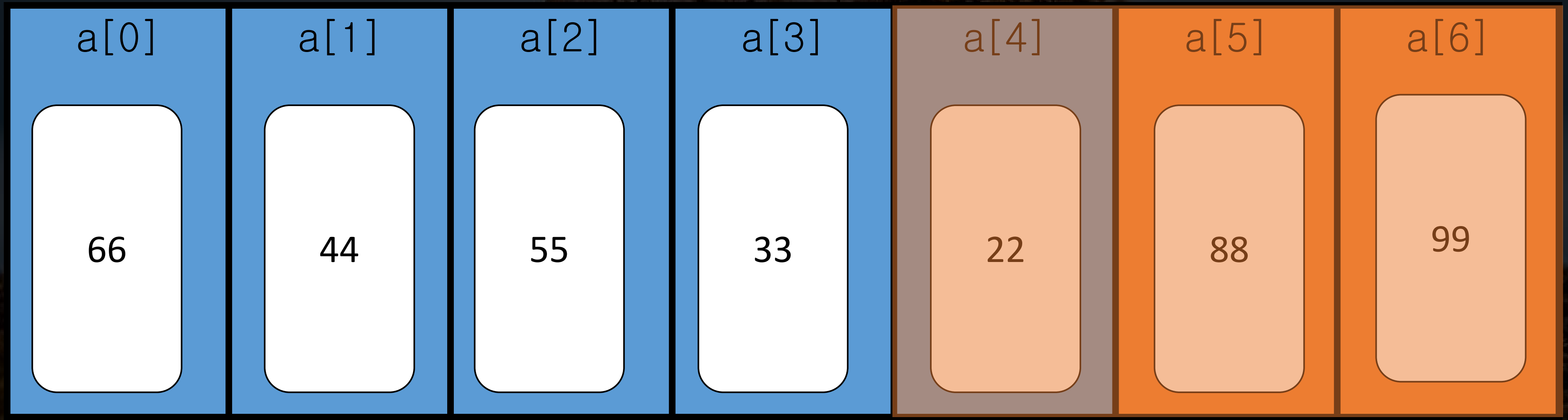


Get the Highest Priority

Remove root – exchange with last (rightmost) leaf

Fix up heap (excluding last leaf)

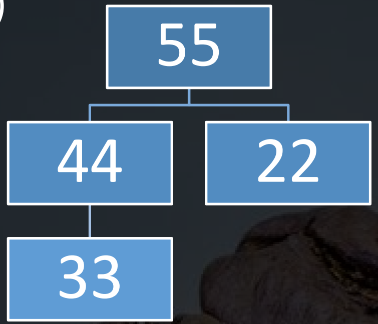


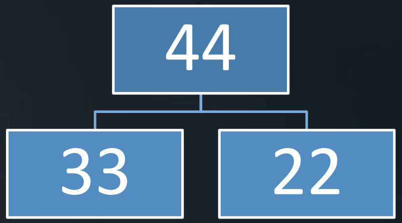


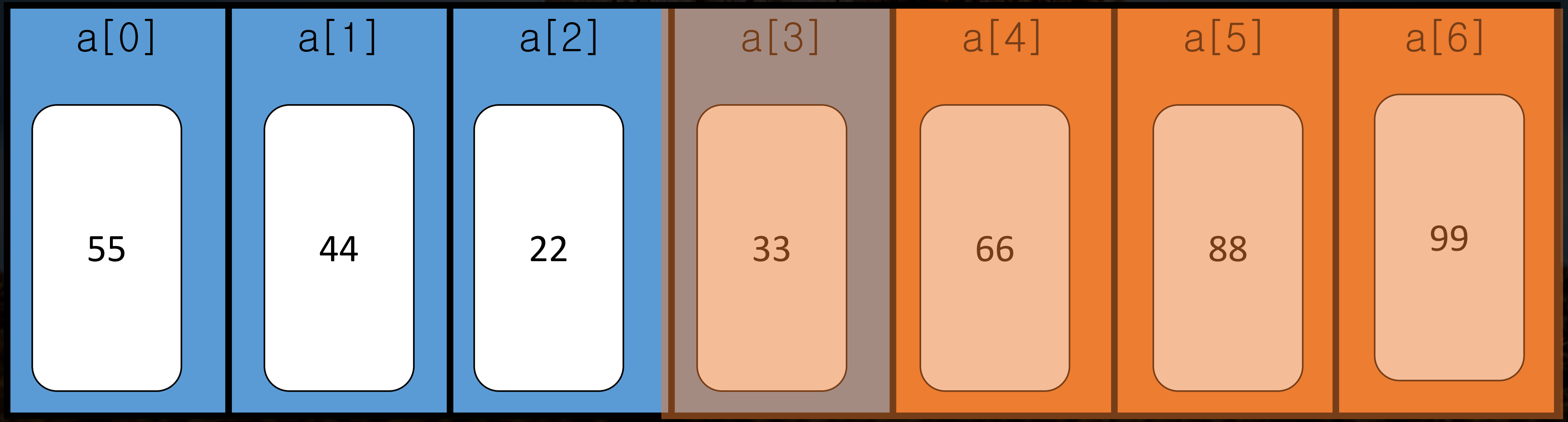
Get the Highest Priority

Remove root – exchange with last (rightmost) leaf

Fix up heap (excluding last leaf)



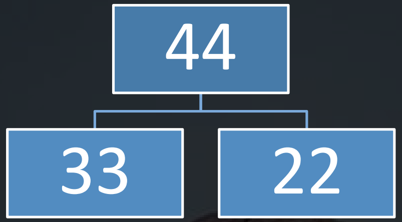


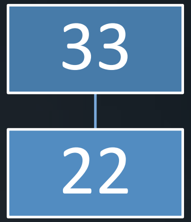


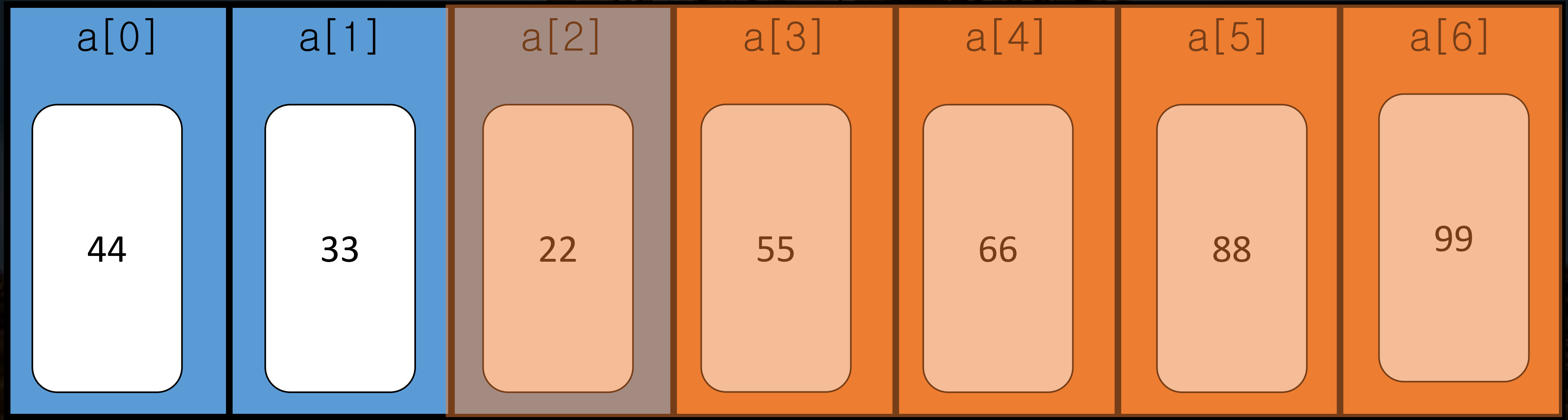
Get the Highest Priority

Remove root – exchange with last (rightmost) leaf

Fix up heap (excluding last leaf)



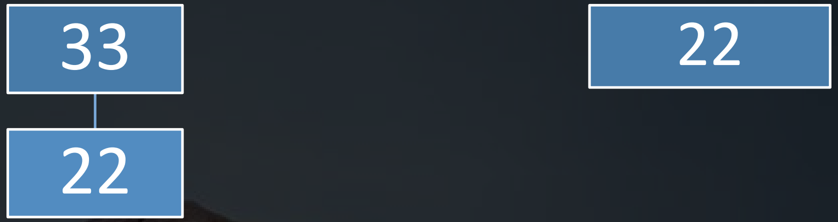


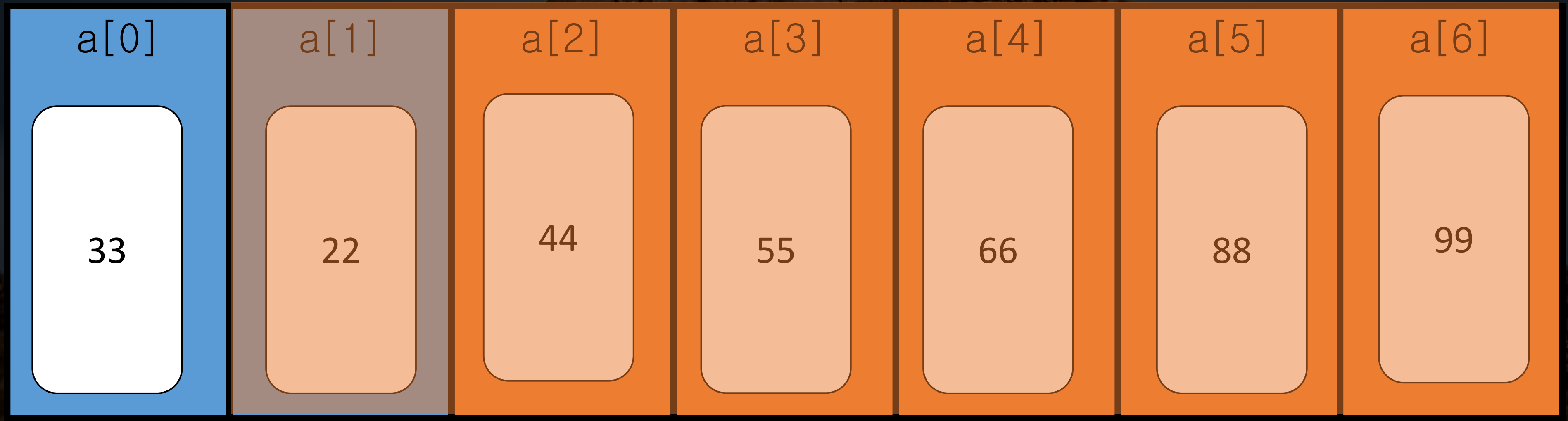


Get the Highest Priority

Remove root – exchange with last (rightmost) leaf

Fix up heap (excluding last leaf)

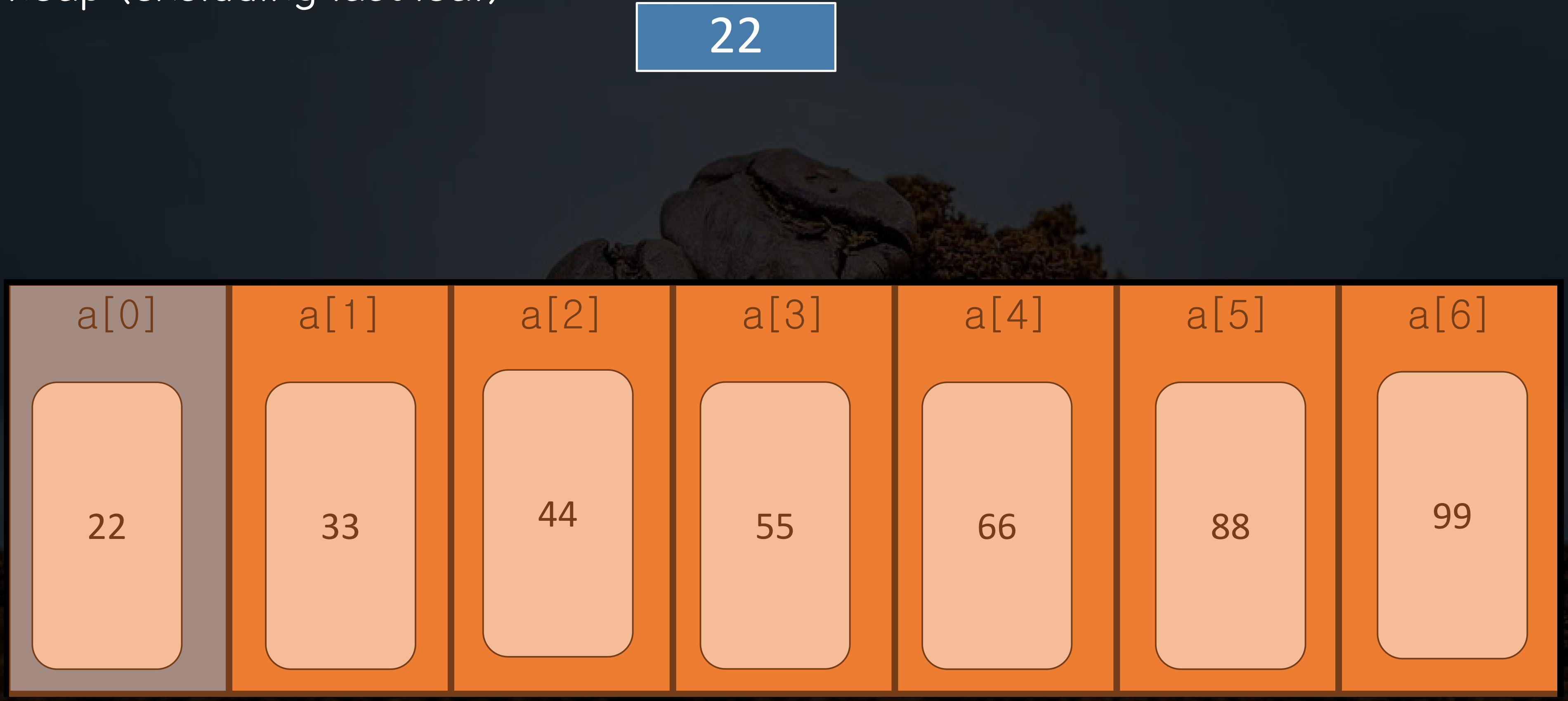




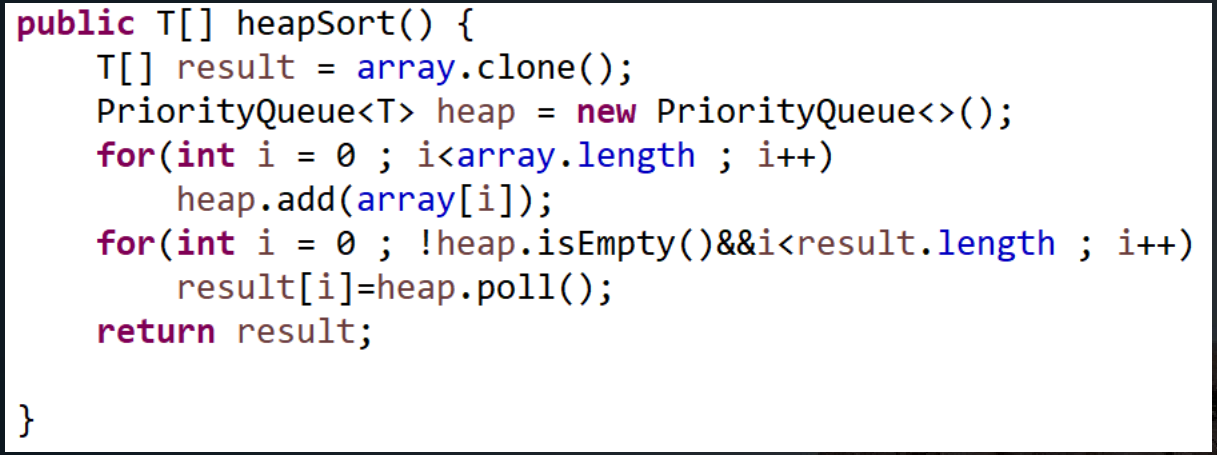
Get the Highest Priority

Remove root – exchange with last (rightmost) leaf

Fix up heap (excluding last leaf)



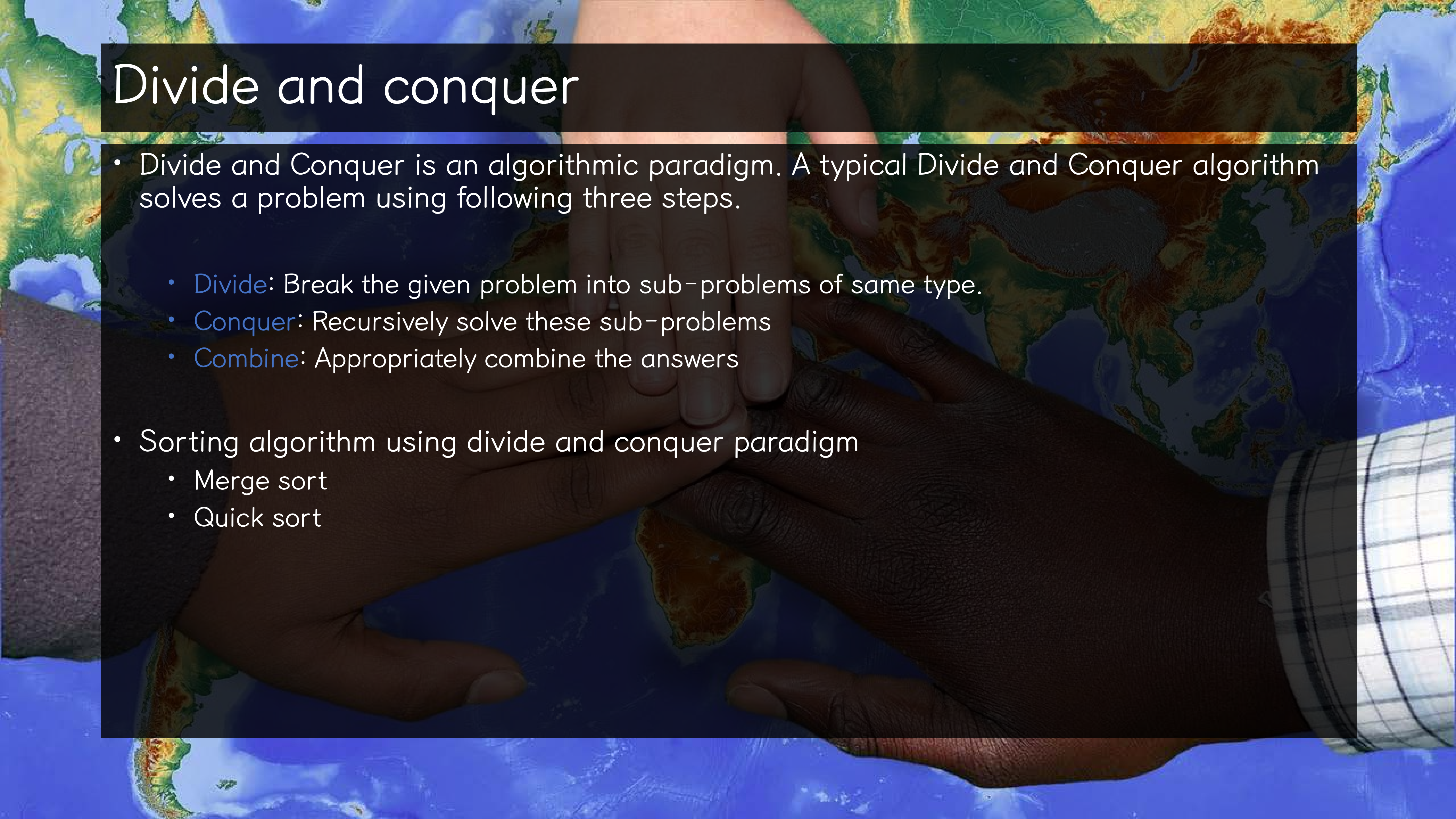
Heap Sort



```
public T[] heapSort() {
    T[] result = array.clone();
    PriorityQueue<T> heap = new PriorityQueue<>();
    for(int i = 0 ; i<array.length ; i++)
        heap.add(array[i]);
    for(int i = 0 ; !heap.isEmpty()&&i<result.length ; i++)
        result[i]=heap.poll();
    return result;

}
```

- Time complexity
  - Bottom-up construction of maxheap: O(n)
  - n X removeMax
  - Time complexity of removeMax = O(log n)
  - O(n)+n X O(log n) = O(*n* log *1*)

Merge Sort



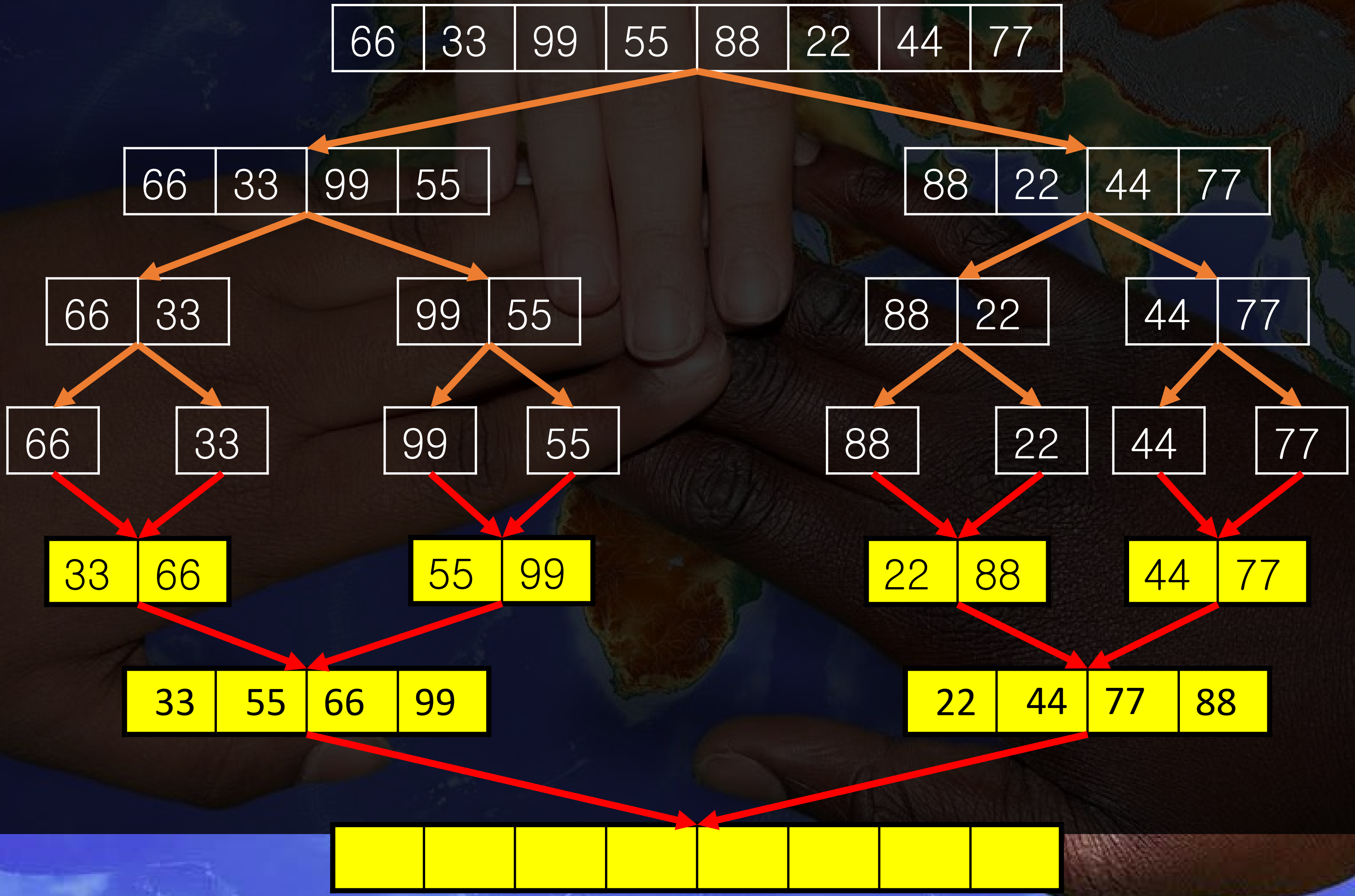
Divide and conquer

- Divide and Conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps.
  - Divide: Break the given problem into sub-problems of same type.
  - Conquer: Recursively solve these sub-problems
  - Combine: Appropriately combine the answers
- Sorting algorithm using divide and conquer paradigm
  - Merge sort
  - Quick sort

Merge sorting

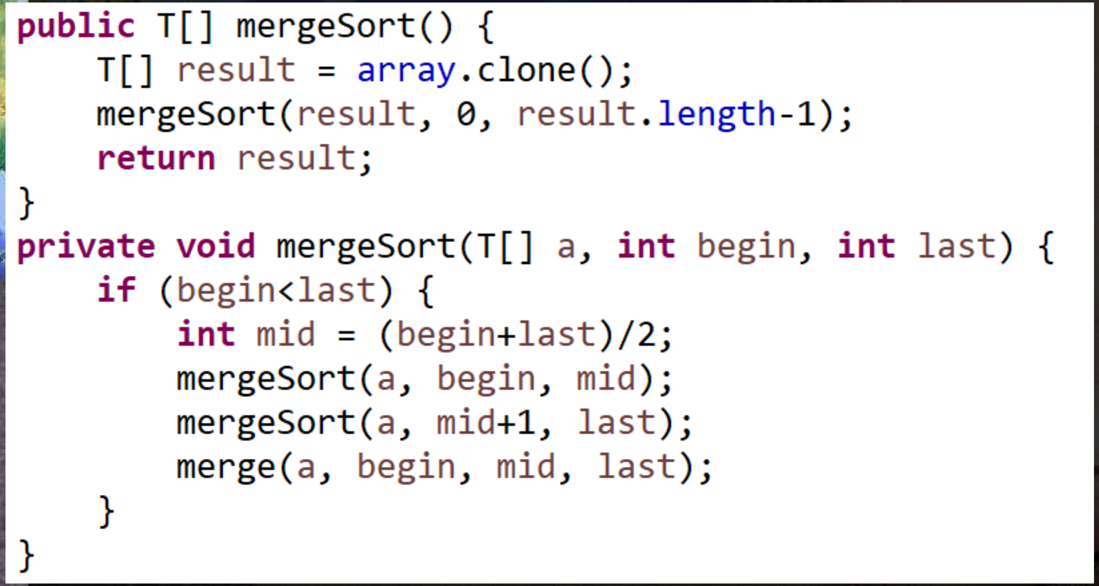
- Pairwise comparison-based sorting algorithm.
- divide a half and merge sort for each half and merge them
- Algorithm
  - Break the array into two halves.
  - Mergesort the left half.
  - Mergesort the right half.
  - Merge the two subarrays into a sorted array.

Merge sort

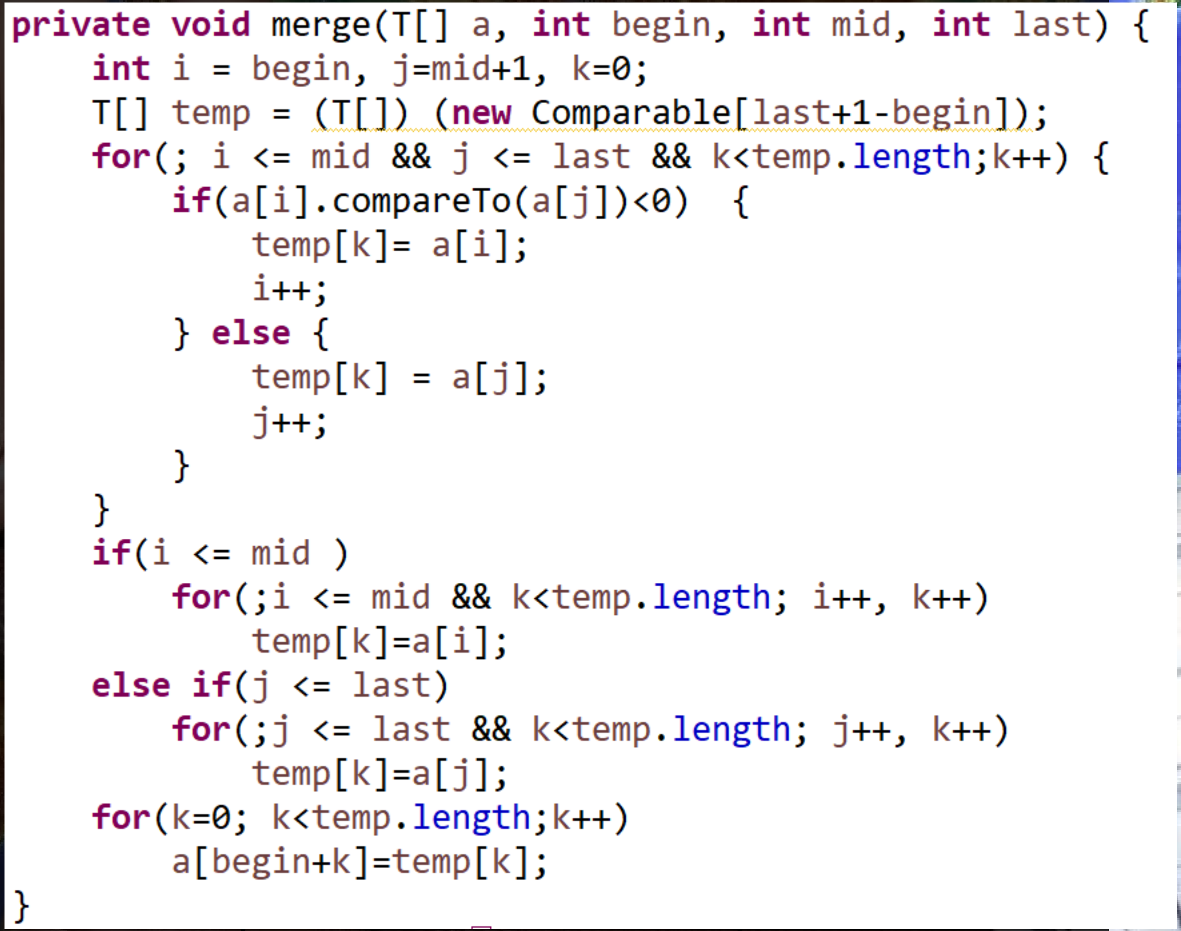


66 33 99 55 88 22 44 77
66 33 99 55
88 22 44 77
66 33
99 55
88 22
44 77
66
33
99
55
88
22
44
77
33 66
55 99
22 88
44 77
33 55 66 99
22 44 77 88

Merge sorting code



```
public T[] mergeSort() {
    T[] result = array.clone();
    mergeSort(result, 0, result.length-1);
    return result;
}
private void mergeSort(T[] a, int begin, int last) {
    if (begin<last) {
        int mid = (begin+last)/2;
        mergeSort(a, begin, mid);
        mergeSort(a, mid+1, last);
        merge(a, begin, mid, last);
    }
}
```



```
private void merge(T[] a, int begin, int mid, int last) {
    int i = begin, j=mid+1, k=0;
    T[] temp = (T[]) (new Comparable[last+1-begin]);
    for(; i <= mid && j <= last && k<temp.length;k++) {
        if(a[i].compareTo(a[j])<0)  {
            temp[k]= a[i];
            i++;
        } else {
            temp[k] = a[j];
            j++;
        }
    }
    if(i <= mid )
        for(;i <= mid && k<temp.length; i++, k++)
            temp[k]=a[i];
    else if(j <= last)
        for(;j <= last && k<temp.length; j++, k++)
            temp[k]=a[j];
    for(k=0; k<temp.length;k++)
        a[begin+k]=temp[k];
}
```

Time complexity: $T(n) = \Theta(n \log n)$

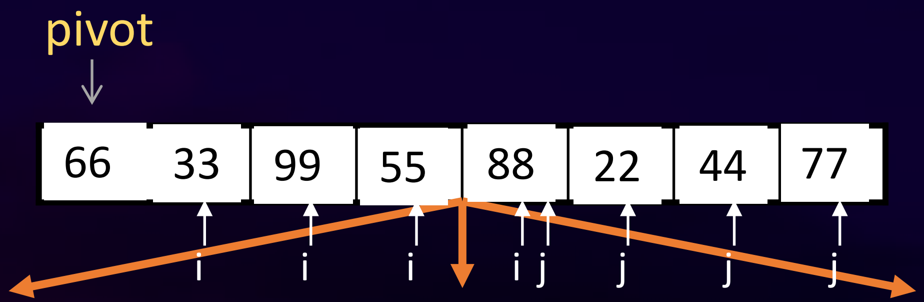
p q

Quick Sort

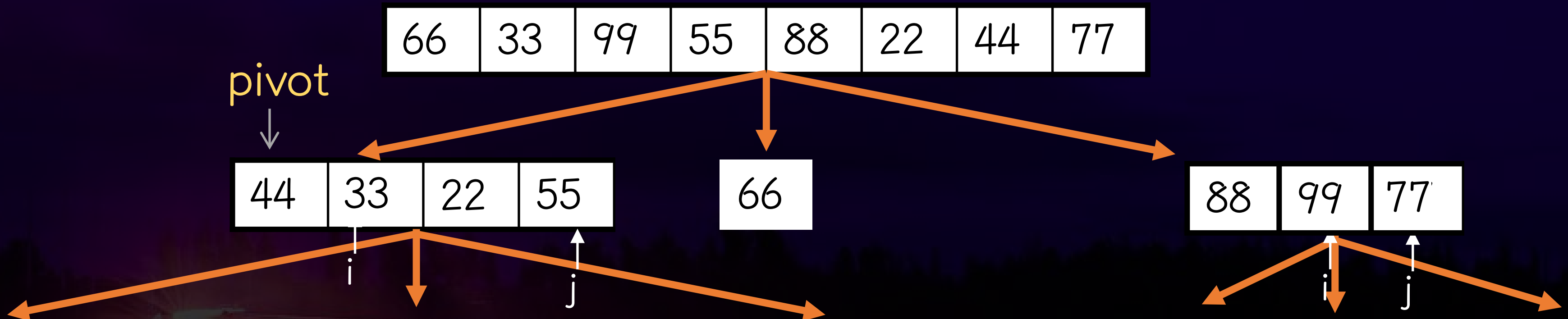
Quick sort

- recursive divide and conquer algorithm
- Algorithm
  - Pick an element, called a pivot, from the list.
  - Reorder the list so that all elements with values less than the pivot come before the pivot, whil e all elements with values greater than the pivot come after it (equal values can go either way).
  - After this partitioning, the pivot is in its final position. This is called the partition operation.
  - Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

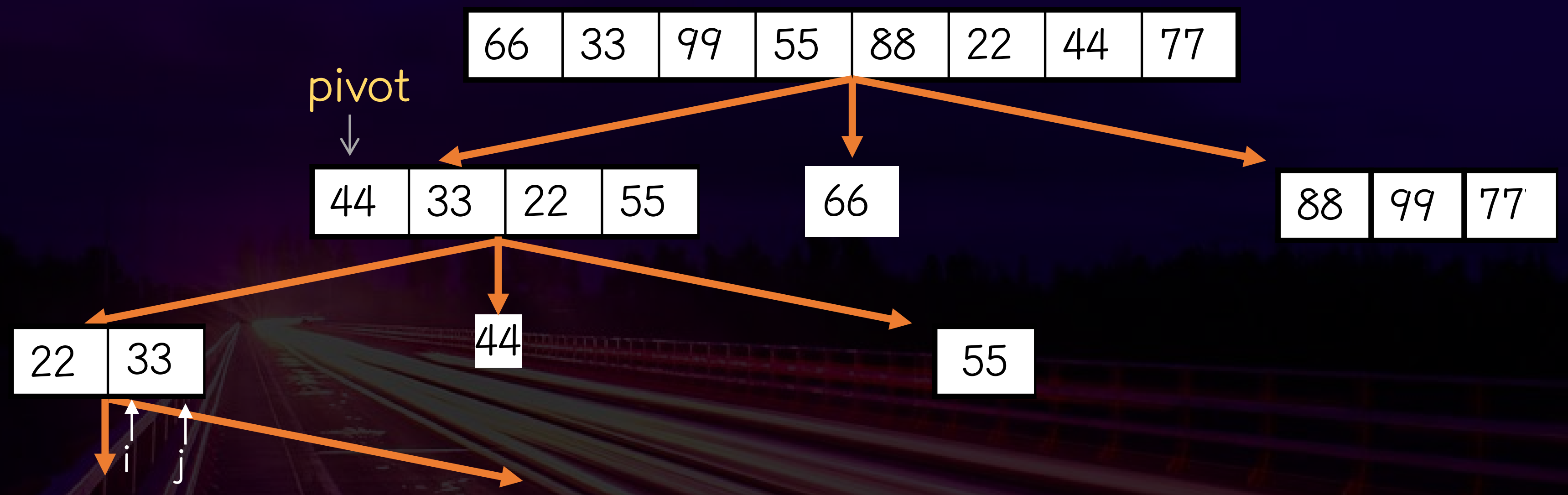
Quick sort



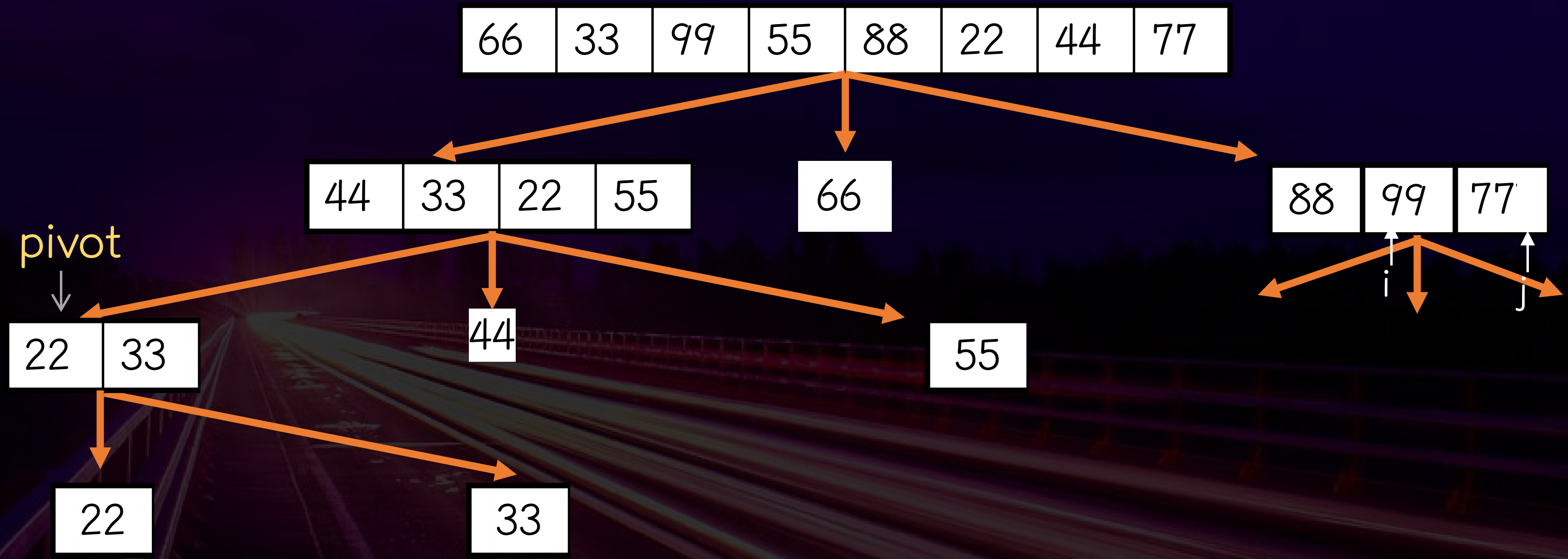
Quick sort



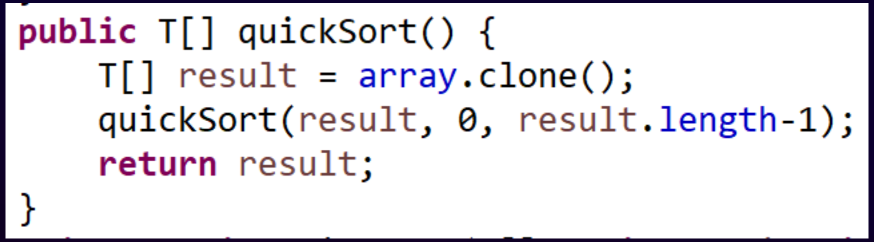
Quick sort



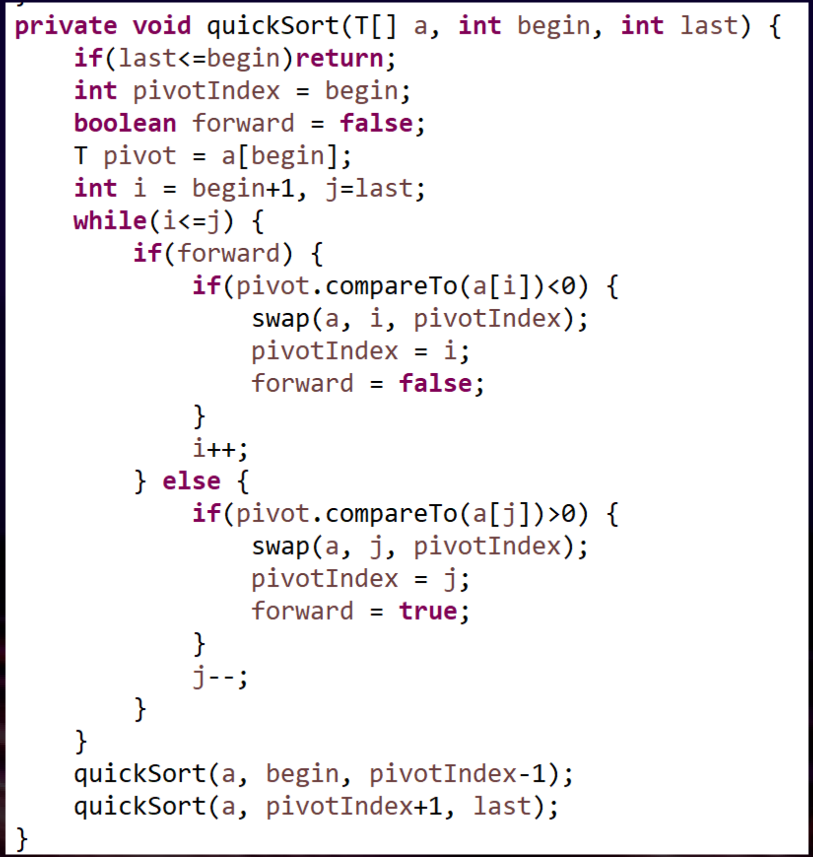
Quick sort


66 33 99 55 88 22 44 77
44 33 22 55
66
88 99 77
i
j
pivot
22 33
44
55
22
33

Quick sort



```java
public T[] quickSort() {
    T[] result = array.clone();
    quickSort(result, 0, result.length-1);
    return result;
}
```

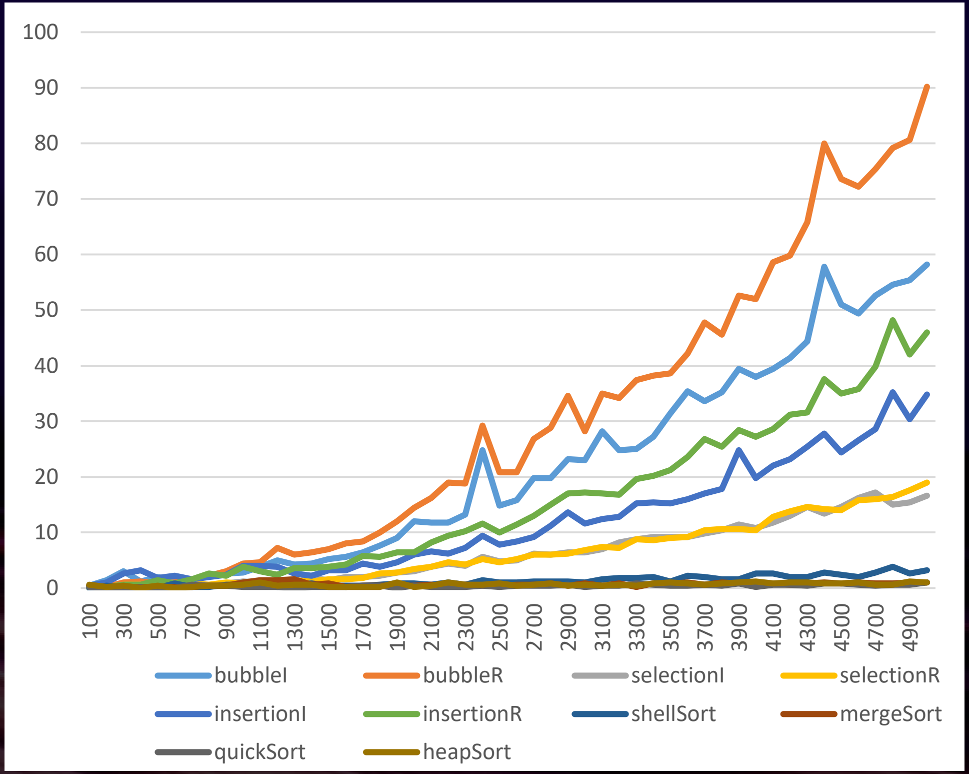


```java
private void quickSort(T[] a, int begin, int last) {
    if(last<=begin)return;
    int pivotIndex = begin;
    boolean forward = false;
    T pivot = a[begin];
    int i = begin+1, j=last;
    while(i<=j) {
        if(forward) {
            if(pivot.compareTo(a[i])<0) {
                swap(a, i, pivotIndex);
                pivotIndex = i;
                forward = false;
            }
            i++;
        } else {
            if(pivot.compareTo(a[j])>0) {
                swap(a, j, pivotIndex);
                pivotIndex = j;
                forward = true;
            }
            j--;
        }
    }
    quickSort(a, begin, pivotIndex-1);
    quickSort(a, pivotIndex+1, last);
}
```

- Time complexity
  - average : $\Theta(n\log n)$
  - worst case: $\Theta(n^2)$
  - quick sort is significantly faster in practice than other $\Theta(n\log n)$ algorithms, because its inner l
    oop can be efficiently implemented on most architectures.

Comparison of pairwise sorting

	Worst case	Average
bubble sorting	$O(n^2)$	$O(n^2)$
selection sorting	$O(n^2)$	$O(n^2)$
insertion sorting	$O(n^2)$	$O(n^2)$
shell sorting	$O(n^2)$	$O(n^{1.5})$
heap sorting	$O(nlgn)$	$O(nlgn)$
merge sorting	$O((nlgn)$	$O(nlgn)$
quick sorting	$O(n^2)$	$O(nlgn)$



Distribute Sort

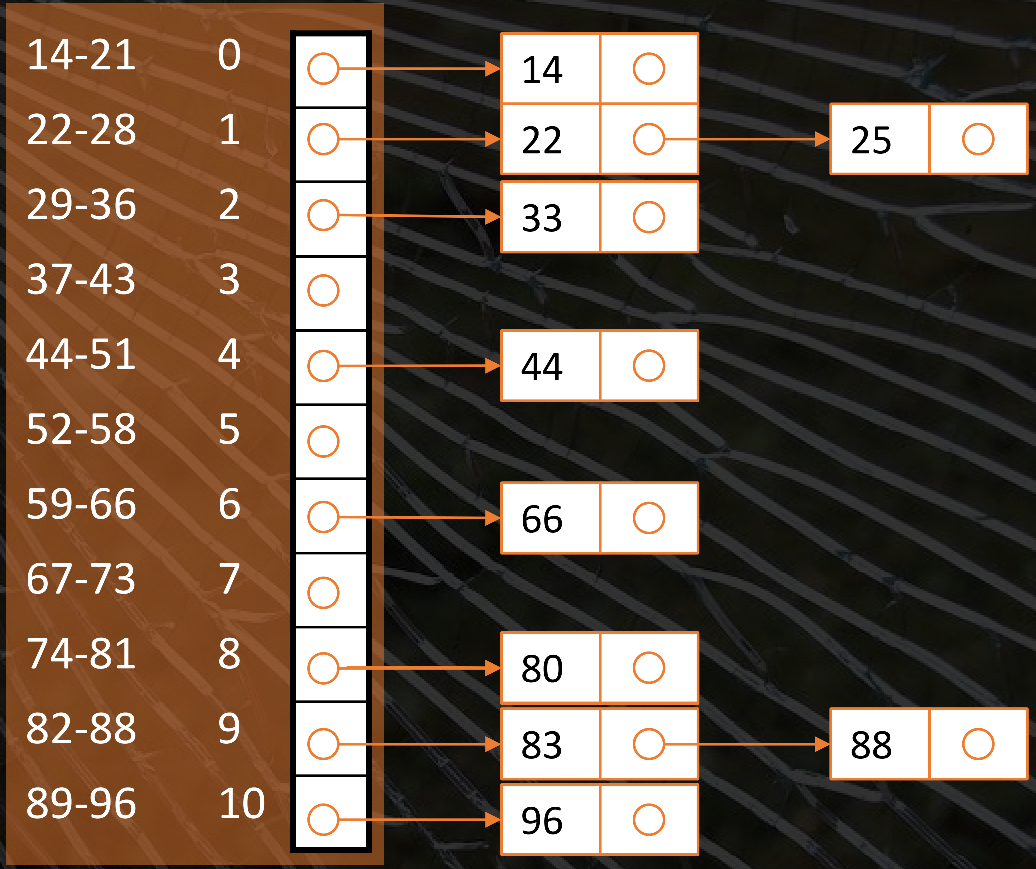
Bucket sorting

- The bucket sort uses an array of containers called “buckets” to hold the keys temporarily . It is an example of a distribution sort, which begins by distributing the keys among the buckets.
- Algorithm
  - Input: a sequence of *n* numeric keys $\{a_0, a_1, \cdots a_{n-1}\}$
  - Allocate a sequence of n buckets $\{B_0, B_1, \cdots B_{n-1}\}$
  - Partition the interval $[a, b)$ into a sequence of *n* equal subintervals $\{I_0, I_1, \cdots I_{n-1}\}$ where each $I_j = [x_j, x_{j+1})$has length $\Delta x_j = \frac{b-a}{n}$.
  - Distribute the keys a_i among the buckets according to which subinterval a_j is in:$a_i \in I_j \Rightarrow a_i$ is put into bucket B_j.
  - Sort each bucket.
  - Copy the keys back from the buckets to the sequence in order of the buckets, $B_0, B_1, \cdots B_{n-1}$, keeping the order within each bucket.

$I_j = [x_j, x_{j+1})$

Example

- { 66, 96, 22, 80, 14, 83, 77, 44, 25, 88, 33}
  - 11 keys. ➔ 11 buckets
  - Min: 14, max: 96
- Make buckets
- Insertion sort

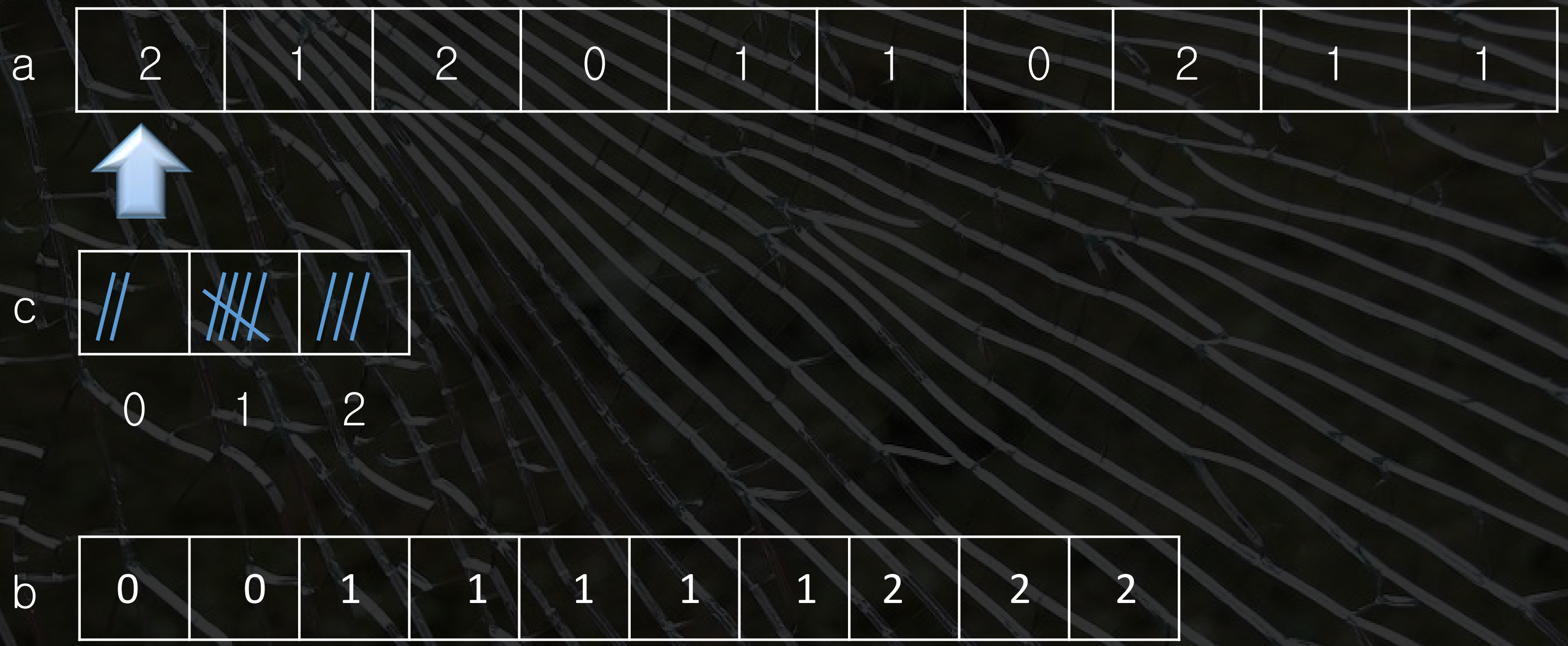


Counting sorting

- The counting sort is another distribution sort.
- Like the bucket sort, it distributes the keys according to their values. It works well when the number values are much less than the number of keys.
- Algorithm
  - Find the highest and lowest elements of the set
  - Count the different elements in the array. (E.g. Set[4,4,4,1,1] would give three 4's and two 1's)
  - Accumulate the counts. Fill the destination array from backwards: put each element to its co untth position.
  - Each time you put in a new element decrease its count.

Example

$a = \{2, 1, 2, 0, 1, 1, 0, 2, 1, 1\}$



Radix sorting

- https://youtu.be/ibtN8rY7V5k
- Sorting algorithm that sorts integers by processing individual digits, by comparing individual digits sharing the same significant position.
- Two classifications
  - LSD: Least Signification Digit
  - MSD: Most Significant Digit
- Algorithm of LSD
  - Do step 2 for each digit j, starting with the least significant j=0
  - Apply the counting sort, keying on digit j.

Example: LSD

3	6

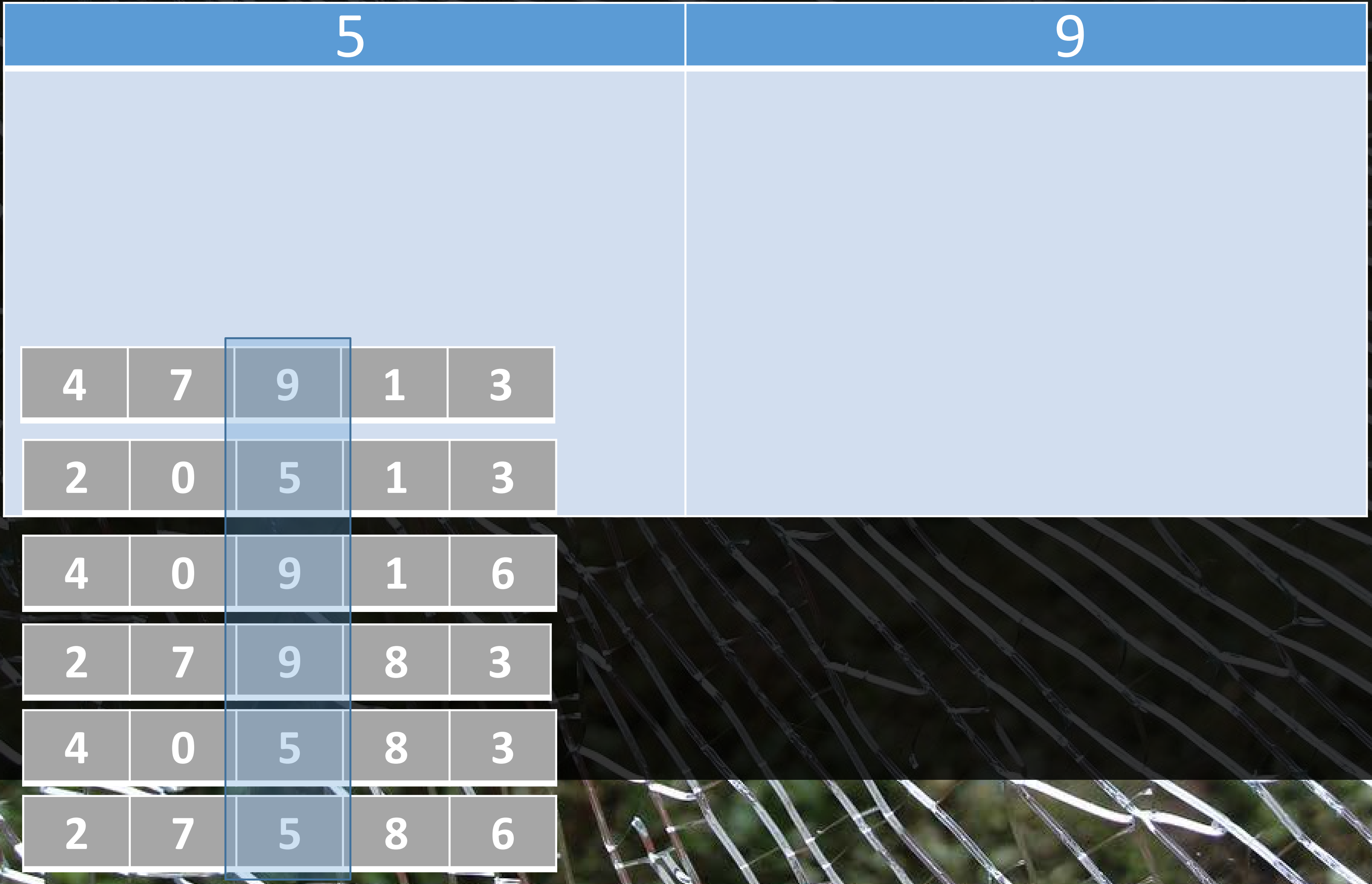
2	7	5	8	6
4	7	9	1	3
4	0	9	1	6
2	7	9	8	3
4	0	5	8	3
2	0	5	1	3

Example: LSD

1	8

4	7	9	1	3
2	7	9	8	3
4	0	5	8	3
2	0	5	1	3
2	7	5	8	6
4	0	9	1	6

Example: LSD

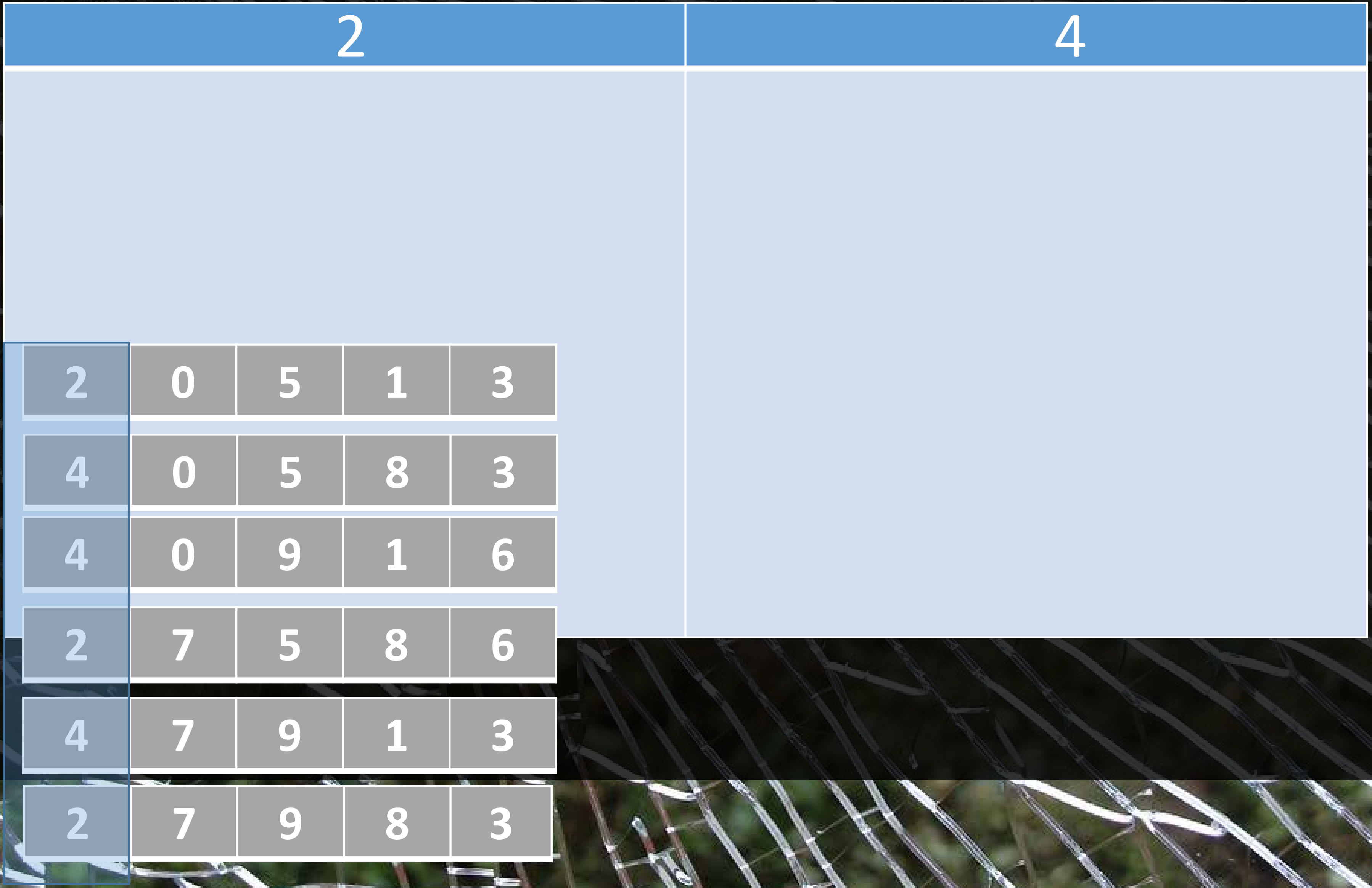


Example: LSD

0	7

2	0	5	1	3
4	0	5	8	3
2	7	5	8	6
4	7	9	1	3
4	0	9	1	6
2	7	9	8	3

Example: LSD



2	0	5	1	3
4	0	5	8	3
4	0	9	1	6
2	7	5	8	6
4	7	9	1	3
2	7	9	8	3

Thank you