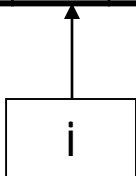


Searching and Sorting

Sequential search

- **sequential search:** Locates a target value in an array/list by examining each element from start to finish.
 - How many elements will it need to examine?
 - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103



- Notice that the array is sorted. Could we take advantage of this?

Binary search

- **binary search:** Locates a target value in a *sorted* array/list by successively eliminating half of the array from consideration.
 - How many elements will it need to examine?
 - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

min

mid

max

The Arrays class

- Class Arrays in `java.util` has many useful array methods:

Method name	Description
<code>binarySearch(array, value)</code>	returns the index of the given value in a <i>sorted</i> array (or <code>< 0</code> if not found)
<code>binarySearch(array, minIndex, maxIndex, value)</code>	returns index of given value in a <i>sorted</i> array between indexes <i>min</i> / <i>max</i> - 1 (<code>< 0</code> if not found)
<code>copyOf(array, length)</code>	returns a new resized copy of an array
<code>equals(array1, array2)</code>	returns <code>true</code> if the two arrays contain same elements in the same order
<code>fill(array, value)</code>	sets every element to the given value
<code>sort(array)</code>	arranges the elements into sorted order
<code>toString(array)</code>	returns a string representing the array, such as <code>"[10, 30, -25, 17]"</code>

- Syntax: `Arrays.methodName(parameters)`

Arrays.binarySearch

```
// searches an entire sorted array for a given value  
// returns its index if found; a negative number if not found  
// Precondition: array is sorted
```

```
Arrays.binarySearch(array, value)
```

```
// searches given portion of a sorted array for a given value  
// examines minIndex (inclusive) through maxIndex (exclusive)  
// returns its index if found; a negative number if not found  
// Precondition: array is sorted
```

```
Arrays.binarySearch(array, minIndex, maxIndex, value)
```

- The `binarySearch` method in the `Arrays` class searches an array very efficiently if the array is sorted.
 - You can search the entire array, or just a range of indexes (useful for "unfilled" arrays such as the one in `ArrayIntList`)
 - If the array is not sorted, you may need to sort it first

Using `binarySearch`

```
// index    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
int[] a = {-4, 2, 7, 9, 15, 19, 25, 28, 30, 36, 42, 50, 56, 68, 85, 92};

int index  = Arrays.binarySearch(a, 0, 16, 42);    // index1 is 10
int index2 = Arrays.binarySearch(a, 0, 16, 21);    // index2 is -7
```

- `binarySearch` returns the index where the value is found
- if the value is *not* found, `binarySearch` returns:
 - (`insertionPoint` + 1)
 - where `insertionPoint` is the index where the element *would* have been, if it had been in the array in sorted order.
 - To insert the value into the array, negate `insertionPoint` + 1

```
int indexToInsert21 = -(index2 + 1);    // 6
```

Binary search code

```
// Returns the index of an occurrence of target in a,  
// or a negative number if the target is not found.  
// Precondition: elements of a are in sorted order  
public static int binarySearch(int[] a, int target) {  
    int min = 0;  
    int max = a.length - 1;  
  
    while (min <= max) {  
        int mid = (min + max) / 2;  
        if (a[mid] < target) {  
            min = mid + 1;  
        } else if (a[mid] > target) {  
            max = mid - 1;  
        } else {  
            return mid;    // target found  
        }  
    }  
  
    return -(min + 1);    // target not found  
}
```

Recursive binary search

- Write a recursive `binarySearch` method.
 - If the target value is not found, return its negative insertion point.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

```
int index  = binarySearch(data, 42);  // 10
int index2 = binarySearch(data, 66);  // -14
```


Exercise solution

```
// Returns the index of an occurrence of the given value in  
// the given array, or a negative number if not found.
```

```
// Precondition: elements of a are in sorted order
```

```
public static int binarySearch(int[] a, int target) {  
    return binarySearch(a, target, 0, a.length - 1);  
}
```

```
// Recursive helper to implement search behavior.
```

```
private static int binarySearch(int[] a, int target,  
                                int min, int max) {  
    if (min > max) {  
        return -(min + 1);           // target not found  
    } else {  
        int mid = (min + max) / 2;  
        if (a[mid] < target) {       // too small; go right  
            return binarySearch(a, target, mid + 1, max);  
        } else if (a[mid] > target) { // too large; go left  
            return binarySearch(a, target, min, mid - 1);  
        } else {  
            return mid;              // target found; a[mid] == target  
        }  
    }  
}
```

Binary search and objects

- Can we `binarySearch` an array of `Strings`?
 - Operators like `<` and `>` do not work with `String` objects.
 - But we do think of strings as having an alphabetical ordering.
- **natural ordering**: Rules governing the relative placement of all values of a given type.
- **comparison function**: Code that, when given two values *A* and *B* of a given type, decides their relative ordering:
 - $A < B$, $A == B$, $A > B$

The compareTo method

- The standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method.
 - Example: in the `String` class, there is a method:

```
public int compareTo(String other)
```
- A call of `A.compareTo(B)` will return:
 - a value < 0 if **A** comes "before" **B** in the ordering,
 - a value > 0 if **A** comes "after" **B** in the ordering,
 - or 0 if **A** and **B** are considered "equal" in the ordering.

Runtime Efficiency

- **efficiency**: A measure of the use of computing resources by code.
 - can be relative to speed (time), memory (space), etc.
 - most commonly refers to run time
- Assume the following:
 - Any single Java statement takes the same amount of time to run.
 - A method call's runtime is measured by the total of the statements inside the method's body.
 - A loop's runtime, if the loop repeats N times, is N times the runtime of the statements in its body.

Range algorithm

```
// returns the range of values in the given array;  
// the difference between elements furthest apart  
// example: range({17, 29, 11, 4, 20, 8}) is 25  
public static int range(int[] numbers) {  
    int maxDiff = 0;        // look at each pair of values  
    for (int i = 0; i < numbers.length; i++) {  
        for (int j = 0; j < numbers.length; j++) {  
            int diff = Math.abs(numbers[j] - numbers[i]);  
            if (diff > maxDiff) {  
                maxDiff = diff;  
            }  
        }  
    }  
    return maxDiff;  
}
```

Range algorithm 2

```
// returns the range of values in the given array;  
// the difference between elements furthest apart  
// example: range({17, 29, 11, 4, 20, 8}) is 25  
public static int range(int[] numbers) {  
    int maxDiff = 0;        // look at each pair of values  
    for (int i = 0; i < numbers.length; i++) {  
        for (int j = i + 1; j < numbers.length; j++) {  
            int diff = Math.abs(numbers[j] - numbers[i]);  
            if (diff > maxDiff) {  
                maxDiff = diff;  
            }  
        }  
    }  
    return diff;  
}
```

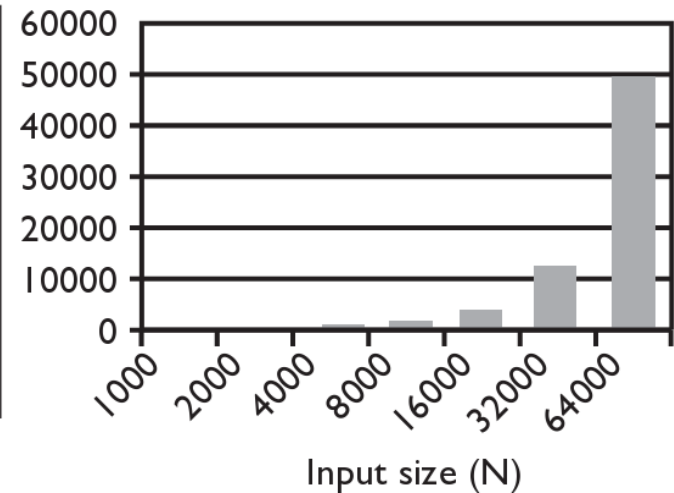
Range algorithm 3

```
// returns the range of values in the given array;  
// example: range({17, 29, 11, 4, 20, 8}) is 25  
public static int range(int[] numbers) {  
    int max = numbers[0];    // find max/min values  
    int min = max;  
    for (int i = 1; i < numbers.length; i++) {  
        if (numbers[i] < min) {  
            min = numbers[i];  
        }  
        if (numbers[i] > max) {  
            max = numbers[i];  
        }  
    }  
    return max - min;  
}
```

Runtime of first 2 versions

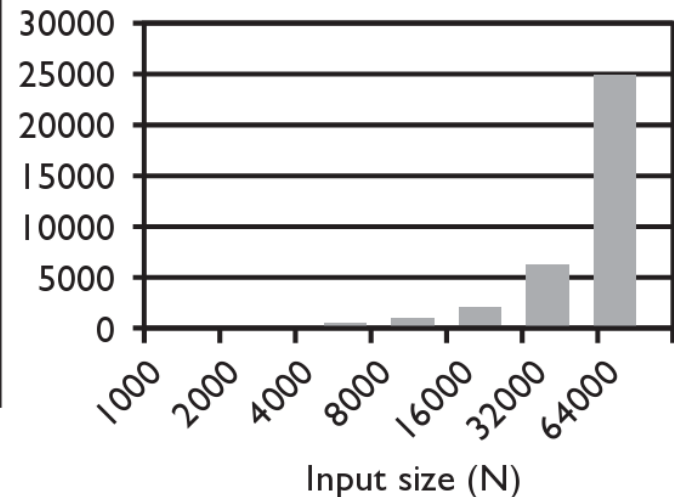
- Version 1:

N	Runtime (ms)
1000	15
2000	47
4000	203
8000	781
16000	3110
32000	12563
64000	49937



- Version 2:

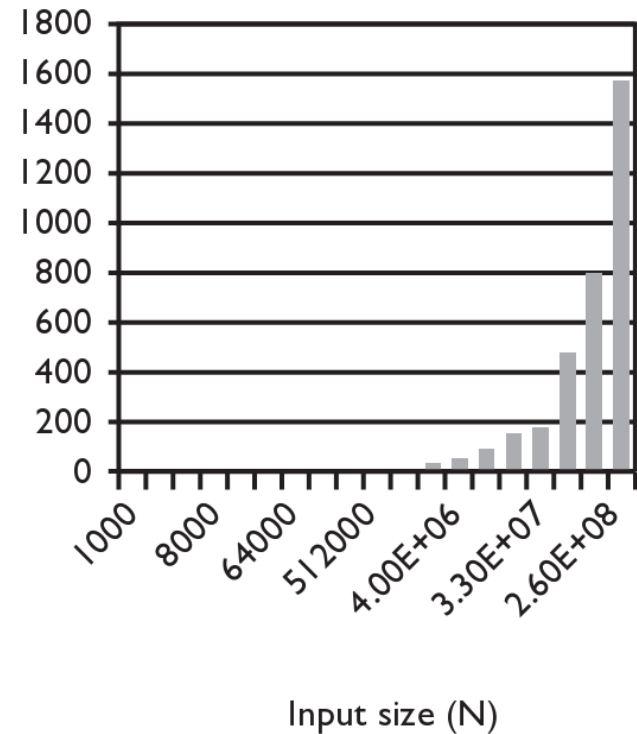
N	Runtime (ms)
1000	16
2000	16
4000	110
8000	406
16000	1578
32000	6265
64000	25031



Runtime of 3rd version

- Version 3:

N	Runtime (ms)
1000	0
2000	0
4000	0
8000	0
16000	0
32000	0
64000	0
128000	0
256000	0
512000	0
1e6	0
2e6	16
4e6	31
8e6	47
1.67e7	94
3.3e7	188
6.5e7	453
1.3e8	797
2.6e8	1578



Sorting

- **sorting**: Rearranging the values in an array or collection into a specific order (usually into their "natural ordering").
 - one of the fundamental problems in computer science
 - can be solved in many ways:
 - there are many sorting algorithms
 - some are faster/slower than others
 - some use more/less memory than others
 - some work better with specific kinds of data
 - some can utilize multiple computers / processors, ...
 - *comparison-based sorting* : determining order by comparing pairs of elements:
 - `<`, `>`, `compareTo`, ...

Sorting methods in Java

- The Arrays and Collections classes in java.util have a static method sort that sorts the elements of an array/list

```
String[] words = {"foo", "bar", "baz", "ball"};
Arrays.sort(words);
System.out.println(Arrays.toString(words));
// [ball, bar, baz, foo]
```

```
List<String> words2 = new ArrayList<String>();
for (String word : words) {
    words2.add(word);
}
Collections.sort(words2);
System.out.println(words2);
// [ball, bar, baz, foo]
```

Sorting algorithms

- **bogo sort**: shuffle and pray
- **bubble sort**: swap adjacent pairs that are out of order
- **selection sort**: look for the smallest element, move to front
- **insertion sort**: build an increasingly large sorted front portion
- **merge sort**: recursively divide the array in half and sort it
- **heap sort**: place the values into a sorted tree structure
- **quick sort**: recursively partition array based on a middle value

other specialized sorting algorithms:

- **bucket sort**: cluster elements into smaller groups, sort them
- **radix sort**: sort integers by last digit, then 2nd to last, then ...
- ...

Bogo sort

- **bogo sort**: Orders a list of values by repetitively shuffling them and checking if they are sorted.
 - name comes from the word "bogus"

The algorithm:

- Scan the list, seeing if it is sorted. If so, stop.
 - Else, shuffle the values in the list and repeat.
-
- This sorting algorithm (obviously) has terrible performance!
 - What is its runtime?

Bogo sort code

// Places the elements of a into sorted order.

```
public static void bogoSort(int[] a) {  
    while (!isSorted(a)) {  
        shuffle(a);  
    }  
}
```

// Returns true if a's elements are in sorted order.

```
public static boolean isSorted(int[] a) {  
    for (int i = 0; i < a.length - 1; i++) {  
        if (a[i] > a[i + 1]) {  
            return false;  
        }  
    }  
    return true;  
}
```

Bogo sort code, cont'd.

```
// Shuffles an array of ints by randomly swapping each
// element with an element ahead of it in the array.
public static void shuffle(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        // pick a random index in [i+1, a.length-1]
        int range = a.length - 1 - (i + 1) + 1;
        int j = (int) (Math.random() * range + (i + 1));
        swap(a, i, j);
    }
}
```

```
// Swaps a[i] with a[j].
public static void swap(int[] a, int i, int j) {
    if (i != j) {
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}
```

Selection sort

- **selection sort:** Orders a list of values by repeatedly putting the smallest or largest unplaced value into its final position.

The algorithm:

- Look through the list to find the smallest value.
- Swap it so that it is at index 0.
- Look through the list to find the second-smallest value.
- Swap it so that it is at index 1.
- ...
- Repeat until all values are in their proper places.

Selection sort example

- Initial array:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98	25

- After 1st, 2nd, and 3rd passes:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	18	12	22	27	30	36	50	7	68	91	56	2	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	12	22	27	30	36	50	7	68	91	56	18	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	22	27	30	36	50	12	68	91	56	18	85	42	98	25

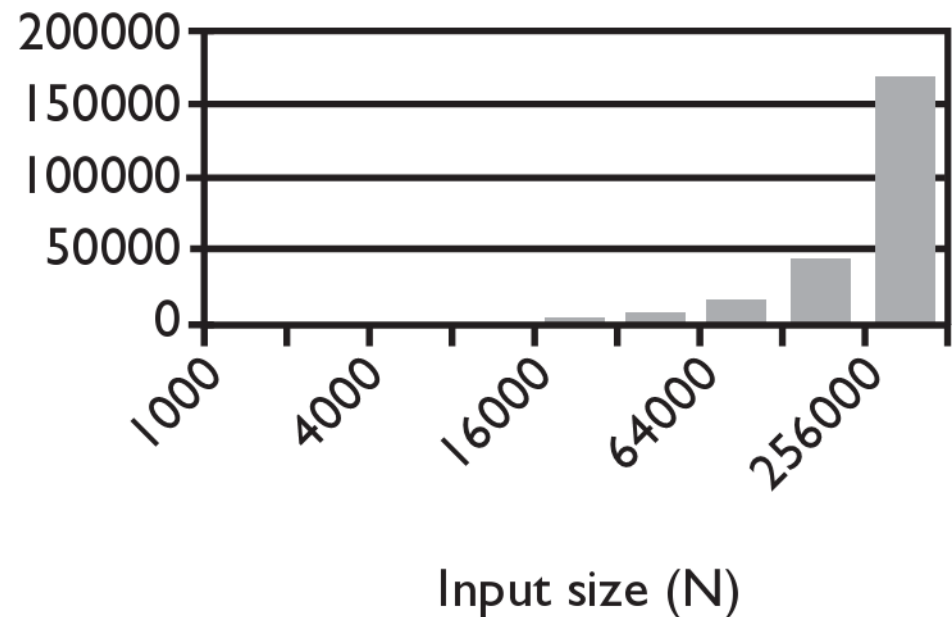
Selection sort code

```
// Rearranges the elements of a into sorted order using
// the selection sort algorithm.
public static void selectionSort(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        // find index of smallest remaining value
        int min = i;
        for (int j = i + 1; j < a.length; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
        }

        // swap smallest value its proper place, a[i]
        swap(a, i, min);
    }
}
```

Selection sort runtime

N	Runtime (ms)
1000	0
2000	16
4000	47
8000	234
16000	657
32000	2562
64000	10265
128000	41141
256000	164985



Similar algorithms

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98	25

- **bubble sort:** Make repeated passes, swapping adjacent values
 - slower than selection sort (has to do more swaps)

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	18	12	-4	22	27	30	36	7	50	68	56	2	85	42	91	25	98

22 \longrightarrow 50 \longrightarrow 91 \longrightarrow 98 \longrightarrow

- **insertion sort:** Shift each element into a sorted sub-array
 - faster than selection sort (examines fewer values)

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	12	18	22	27	30	36	50	7	68	91	56	2	85	42	98	25

sorted sub-array (indexes 0-7)

← 7

Merge sort

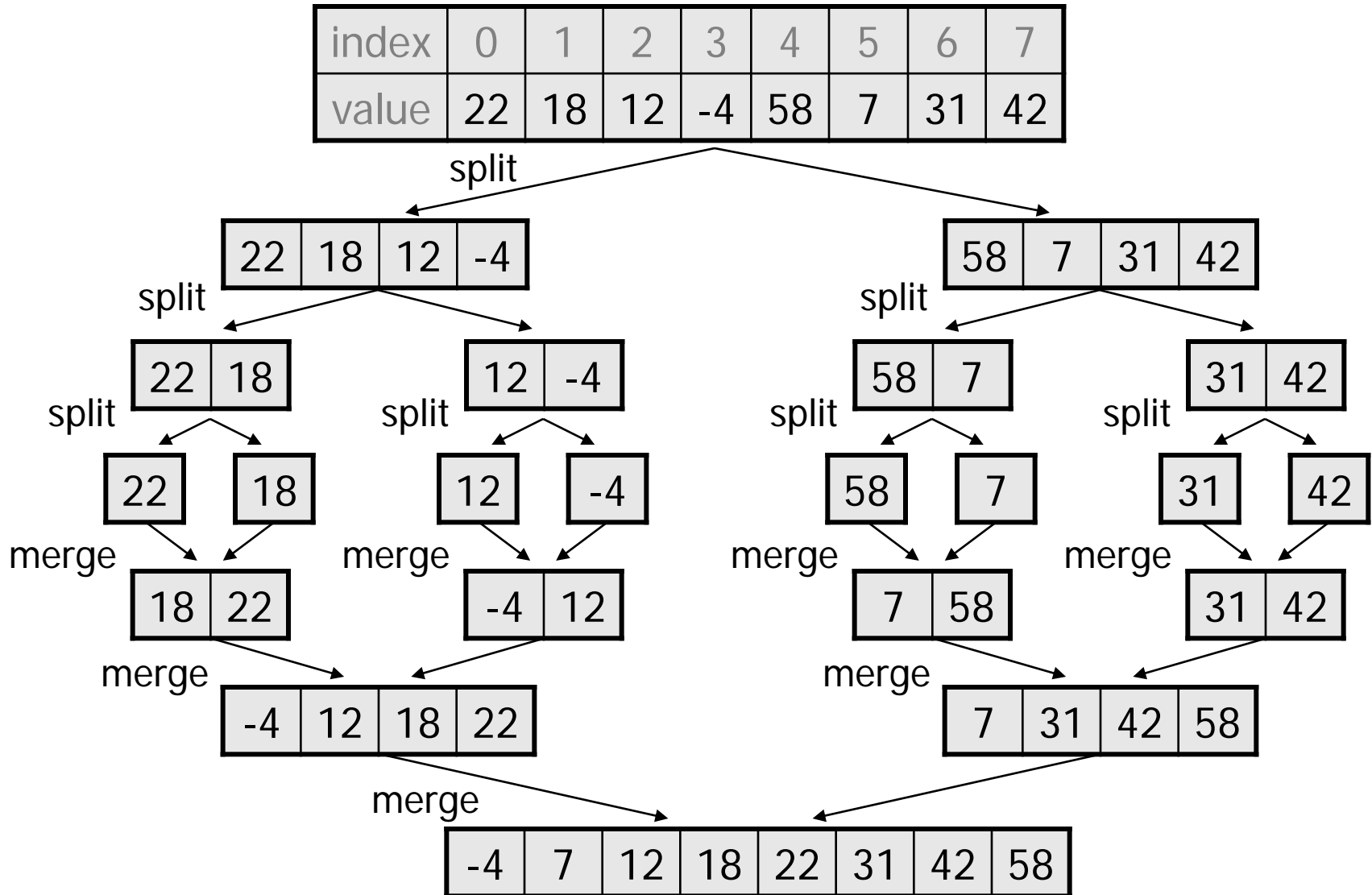
- **merge sort**: Repeatedly divides the data in half, sorts each half, and combines the sorted halves into a sorted whole.

The algorithm:

- Divide the list into two roughly equal halves.
- Sort the left half.
- Sort the right half.
- Merge the two sorted halves into one sorted list.

- Often implemented recursively.
- An example of a "divide and conquer" algorithm.
 - Invented by John von Neumann in 1945

Merge sort example



Merging sorted halves

Subarrays								Next include	Merged array							
0	1	2	3	0	1	2	3		0	1	2	3	4	5	6	7
14	32	67	76	23	41	58	85	14 from left	14							
i1				i2					i							
14	32	67	76	23	41	58	85	23 from right	14	23						
i1				i2					i							
14	32	67	76	23	41	58	85	32 from left	14	23	32					
i1				i2					i							
14	32	67	76	23	41	58	85	41 from right	14	23	32	41				
i1				i2					i							
14	32	67	76	23	41	58	85	58 from right	14	23	32	41	58			
i1				i2					i							
14	32	67	76	23	41	58	85	67 from left	14	23	32	41	58	67		
i1				i2					i							
14	32	67	76	23	41	58	85	76 from left	14	23	32	41	58	67	76	
i1				i2					i							
14	32	67	76	23	41	58	85	85 from right	14	23	32	41	58	67	76	85
				i2					i							i

Merge halves code

```
// Merges the left/right elements into a sorted result.
// Precondition: left/right are sorted
public static void merge(int[] result, int[] left,
                        int[] right) {
    int i1 = 0;    // index into left array
    int i2 = 0;    // index into right array

    for (int i = 0; i < result.length; i++) {
        if (i2 >= right.length ||
            (i1 < left.length && left[i1] <= right[i2])) {
            result[i] = left[i1];    // take from left
            i1++;
        } else {
            result[i] = right[i2];    // take from right
            i2++;
        }
    }
}
```


Merge sort code

```
// Rearranges the elements of a into sorted order using
// the merge sort algorithm.
public static void mergeSort(int[] a) {
    // split array into two halves
    int[] left  = Arrays.copyOfRange(a, 0, a.length/2);
    int[] right = Arrays.copyOfRange(a, a.length/2, a.length);

    // sort the two halves
    ...

    // merge the sorted halves into a sorted whole
    merge(a, left, right);
}
```

Merge sort code 2

```
// Rearranges the elements of a into sorted order using
// the merge sort algorithm (recursive).
public static void mergeSort(int[] a) {
    if (a.length >= 2) {
        // split array into two halves
        int[] left  = Arrays.copyOfRange(a, 0, a.length/2);
        int[] right = Arrays.copyOfRange(a, a.length/2, a.length);

        // sort the two halves
        mergeSort(left);
        mergeSort(right);

        // merge the sorted halves into a sorted whole
        merge(a, left, right);
    }
}
```

Merge sort runtime

N	Runtime (ms)
1000	0
2000	0
4000	0
8000	0
16000	0
32000	15
64000	16
128000	47
256000	125
512000	250
1e6	532
2e6	1078
4e6	2265
8e6	4781
1.6e7	9828
3.3e7	20422
6.5e7	42406
1.3e8	88344

