

COEN 352: Tutorial 4

Recursion, selection sort and insertion sort

Recursion

How do you look up a name in
the phone book?

One Possible Way

Search:

middle page = (first page + last page)/2

Go to middle page;

If (name is on middle page)

done; **//this is the base case**

else if (name is alphabetically before middle page)

last page = middle page **//redefine search area to front half**

Search **//same process on reduced number of pages**

else **//name must be after middle page**

first page = middle page **//redefine search area to back half**

Search **//same process on reduced number of pages**

Overview

Recursion: a definition in terms of itself.

Recursion in algorithms:

- Natural approach to **some** (not all) problems
- A *recursive algorithm* uses itself to solve one or more smaller identical problems

Recursion in Java:

- Recursive methods implement recursive algorithms
- A *recursive method* includes a call to itself

Recursive Methods Must Eventually Terminate

*A recursive method must have
at least one base, or stopping, case.*

- A base case does not execute a recursive call
 - stops the recursion
- Each successive call to itself must be a "smaller version of itself"
 - an argument that describes a smaller problem
 - a base case is eventually reached

Key Components of a Recursive Algorithm Design

1. What is a smaller ***identical*** problem(s)?

□ Decomposition

2. How are the answers to smaller problems combined to form the answer to the larger problem?

□ Composition

3. Which is the smallest problem that can be solved easily (without further decomposition)?

□ Base/stopping case

Factorial ($N!$)

- $N! = (N-1)! * N$ [for $N > 1$]
- $1! = 1$
- $3!$
 - $= 2! * 3$
 - $= (1! * 2) * 3$
 - $= 1 * 2 * 3$
- Recursive design:
 - Decomposition: $(N-1)!$
 - Composition: $* N$
 - Base case: $1!$

factorial Method

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition)
        fact = factorial(n - 1) * n; // composition
    else // base case
        fact = 1;

    return fact;
}
```

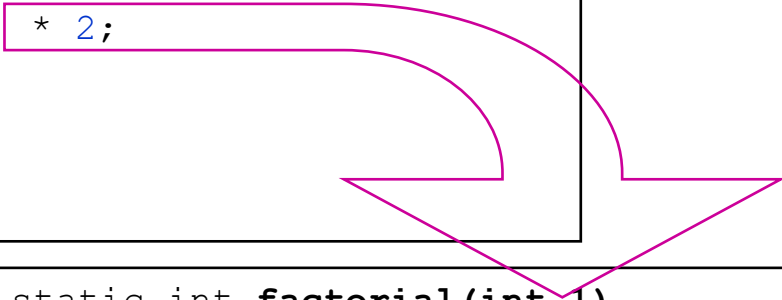
```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = factorial(1) * 2;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = factorial(1) * 2;
    else
        fact = 1;
    return fact;
}
```

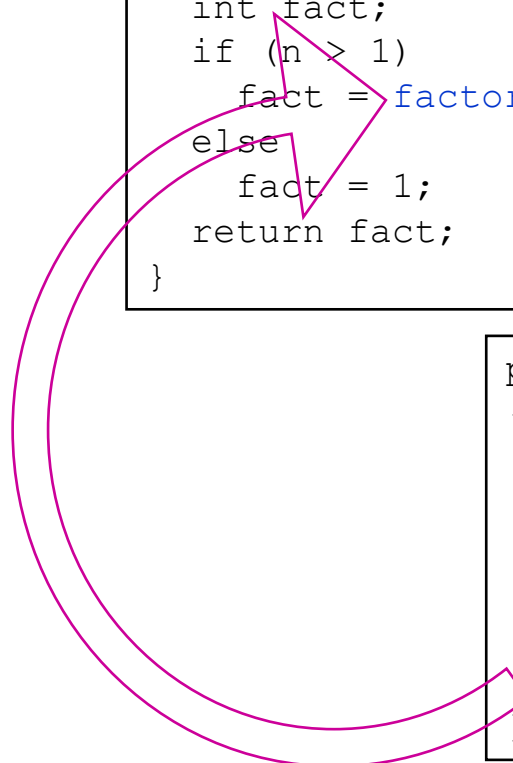


```
public static int factorial(int 1)
{
    int fact;
    if (n > 1)
        fact = factorial(n - 1) * n;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = factorial(1) * 2;
    else
        fact = 1;
    return fact;
}
```

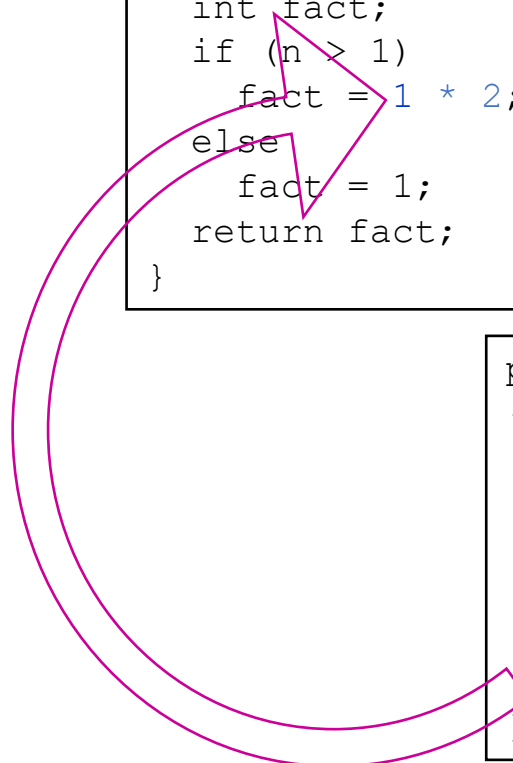
```
public static int factorial(int 1)
{
    int fact;
    if (n > 1)
        fact = factorial(n - 1) * n;
    else
        fact = 1;
    return 1;
}
```



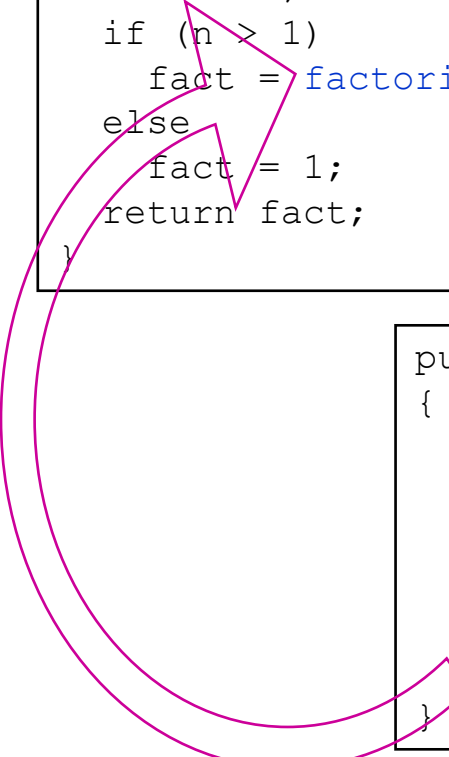
```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = 1 * 2;
    else
        fact = 1;
    return fact;
}
```

```
public static int factorial(int 1)
{
    int fact;
    if (n > 1)
        fact = factorial(n - 1) * n;
    else
        fact = 1;
    return 1;
}
```

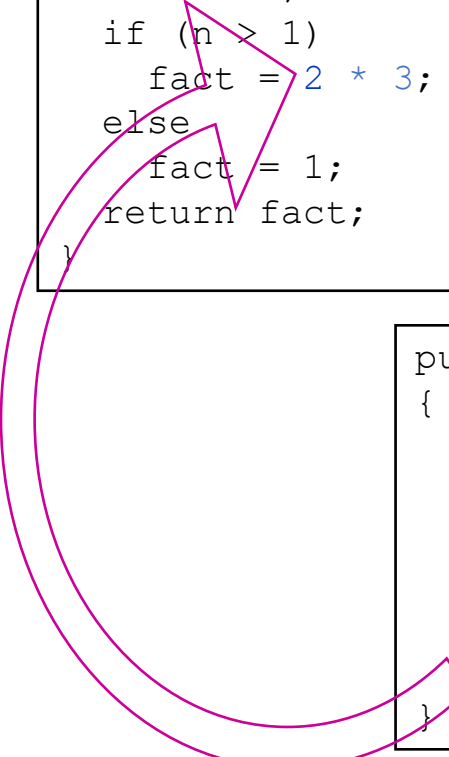


```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = factorial(2) * 3;
    else
        fact = 1;
    return fact;
}
```

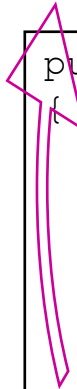


```
public static int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = 1 * 2;
    else
        fact = 1;
    return 2;
}
```

```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = 2 * 3;
    else
        fact = 1;
    return fact;
}
```



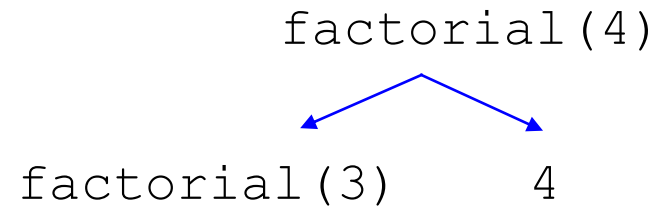
```
public static int factorial(int 2)
{
    int fact;
    if (n > 1)
        fact = 1 * 2;
    else
        fact = 1;
    return 2;
}
```

```
public static int factorial(int 3)
{
    int fact;
    if (n > 1)
        fact = 2 * 3;
    else
        fact = 1;
    return 6;
}
```

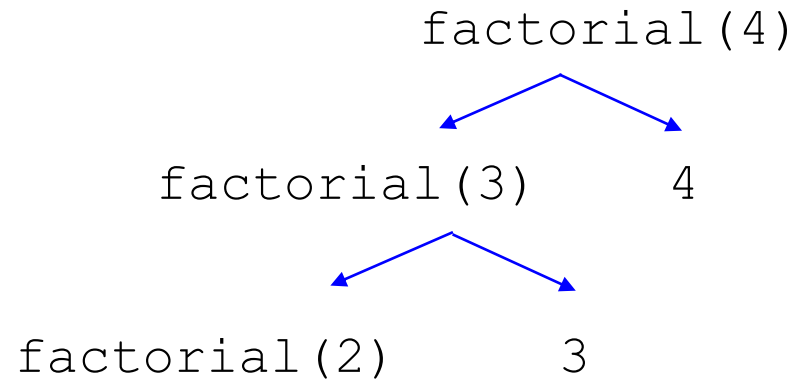
Execution Trace (decomposition)

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition)
        fact = factorial(n - 1) * n; (composition)
    else // base case
        fact = 1;
    return fact;
}
```



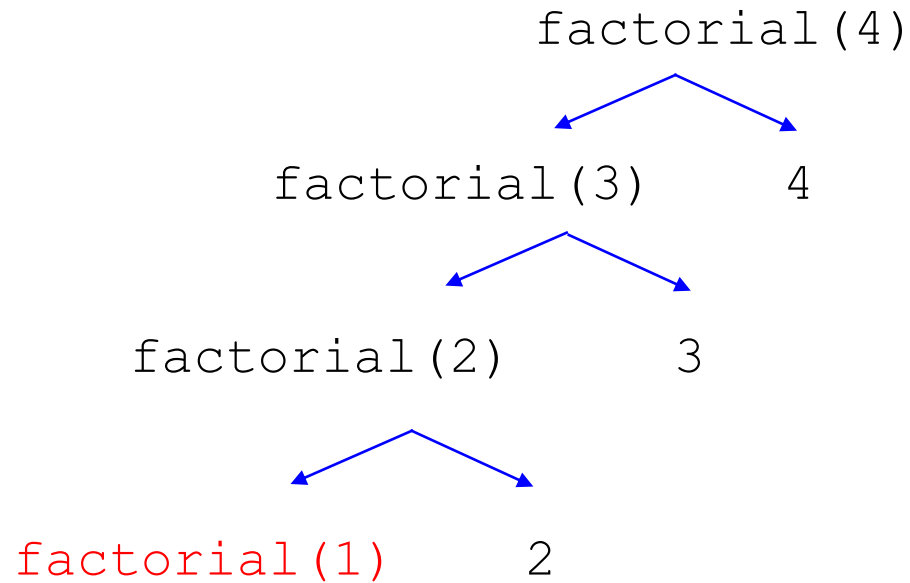
Execution Trace (decomposition)

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition)
        fact = factorial(n - 1) * n; (composition)
    else // base case
        fact = 1;
    return fact;
}
```



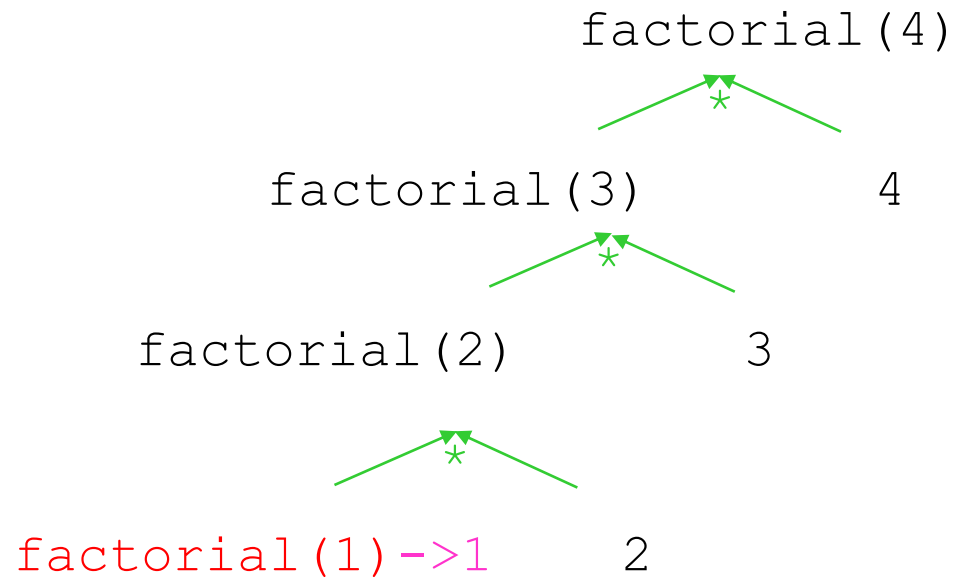
Execution Trace (decomposition)

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition)
        fact = factorial(n - 1) * n; (composition)
    else // base case
        fact = 1;
    return fact;
}
```



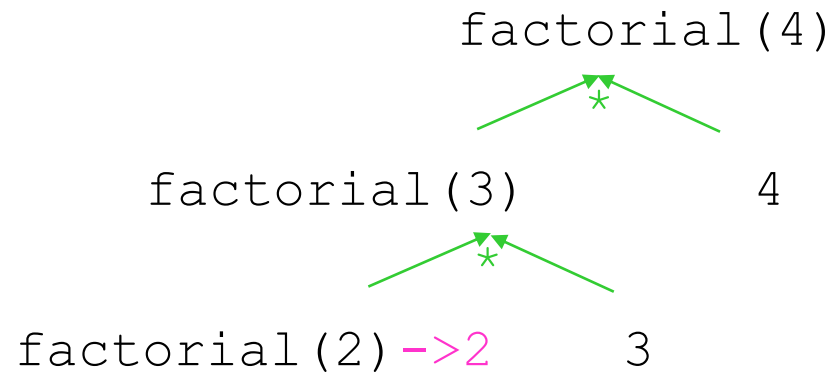
Execution Trace (composition)

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition)
        fact = factorial(n - 1) * n; (composition)
    else // base case
        fact = 1;
    return fact;
}
```



Execution Trace (composition)


```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition)
        fact = factorial(n - 1) * n; (composition)
    else // base case
        fact = 1;
    return fact;
}
```



Execution Trace (composition)

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition)
        fact = factorial(n - 1) * n; (composition)
    else // base case
        fact = 1;
    return fact;
}
```

factorial(4)
 *
factorial(3) → 6 4

A diagram illustrating the composition step of the factorial function. It shows 'factorial(4)' at the top, with a green asterisk '*' below it. Two green lines extend downwards from the asterisk to 'factorial(3)' on the left and '4' on the right. A pink arrow points from 'factorial(3)' to the pink number '6'.

Execution Trace (composition)

```
public static int factorial(int n)
{
    int fact;
    if (n > 1) // recursive case (decomposition)
        fact = factorial(n - 1) * n; (composition)
    else // base case
        fact = 1;
    return fact;
}
```

factorial(4) -> 24

Improved factorial Method

```
public static int factorial(int n)
{
    int fact=1;    // base case value

    if (n > 1)    // recursive case (decomposition)
        fact = factorial(n - 1) * n; // composition
    // else do nothing; base case

    return fact;
}
```

Recursion – Example 1

Give a recursive algorithm to compute the product of two positive integers, m and n , using only addition and subtraction.

```
def prod(m, n) :  
    if m == 0 :  
        return 0  
    else :  
        return n + prod(m - 1, n)
```

Recursion – Example 2

Describe a recursive algorithm for finding the maximum element in an array A of n elements.

```
def max(A) :  
    if length(A) = 1:    // one element list  
        return A[0]      // the max value is the only  
value  
    m = max(A[1:]) //recurs over the 2nd element  
    if m > A[0]:        //      and beyond  
        return m  
    else:  
        return A[0]
```

Recursion – Example 3

Given an array A of length n containing values in increasing order, write a recursive algorithm to find the first repeated pair of values if such a pair exists.

```
def fRepeat(A):  
    if length(A) < 2: // list too small to find a pair?  
        return False // fail  
    if A[0] = A[1] // first two elements match?  
        return A[0] // return value of first element  
    else: // otherwise  
        // recur over the second element onward  
        fRepeat(A[1:])
```

Tail recursive!

Selection Sort

- A sorting algorithm rearranges the elements of a collection so that they are stored in sorted order.
- Selection sort sorts an array by repeatedly finding the smallest element of the unsorted tail region and moving it to the front.
- Slow when run on large data sets.
- Example: sorting an array of integers

11	9	17	5	12
----	---	----	---	----

Sorting an Array of Integers

1. Find the smallest and swap it with the first element

5	9	17	11	12
---	---	----	----	----

1. Find the next smallest. It is already in the correct place

5	9	17	11	12
---	---	----	----	----

1. Find the next smallest and swap it with first element of unsorted portion

5	9	11	17	12
---	---	----	----	----

2. Repeat

5	9	11	12	17
---	---	----	----	----

1. When the unsorted portion is of length 1, we are done

5	9	11	12	17
---	---	----	----	----

Selection Sort

In selection sort, pick the smallest element and swap it with the first one. Pick the smallest element of the remaining ones and swap it with the next one, and so on.

section_1/SelectionSorter.java

```
1  /**
2     The sort method of this class sorts an array, using the selection
3     sort algorithm.
4  */
5  public class SelectionSorter
6  {
7      /**
8         Sorts an array, using selection sort.
9         @param a the array to sort
10     */
11     public static void sort(int[] a)
12     {
13         for (int i = 0; i < a.length - 1; i++)
14         {
15             int minPos = minimumPosition(a, i);
16             ArrayUtil.swap(a, minPos, i);
17         }
18     }
19 }
```

Continued

section_1/SelectionSorter.java

```
20    /**
21     * Finds the smallest element in a tail range of the array.
22     * @param a the array to sort
23     * @param from the first position in a to compare
24     * @return the position of the smallest element in the
25     *         range a[from] . . . a[a.length - 1]
26     */
27    private static int minimumPosition(int[] a, int from)
28    {
29        int minPos = from;
30        for (int i = from + 1; i < a.length; i++)
31        {
32            if (a[i] < a[minPos]) { minPos = i; }
33        }
34        return minPos;
35    }
36 }
```

section_1/SelectionSortDemo.java

```
1  import java.util.Arrays;
2
3  /**
4   * This program demonstrates the selection sort algorithm by
5   * sorting an array that is filled with random numbers.
6   */
7  public class SelectionSortDemo
8  {
9      public static void main(String[] args)
10     {
11         int[] a = ArrayUtil.randomIntArray(20, 100);
12         System.out.println(Arrays.toString(a));
13
14         SelectionSorter.sort(a);
15
16         System.out.println(Arrays.toString(a));
17     }
18 }
19
20
```

Typical Program Run:

```
[65, 46, 14, 52, 38, 2, 96, 39, 14, 33, 13, 4, 24, 99, 89, 77, 73, 87, 36, 81]
[2, 4, 13, 14, 14, 24, 33, 36, 38, 39, 46, 52, 65, 73, 77, 81, 87, 89, 96, 99]
```

Self Check 14.1

Why do we need the `temp` variable in the `swap` method?
What would happen if you simply assigned `a[i]` to `a[j]`
and `a[j]` to `a[i]`?

Answer: Dropping the `temp` variable would not work.
Then `a[i]` and `a[j]` would end up being the same
value.

Self Check 14.2

What steps does the selection sort algorithm go through to sort the sequence 6 5 4 3 2 1?

Answer:

1	5	4	3	2	6
---	---	---	---	---	---

1	2	4	3	5	6
---	---	---	---	---	---

1	2	3	4	5	6
---	---	---	---	---	---

Self Check 14.3

How can you change the selection sort algorithm so that it sorts the elements in descending order (that is, with the largest element at the beginning of the array)?

Answer: In each step, find the maximum of the remaining elements and swap it with the current element (or see Self Check 4).

Self Check 14.4

Suppose we modified the selection sort algorithm to start at the end of the array, working toward the beginning. In each step, the current position is swapped with the minimum. What is the result of this modification?

Answer: The modified algorithm sorts the array in descending order.

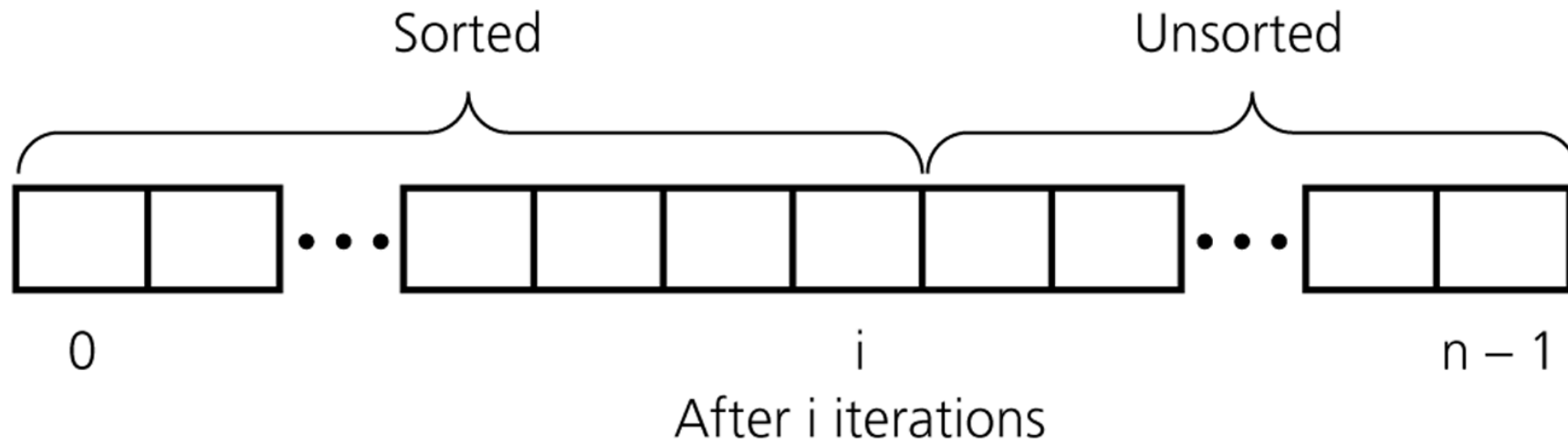
Insertion Sort

- while some elements unsorted:
 - Using linear search, find the location in the sorted portion where the 1st element of the unsorted portion should be inserted
 - Move all the elements after the insertion location up one position to make space for the new element

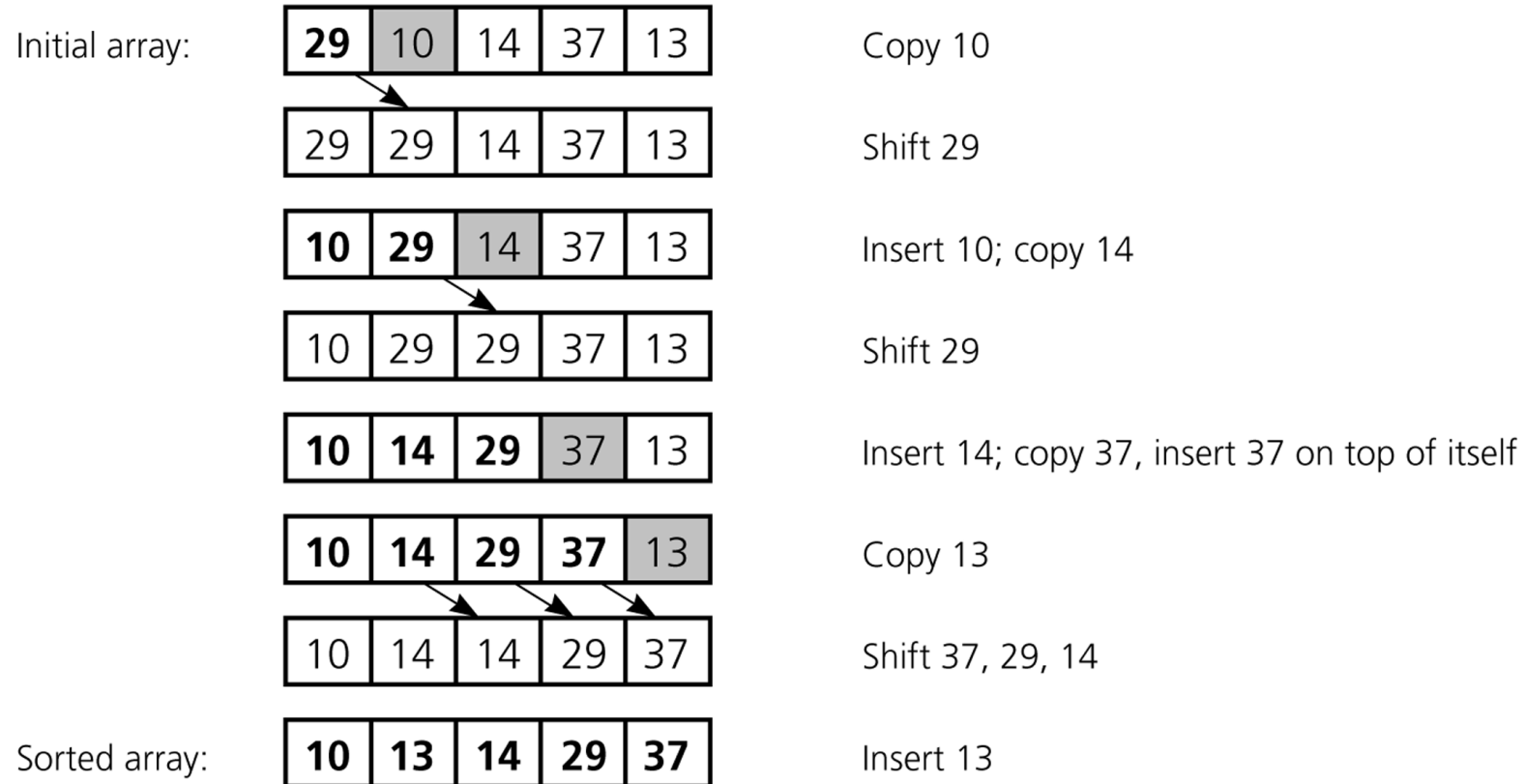
45

38	45	60	66	79	47	13	74	36	21	94	22	57	16	29	81
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

the fourth iteration of this loop is shown here



An insertion sort partitions the array into two regions



An insertion sort of an array of five integers

Insertion Sort Algorithm

```
public void insertionSort(Comparable[] arr) {  
    for (int i = 1; i < arr.length; ++i) {  
        Comparable temp = arr[i];  
        int pos = i;  
        // Shuffle up all sorted items > arr[i]  
        while (pos > 0 &&  
            arr[pos-1].compareTo(temp) > 0) {  
            arr[pos] = arr[pos-1];  
            pos--;  
        } // end while  
        // Insert the current item  
        arr[pos] = temp;  
    }  
}
```

Insertion Sort Analysis

```
public void insertionSort(Comparable[] arr) {  
    for (int i = 1; i < arr.length; ++i) { ← outer loop  
        Comparable temp = arr[i]; ← outer times  
        int pos = i; ← outer times  
        // Shuffle up all sorted items > arr[i]  
        while (pos > 0 &&  
            arr[pos-1].compareTo(temp) > 0) {  
            arr[pos] = arr[pos-1];  
            pos--; ← inner times  
        } // end while  
        // Insert the current item  
        arr[pos] = temp; ← outer times  
    }  
}
```

The diagram illustrates the execution flow of the Insertion Sort algorithm. It features four yellow boxes with black text: 'outer loop', 'outer times', 'inner loop', and 'inner times'. Arrows indicate the flow of execution: an arrow points from the 'outer loop' box to the 'for' loop's opening brace; an arrow points from the 'outer times' box to the assignment 'Comparable temp = arr[i];'; another arrow points from the 'outer times' box to the assignment 'int pos = i;'; a third arrow points from the 'inner loop' box to the 'while' loop's opening brace; and a fourth arrow points from the 'inner times' box to the decrement 'pos--;'. A long curved arrow also points from the 'outer times' box to the assignment 'arr[pos] = temp;'.

Insertion Sort: Number of Comparisons

# of Sorted Elements	Best case	Worst case
0	0	0
1	1	1
2	1	2
...
n-1	1	n-1
	<hr/>	<hr/>
	n-1	$n(n-1)/2$
	<hr/>	<hr/>

Remark: we only count comparisons of elements in the array.

Insertion Sort: Cost Function

- 1 operation to initialize the outer loop
- The outer loop is evaluated $n-1$ times
 - 5 instructions (including outer loop comparison and increment)
 - Total cost of the outer loop: $5(n-1)$
- How many times the inner loop is evaluated is affected by the state of the array to be sorted
- Best case: the array is already completely sorted so no “shifting” of array elements is required.
 - We only test the condition of the inner loop once (2 operations = 1 comparison + 1 element comparison), and the body is never executed
 - Requires $2(n-1)$ operations.

Insertion Sort: Cost Function

- Worst case: the array is sorted in reverse order (so each item has to be moved to the front of the array)
 - In the i -th iteration of the outer loop, the inner loop will perform $4i+1$ operations
 - Therefore, the total cost of the inner loop will be $2n(n-1)+n-1$
- Time cost:
 - Best case: $7(n-1)$
 - Worst case: $5(n-1)+2n(n-1)+n-1$
- What about the number of moves?
 - Best case: $2(n-1)$ moves
 - Worst case: $2(n-1)+n(n-1)/2$

Insertion Sort: Average Case

- Is it closer to the best case (n comparisons)?
- The worst case ($n * (n-1) / 2$) comparisons?
- It turns out that when random data is sorted, insertion sort is usually closer to the worst case
 - Around $n * (n-1) / 4$ comparisons
 - Calculating the average number of comparisons more exactly would require us to state assumptions about what the “average” input data set looked like
 - This would, for example, necessitate discussion of how items were distributed over the array
- Exact calculation of the number of operations required to perform even simple algorithms can be challenging
(for instance, assume that each initial order of elements has the same probability to occur)

Sort- Problem Analysis Example

- You are given a set of n real numbers and another real number x . Describe an $O(n \log n)$ time algorithm that determines whether or not there exists 2 elements in S whose sum is exactly x .

Solution

- First sort the array ($n \log n$)

```
public static boolean test(int[] a, int val)
{
    Arrays.sort(a);

    int i = 0;           // index of first element.
    int j = a.length - 1; // index of last element.

    while(i < j)
    {
        // check if the sum of elements at index i and j equals val, if yes we are done
        if(a[i] + a[j] == val)
            return true;
        // else if sum is more than val, decrease the sum.
        else if(a[i] + a[j] > val)
            j--;
        // else if sum is less than val, increase the sum.
        else
            i++;
    }
    // failed to find any such pair..return false.
    return false;
}
```