Lecture2:
Performance of Algorithms. Sorting revisited

Review: Asymptotic Complexity Analysis

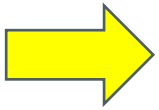
Measuring Runtime

Sorting in Java

Comparable and Comparator

Generics: classes and methods

Performance of Algorithms. Sorting revisited



Review: Asymptotic Complexity Analysis

Measuring Runtime

Sorting in Java

Comparable and Comparator

Generics: classes and methods

Evaluating Performance of Algorithms

- Empirical:
  - Measure running time
  - Needs a complete implementation of the algorithm
  - Can be an anecdotical evidence (for a particular value of input)
- Analytical
  - Assesses correctness and efficiency of a solution in its design phase
  - Predict performance
  - Compare algorithms
  - Provide guarantees
  - Time complexity, Asymptotic analysis

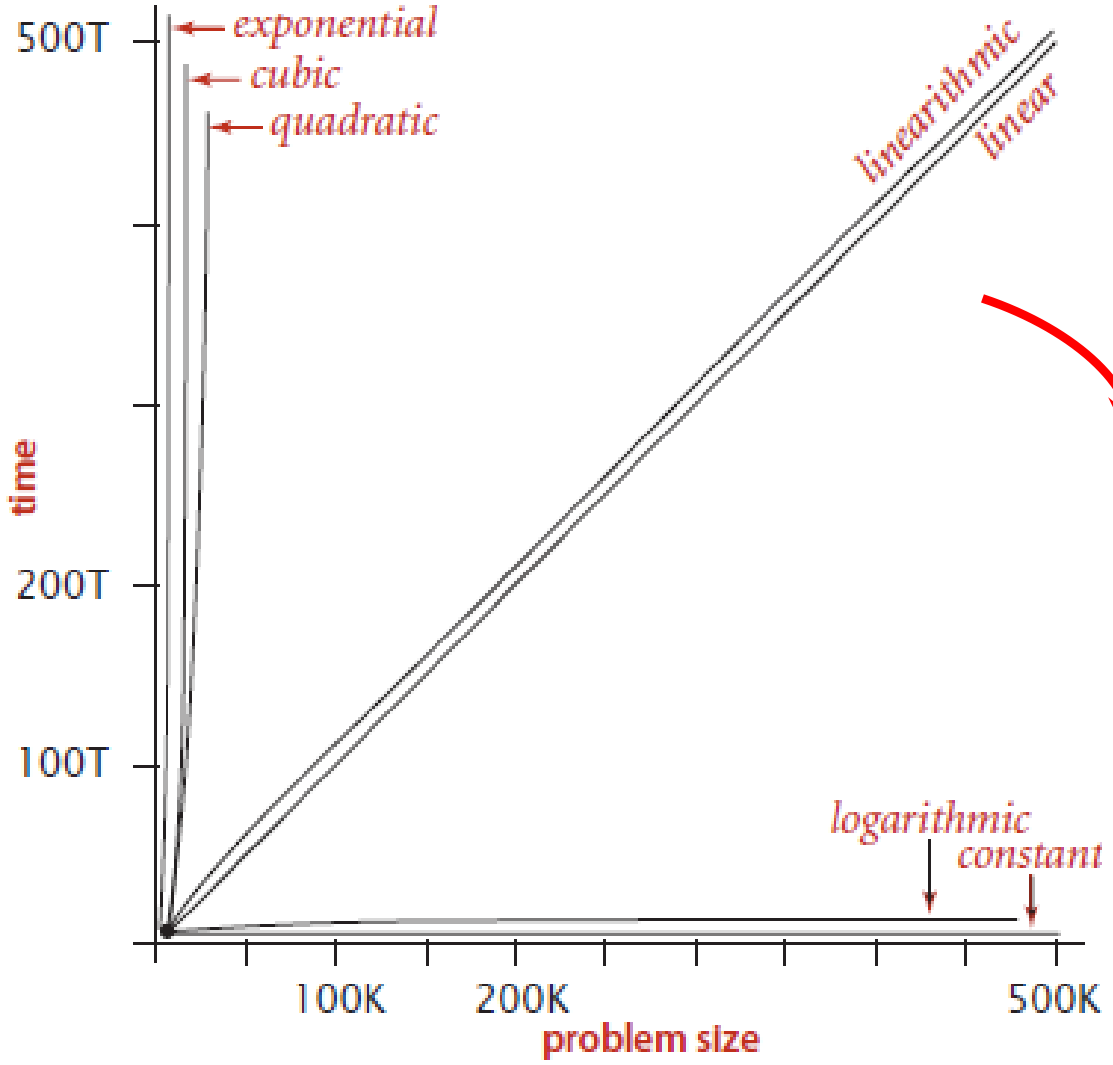
Asymptotic Complexity Analysis

- Running time depends on input size:
  - ***T(n)*** refers to the time it takes for the algorithm to process an input of size n. It is an ***increasing*** function.
  - The exact values of the running time T(n) are influenced by system characteristics like processor speed and memory
  - ***Rate of growth*** (or ***Order of growth***) refers to ***how fast increases*** the running time function T(n) as n increases
    - This is ***independent of the system's characteristics*** and depends only on the algorithm !
- The goal of algorithm analysis is to evaluate and compare different algorithms based on their order of growth

Mathematical models for running time

- ***Starting assumption: Basic operations take a constant time*** (the values of the constants depend on computer and platform)
- ***Approximate running time by counting total number of basic operations***. The number of basic operations may depend on input size => Approximates running time as a function of input size
- ***If we have the running time as a function of input size***, we can approximate further:
  - ***keep only the most significant term (tilda approximation)***
  - ***ignore its coefficient.*** This gives the **order of growth** of the running time function
  - Example: $T(n) = 5*n^2+10*n +7*\log(n)+28$ order of growth is n^2

Common order of growth classification



The smaller the growth rate, the better!

Using Approximations for Running Time

- In practice, finding the running time function T(n) can be difficult for some algorithms
- Instead of finding T(n), we try to find Upper bounds and Lower bounds for T(n)
- **Upper bound**: find a function such that:
  - running time is not worse than <upper bound function> for all *big* input sizes
  - order of growth is smaller than <upper bound function>
- **Lower bound**: find a function such that:
  - running time is not better than <lower bound function> for all *big* input sizes
  - order of growth is bigger than <lower bound function>
- **Tight bound**: find a function that is both upper and lower bound
  - this function is the order of growth

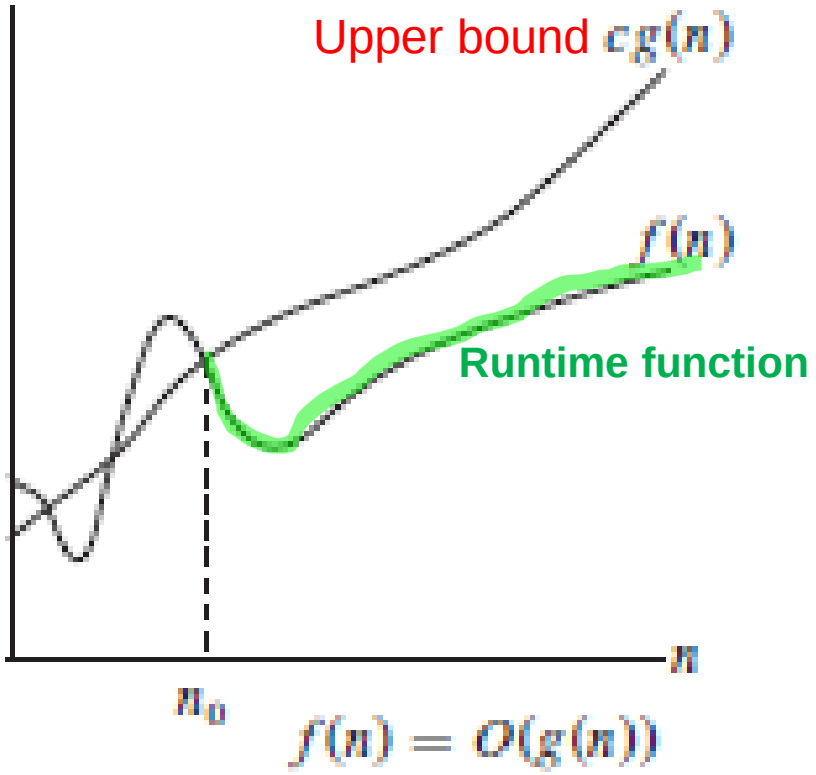
Asymptotic Notations

- O: Big-Oh = asymptotic upper bound
- Ω: Big-Omega = asymptotic lower bound
- Θ: Theta = asymptotically tight bound

[CLRS] – chap 3

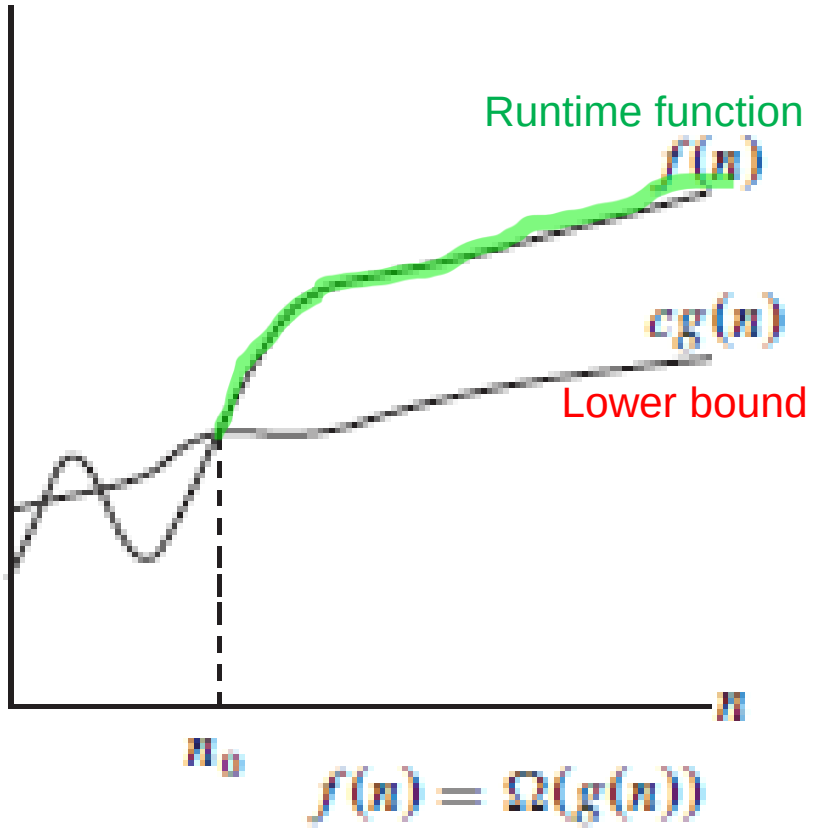
Big O

- O (g(n)) is the ***set*** of ***all*** functions with a smaller or same order of growth as g(n), within a constant multiple
- If f(n) ∈ O(g(n)) (f(n) *is in* O(g(n)), it means that **g(n)** is an asymptotic ***upper bound*** of f(n)
  - Intuitively, it is like f(n) ≤ g(n)
  - We write f(n)=O(g(n))



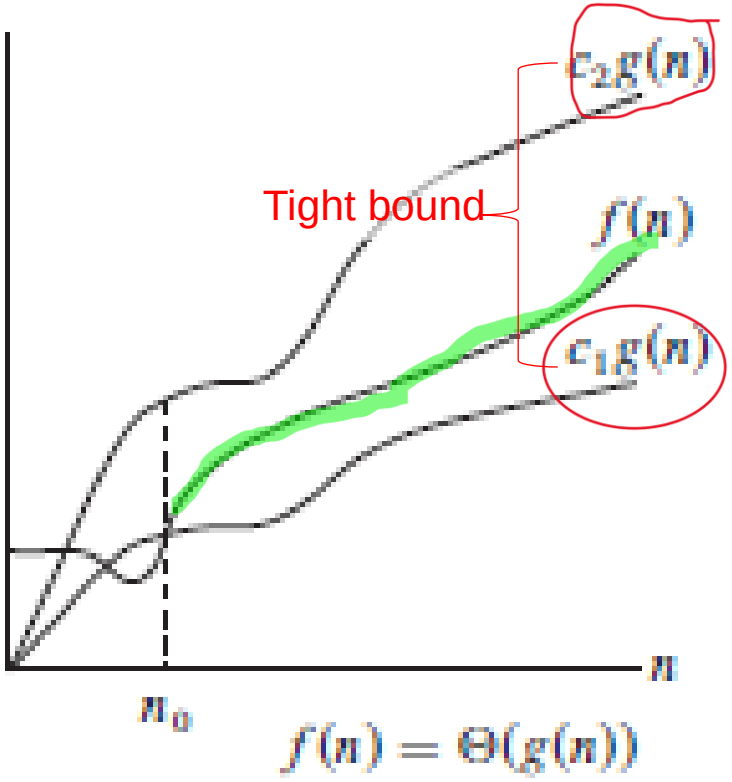
Big Ω

- Ω (g(n)) is the set of ***all*** functions with a larger or same order of growth as g(n), within a constant multiple
- f(n) ∈ Ω(g(n)) means **g(n)** is an asymptotic ***lower bound*** of f(n)
  - Intuitively, it is like g(n) ≤ f(n)



Theta (Θ)

- Informally, Θ (g(n)) is the set of all functions with the same order of growth as g(n), within a constant multiple
- f(n) ∈ Θ(g(n)) means **g(n)** is an asymptotically ***tight bound*** of f(n)
  - Intuitively, it is like f(n) = g(n)



Running Time Estimation

- In practice, estimating the running time T(n) means finding a function f(n), such that T(n) in O(f(n)) or T(n) in Θ(f(n))
- If we prove that T(n) in O(f(n)) we just guarantee that T(n) “is not worse than f(n)”
  - Attention to overapproximations ! We can say about most algorithms that they are in $O(2^n)$, true but useless asessment
- If we prove that T(n) is in Θ(f(n)) we actually determine the order of growth of the running time.

Running Time Estimation

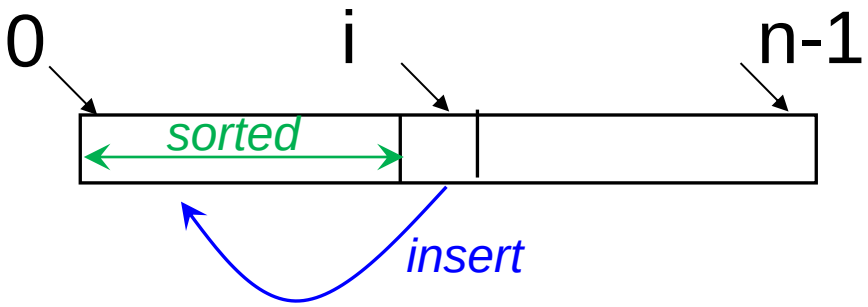
- Approximating running time by counting total number of basic operations can be difficult in more complex situations:
  - **Nested loops:** when inner loop is executed many times, each time with a different complexity: need to solve **Summations**
  - **Recursive algorithms**: need to solve **Recurrences**

Example: running time function for nested loops

Example: **InsertionSort:**

The size of the input n = the length of array
T(n) = running time for an array of length n

```
class SimpleInsertionSorter  {

public void sort(int[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            for (int j = i; j > 0; j--) {
                if (a[j] < a[j - 1]) exch(a, j, j - 1);
                else break;
            }
        }
    }
}
```



In order to estimate running time, we **count how many times** this basic statement is executed

Example: running time function for nested loops (cont.)

InsertionSort:

```
public void sort(int[] a) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        for (int j = i; j > 0; j--) {
            if (a[j] < a[j - 1]) exch(a, j, j - 1);
            else break;
        }
    }
}
```

The size of the input n = the length of array

T(n) = running time for an array of length n

The best-case time: when the array is initially sorted: $T_{Best}(n) = \sum_{i=0}^{n-1} 1 = n$

The ***best-case*** running time is a ***linear function*** of n.

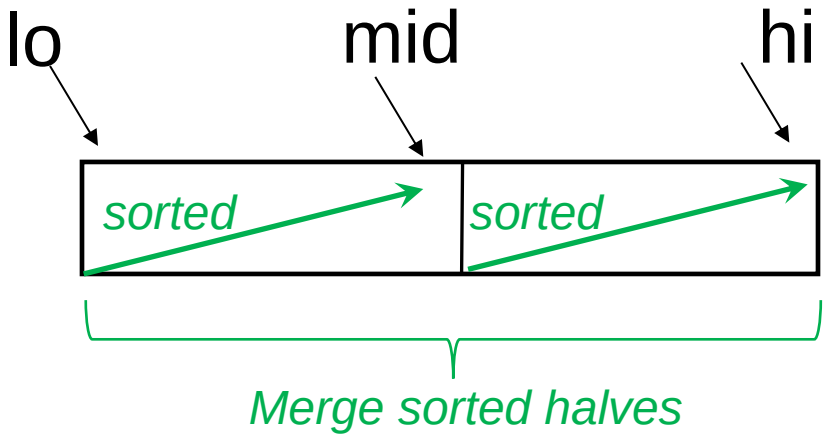
The worst-case: when the array is reversed: $T_{Worst}(n) = \sum_{i=0}^{n-1} i = n * (n-1)/2$

The **worst-case** running time is a ***quadratic function*** of n

InsertionSort is $O(n^2)$

Example: running time function for recursive functions

```
class SimpleMergeSorter  {

   private int[] a;
   private int[] aux;

    public void sort(int[] a) {
       this.a = a;
       this.aux = new int[a.length];
       mergeSort(0, a.length - 1);
    }

    private void mergeSort(int lo, int hi) {
       if (hi <= lo) return;
       int mid = lo + (hi - lo) / 2;

       mergeSort(lo, mid);
       mergeSort(mid + 1, hi);

       merge(lo, mid, hi);
    }
```



The size of the input n = the length of array
n = hi - lo
T(n) = running time for an array of length n

Function mergeSort: contains 2 recursive calls and a call to function merge

Sequential operation: merge()

```
private void merge(int lo, int mid, int hi) {
   //lower half of a is sorted and upper half of a is sorted
   //copy a to aux array
   for (int k = lo; k <= hi; k++) {
      aux[k] = a[k];
   }

   //merge lower half and upper half of aux back into a
   int i = lo; //current position in lower half
   int j = mid + 1; // current position in upper half
   for (int k = lo; k <= hi; k++) {
      if (i > mid) a[k] = aux[j++]; // lower half finished
      else if (j > hi) a[k] = aux[i++]; // upper half finished
      else if (aux[j] < aux[i]) a[k] = aux[j++];
      else a[k] = aux[i++];
   }
}
```

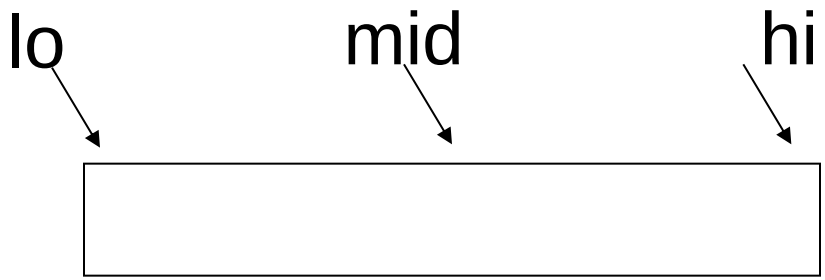
The size of the input n = the length of array
n = hi - lo

T(n) = running time for an array of length n

k is doing exactly n iterations with constant time =>
running time of merge is Θ(n)

Example: Recurrence for merge-sort

```
MERGE-SORT(A[lo..hi])
   if lo < hi
        mid = (lo+hi)/2
        MERGE-SORT(A[lo..mid])
        MERGE-SORT(A[mid+1..hi])
        MERGE(A[lo..hi],mid)
```



Size of input n = length of array = hi – lo

T(n) = runtime of MergeSort for array of length n

Runtime of Merge is Θ(n)

$$T(n) = \begin{cases} \Theta(1), & n=1 \\ 2*T(n/2) + \Theta(n), & n>1 \end{cases}$$

In case of recursive algorithms, we get a recurrence relationship on the run time function

Solving Recurrences

- The recurrence has to be solved in order to find out T(n) as a function of n
- General methods for solving recurrences:
  - **Substitution Method** (a problem of pure math)
  - **Recursion-tree Method**

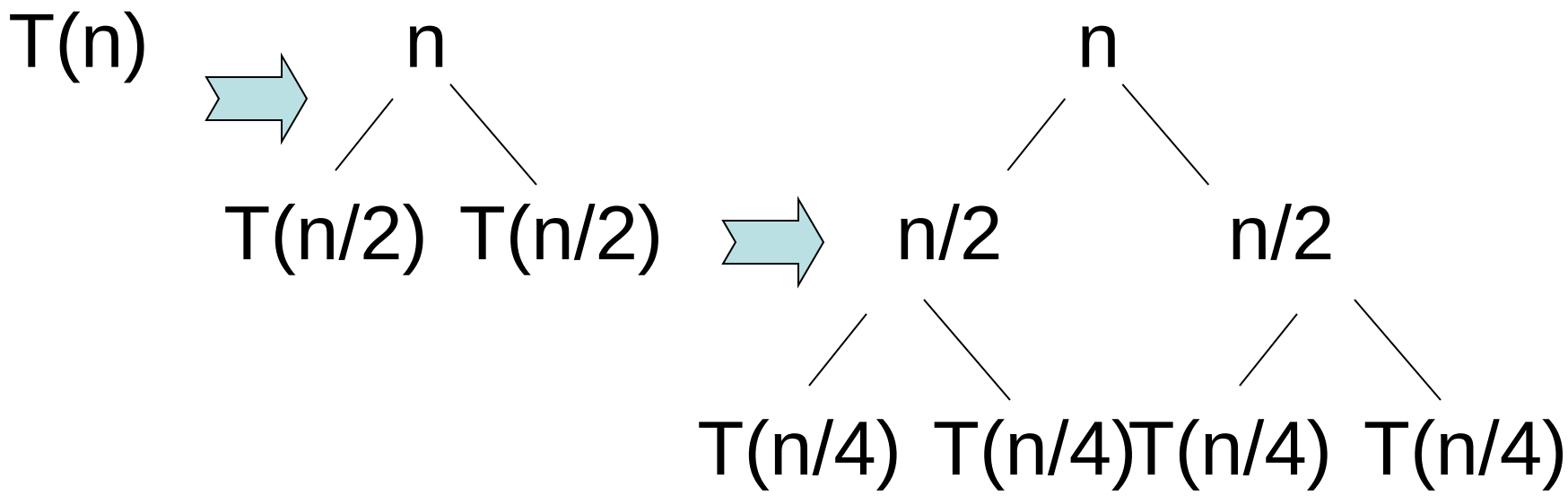
The Recursion Tree Method

- The recursion tree method for solving recurrences:
  - converts the recurrence into a tree of the recursive function calls.
  - Each node has a cost, representing the workload of the corresponding call (without the recursive calls done there). The total workload results by adding the workloads of all nodes. It uses techniques for bounding summations.
- Example: applying the recursion tree method for solving the MergeSort recurrence

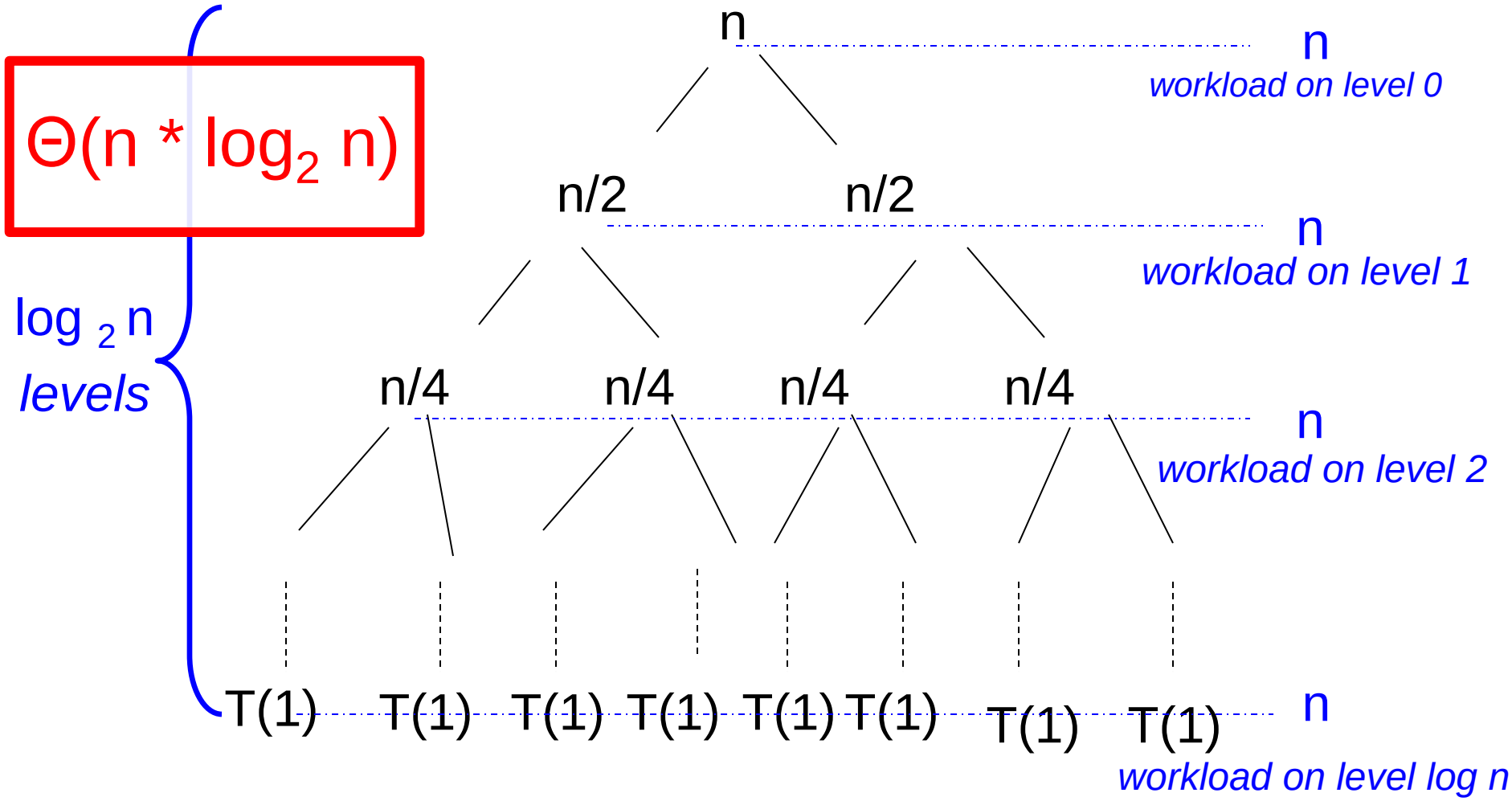
Recursion tree: T(n)=2 *T(n/2)+n

The recursion tree has to be expanded until it reaches its leafs.
To compute the total runtime we have to sum up the costs of all the nodes:

- Find out ***how many levels*** there are
- Find out the ***workload on each level***



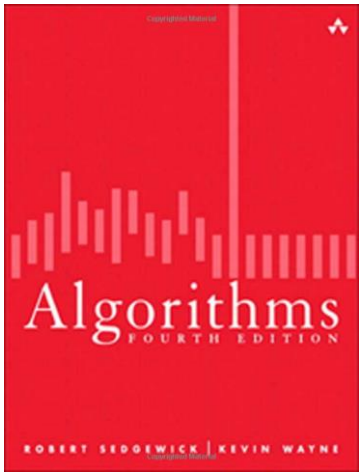
Recursion tree: T(n)=2 *T(n/2)+n



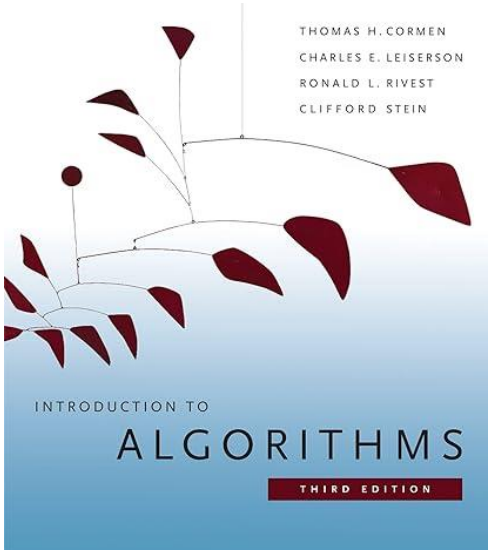
Summary on Asymptotic Analysis

- Running time depends on the size of the input (the problem size)
  - T(n): the time taken on input with size n
- Mathematical models approximate T(n), the running time of an algorithm, by counting the total number of basic operations performed
- Computing T(n) can be mathematically challenging:
  - Nested loops require to solve Summations or Discrete Sums
  - Recursive algorithms require to solve Recurrences
- it is the ***rate of growth***, or ***order of growth of T(n)*** that really matters because it is intrinsic to the algorithm
- We estimate asymptotic complexity with: *Upper Bound* O (Big-O), *Lower Bound* Ω (Big-Omega) and *Tight Bound* Θ (Big-Theta).

Additional reading on complexity analysis



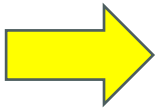
- Main text used for these slides:
  - [Sedgewick] – chap 1.4 (Analysis of algorithms)



- Additional reading and more details:
  - [CLRS] – chap 3 (Growth of functions), chap 4 (Recurrences, Master Theorem)

Performance of Algorithms. Sorting revisited

Review: Asymptotic Complexity Analysis



Measuring Runtime

Sorting in Java

Comparable and Comparator

Generics: classes and methods

Measuring time

- *Wall clock time* *vs* *CPU time*
  - CPU Time: The time the CPU spends actually executing the code block
    - For ***sequential*** algorithms, ideally we want to measure the *CPU time*
  - Wall Clock Time: The real-world time that elapses from the moment the execution starts until it finishes.
    - This includes any delays caused by multitasking, where multiple processes or threads share the CPU through timesharing.
- In Java:
  - **Wall Clock Time** can be easily measured using **`System.nanoTime()`** or **`System.currentTimeMillis()`.** These methods provide ***reasonable profiling*** for most situations
  - CPU Time is more difficult to measure directly (use `ThreadMXBean.getThreadCPUTime()`

Measuring wall-clock time

```
public class Stopwatch {
    private long startTime;

    public Stopwatch(){
        startTime = System.nanoTime();
    }

    // Restart the stopwatch
    public void start() {
        startTime = System.nanoTime();
    }

    // Get the elapsed time in seconds
    public double getElapsedTime() {
        long endTime = System.nanoTime();
        return (endTime - startTime) / 1000000000.0;  // nanoseconds to seconds
    }

}
```

Measuring time

- Measuring runtime using *a Stopwatch*

```
Stopwatch timer = new Stopwatch();

algo(); // algorithm to be measured

System.out.println("elapsed time = " + timer.getElapsedTime());
```

Empirical analysis

- How to methodically evaluate the performance of an algorithm:
  - Run algorithm *for various input sizes* and measure running *time T(N) as a function of input size*
  - Repeat measurements and consider the average time values in order to account for measurement errors
- *Input size* can have different meanings in different contexts:
  - Input size = number of elements in an array: sorting, searching
  - Input size = value of a variable: value of n for factorial n!

Example: empirical analysis for sorting algorithms

- We want to compare: our own implementations for various sorting algorithms (InsertionSort, MergeSort, etc) and Arrays.sort from Java's standard library
- Measure time T(N) for various input sizes N (number of elements), array with ***random*** initial values
- Use the same values for all evaluated algorithms

N	1K	10K	50K	100K	1M
InsertionS					
MergeS					
Arrays.sort					

Example: empirical analysis for sorting algorithms

A basic performance evaluation harness:

```
for each algorithm A
    for each data input  INP
                measure time for A(INP)
```

In order to facilitate evaluation, make all alternatives implement the same interface

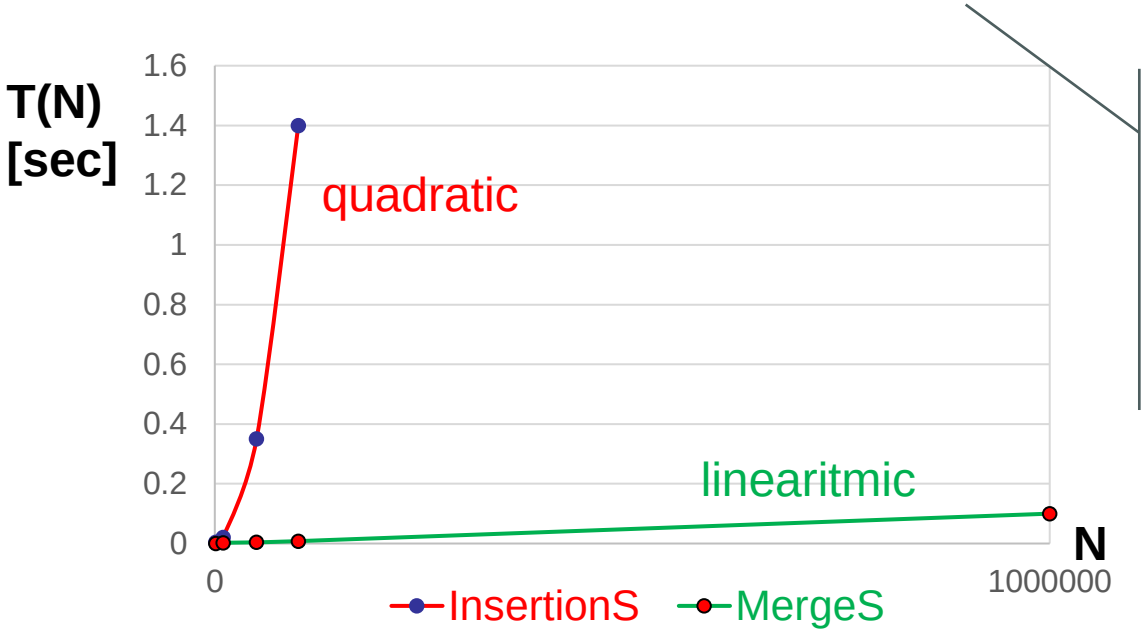
```
interface SimpleSorter {
    String getName();
    void sort(int[] a);
}
```

```
public class SimpleSortersExperiments {

    public static void main(String[] args) {

        // Input file names
        String[] inputFiles = {"randomIntegers_10.txt", "randomIntegers_10K.txt",
                "randomIntegers_50K.txt", "randomIntegers_100K.txt", "randomIntegers_1M.txt"};
        // Sorting algorithms
        SimpleSorter[] sorters = {new JavaArraysSorter(),  new SimpleMergeSorter(),
                new SimpleInsertionSorter()};

        for (String fileName : inputFiles) {
            System.out.println("Processing file: " + fileName);
            for (SimpleSorter sorter : sorters) {
                int[] a = FileInputHelper.readIntegers(fileName);  //read original array again!
                if (a == null) continue;  // Skip if file reading failed

                Stopwatch stopwatch = new Stopwatch();
                sorter.sort(a);
                System.out.println("Elapsed time for " + sorter.getName() + " on file " + fileName +
                        " with n=" + a.length + " is " + stopwatch.getElapsedTime() + " seconds");
            }
        }
    }
}
```

Example: empirical analysis for sorting algorithms

- Measured time T(N) for various input sizes N (number of elements), array with ***random*** initial values

N	1K	10K	50K	100K	1M
InsertionS	0.004	0.021	0.35	1.4	
MergeS	0.0003	0.0025	0.004	0.008	0.1
Arrays.sort	0.0003	0.004	0.003	0.007	0.07



On another computer we will measure other values in the table, BUT the shape of the graphs will be the same!

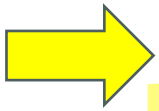
```
class SimpleInsertionSorter implements SimpleSorter {
    private final String name = "InsertionSort";

    public String getName() {  return name;   }

    private void exch(int[] a, int i, int j) {
        int temp = a[i];  a[i] = a[j];  a[j] = temp;
    }

    public void sort(int[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            for (int j = i; j > 0; j--) {
                if (a[j] < a[j - 1]) exch(a, j, j - 1);
                else break;
            }
        }
    }
}
```

```
class SimpleMergeSorter implements SimpleSorter {
   private final String name = "MergeSort";
   private int[] a;
   private int[] aux;

   public String getName() {
      return name;
   }

   public void sort(int[] a) {
      this.a = a;
      this.aux = new int[a.length];
      mergeSort(0, a.length - 1);
   }

   private void mergeSort(int lo, int hi) {
       …
   }
  …
 }
```

Performance of Algorithms. Sorting revisited

Review: Asymptotic Complexity Analysis

Measuring Runtime



Sorting in Java

Comparable and Comparator

Generics: classes and methods

Arrays.sort

```
import java.util.Arrays;


class JavaArraysSort implements SimpleSorter {
    private final String name = "ArraysSort";

    public String getName() {
        return name;
    }

    public void sort(int[] a) {
        Arrays.sort(a);
    }
}
```

The **`Arrays`** class in **`java.util`** has overloaded static methods named **`sort`**, which are used to sort various types of arrays.

`Arrays.sort` can sort arrays of primitive types or arrays of ***objects (under certain conditions)***

Arrays.sort

- Which algorithm is inside Arrays.sort?
- The official documentation on docs.oracle.org says:
  - Sort methods for arrays of **primitive types**:
    - “The sorting algorithm is a **Dual-Pivot Quicksort** by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on all data sets, and is typically faster than traditional (one-pivot) Quicksort implementations”
  - Sort methods for arrays of **objects**:
    - “The implementation was adapted from **Tim Peters's list sort** for Python (TimSort). It uses techniques 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”

Implement Our Generic Sorter

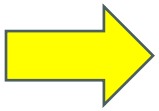
- **Goal: sort arrays with *any* type of data *objects***
  `T[]arr;`
- We create a generic Sorter, with a type parameter T, where T is the type of the array elements.
  - Sorting algorithms: the same
  - Comparing elements: **cannot** use `arr[i]<arr[j]` between objects!
- Two cases:
  - **Case 1:** If there is a ***natural ordering*** between elements of type T, and we use it for sorting
    - Java: use Comparable
  - **Case 2:** If no natural ordering exists, or if we want a different sorting criterion, we have to define ***custom comparison logic***
    - Java: use Comparator

Performance of Algorithms. Sorting revisited

Review: Asymptotic Complexity Analysis

Measuring Runtime

Sorting in Java



Comparable and Comparator

Generics: classes and methods

Generic Sorter – Case 1

- ***We assume that there is a natural ordering relationship between the elements of type T***
  - **natural ordering**: Rules governing the relative placement of all values of a given type.
    - Implies a notion of equality (like `equals`) but also < and > .
    - **total ordering**: All elements can be arranged in A ≤ B ≤ C ≤ ... order.
- The standard way for a Java class to define a comparison function (considered the natural ordering) for its objects is to implement the **`Comparable`** interface.
- The objects themselves "know" the criterion used for their comparison

Interface Comparable

- **`java.lang.Comparable:`**

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

- When a class implements Comparable<T>, it must specify the type of objects that it can be compared with, which is usually the class type itself
- A call of **`A.compareTo(B)`** should return:

  a value < 0 if **A** comes "before" **B** in the ordering,

  a value > 0 if **A** comes "after" **B** in the ordering,

  or exactly 0 if **A** and **B** are considered "equal" in the ordering
  (A.equals(B) should be also true in this case!)

- Classes that implement Comparable: String, Integer, Float, Date, …

Example with Comparable in Arrays.sort

```
import java.util.Arrays;


class Persona implements Comparable<Persona>{
    private String name;
    private int age;

    public Persona(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public int compareTo(Persona o) {
        return (this.name.compareTo(o.name));
    }
}
```

```
Persona[] p = new Persona[3];
p[0] = new Persona("Mary", 20);
p[1] = new Persona("John", 21);
p[2] = new Persona("Ann", 19);

Arrays.sort(p);
```

Sorts array of objects that implement Comparable

Implementing our generic Sorter using Comparable

- We had the example **`SimpleSorter`** for arrays of int
- Now we have **`Sorter`** as a generic API:
  - Type parameter T represents the type of the array elements
  - There must be a natural order relationship over T: T must be a Comparable:
    - **The formal type parameter T is restricted by using the extends keyword: T extends Comparable<T>**

Our Generic Sorter API using Comparable

```
interface Sorter<T extends Comparable<T>> {

    String getName();

    void sort(T[] a);
}
```

Sorter is a **bounded** generic type

Formal type parameter T is restricted to classes or interfaces that implement or extend the interface Comparable

```java
class InsertionSorter<T extends Comparable<T>> implements Sorter<T> {
    private final String name = "InsertionSort";

    public String getName() {
        return name;
    }

    private void exch(T[] a, int i, int j) {
        T temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }

    public void sort(T[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            for (int j = i; j > 0; j--) {
                if (a[j].compareTo(a[j - 1])<0) exch(a, j, j - 1);
                else break;
            }
        }
    }
```

```
class MergeSorter<T extends Comparable<T>> implements Sorter<T> {
    private final String name = "MergeSort";
     private T[] a;
    private  T[] aux;

    public String getName() {
        return name;
    }

    public void sort(T[] a) {
        this.a = a;
        this.aux = (T[]) new Comparable[a.length];    // problem with generic array creation!
        mergeSort(0, a.length - 1);
    }

    private void mergeSort(int lo, int hi) {
    … // the same as before
}

    private void merge(int lo, int mid, int hi) {
   … // the same as before
}
}
```

```
public class SortersExperiments {


   public static void main(String[] args) {
      // Input file names
      String[] inputFiles = {"randomIntegers_10K.txt", "randomIntegers_50K.txt",
     "randomIntegers_100K.txt", "randomIntegers_500K.txt"};

      // Sorting algorithms
      Sorter[] sorters= { new MergeSorter<Integer>(),  new InsertionSorter<Integer>()};

      for (String fileName : inputFiles) {
        for (Sorter<Integer> sorter : sorters) {
            Integer[] a = FileInputHelper.readIntegers(fileName);
            if (a == null) continue;  // Skip if file reading failed

            Stopwatch stopwatch = new Stopwatch();
            sorter.sort(a);
            System.out.println("Elapsed time for " + sorter.getName() + " with n=" +
                        a.length + " is " + stopwatch.getElapsedTime() + " seconds");
         }
      }
   }
}
```

Cannot have array of generic type!

Limitations of Generic Types in Java (1)

- https://dev.java/learn/generics/restrictions/

- Cannot instantiate generic type with primitive types for the type parameter: `Sorter<int> s;` *compile error*
- Cannot create instances of type parameters:

```
class Sorter<T> { …. T x = new T(); }
```
compile error

Limitations of Generic Types in Java (2)

- Cannot create array of elements of the type of the parameter:

  `class Sorter<T> { …. T[] x = new T[10]; }` *compile error*

- Cannot create array of parametrized types:

  `Sorter<Integer> []sorters;` *compile error*

- Cannot declare static fields of the type of the parameter

  `class Sorter<T> { private static T x; … }` *compile error*

- Cannot overload methods that have arguments which differ only by the actual type parameter:

  ```
  void doSmth(Stack<Integer>);
  void doSmth(Stack<Person>);
  ```
  compile error

Cause: Type Erasure: out of scope for this class

- ***optionally if you want*** *you can read more: https://dev.java/learn/generics/type-erasure/*
- *In Java, Parametrized types are compile-time types but not run-time types*
- *Compiler replaces all instances of type Stack<T> with Stack<Object> and all instances of type Sorter<T extends Comparable> with Sorter<Comparable>*

Generic Sorter – Case 2

- ***There is no natural ordering relationship between the elements of type T, we use an external sorting criterion***
- The objects themselves do NOT “know” the criterion used for their comparison
- The criterion used for comparison is “external” to these objects, it is defined within a Comparator
- Decouples the definition of the data type, from the definition of what it means to compare two objects of that type

Interface Comparator

- **`java.util.Comparator`**

```
public interface Comparator<T> {
    public int compare(T first, T second);
}
```

- Interface `Comparator` helps to have an external object that specifies a custom sorting criterion for two objects of type T
  - *Allows you to define multiple orderings for the same type!*
  - *Allows you to define a specific ordering for a type even if there is no obvious "natural" ordering for that type!*
  - *It is a generic interface with formal type parameter T: this enforces that both compared objects are of the same type T*

Arrays.sort Example with Comparator

```
class Persona {
    private String name;
    private int age;
    public Persona(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public int getAge() {
        return this.age;
    }
}
```

```
class PersonAgeComparator implements
Comparator<Persona> {
    @Override
    public int compare(Persona o1, Persona o2) {
        if (o1.getAge()<o2.getAge()) return -1;
        else if (o1.getAge()>o2.getAge()) return 1;
        return 0;
    }
}
```

```
Persona[] p = new Persona[3];
 p[0] = new Persona("Mary", 20);
 p[1] = new Persona("John", 11);
 p[2] = new Persona("Ann", 19);

Arrays.sort(p, new PersonAgeComparator());
```

```
Arrays.sort(p, new PersonAddressComparator());
Arrays.sort(p, new PersonSalaryComparator());
Arrays.sort(p, new WeirdCriteriaComparator());
```

With comparator you can use multiple sorting criteria

Implementing our generic Sorter using Comparator

- Sorter2 as a generic API:
  - Type parameter T represents the type of the array elements
  - There is no natural order relationship over T, thus type parameter T is not restricted (bot bounded)
  - Sort() gets the sorting criteria through another argument, a Comparator!

Our Generic Sorter API using Comparator

```
interface Sorter2<T> {

    String getName();

    void sort(T[] a, Comparator<T> comparator);
}
```

Formal type parameter T is NOT restricted

A Sorter Utility

- The case of `Arrays.sort()`
- A Sorter utility as a library provides just one implementation of a sort => client has *no choice of implementation*
- Clients will call sort, and there is *no state to be maintained in the Sorter in between different calls to the sort method* => there is no need to have different instances of the Sorter! => sort defined as a *static* method!
- All sort methods `Arrays.sort` are static methods
- A generic class cannot have a static method, but a class can have a generic method

Generic Methods

- *Generic methods* are methods that introduce their own type parameters.
  - This is similar to declaring a generic type, but the type parameter's scope is limited to the method only where it is declared
  - The syntax for a generic method includes a list of type parameters, inside angle brackets, which appears before the method's return type

Our generic sort methods

- Class MyArrays is NOT generic
- Has generic sort methods

```
public class MyArrays{
...
public static <T> void sort(T[] a, Comparator<T> c) { …}
...
}
```

list of type parameters, inside angle brackets, must appear before the method's return type.

Type parameters can be used in the method

Our generic sort method – with Comparator

```
class MyArrays {
   …
   static <T> void sort(T[]a, Comparator<T> c){
      int n = a.length;
      for (int i = 0; i < n; i++) {
         for (int j = i; j > 0; j--) {
            if (c.compare(a[j], a[j - 1]) <0) {
               T temp = a[i];
               a[i] = a[j];
               a[j] = temp;
            }
            else break;
         }
      }
}
```

The **complete syntax** for **invoking** this method:

```
MyArrays.<Persona>sort(p, new PersonAgeComparator());
```

The **short syntax** for **invoking** this method, based on type inference:

```
MyArrays.sort(p, new PersonAgeComparator());
```

Our generic sort method – with Comparable

```
class MyArrays {

    static <T extends Comparable<T>> void sort(T[]a){
        int n = a.length;
        for (int i = 0; i < n; i++) {
            for (int j = i; j > 0; j--) {
                if (a[j].compareTo( a[j - 1]) <0) {
                    T temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
                else break;
            }
        }
    }
```

Type parameter T is bounded !

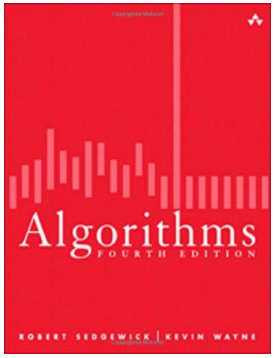
Source code of examples

- Stopwatch.java
- Sorter.java (generic sorter interface, with Comparable)
  - InsertionSorter.java (implements Sorter)
  - MergeSorter.java (implements Sorter)
  - JavaArraysSorter.java (implements Sorter using java.utils.Array)
  - SortersExperiments.java (test harness for all Sorters)
  - FileInputV1.java (utility to load numbers from file into array)
- Sorter2.java (generic sorter interface, with Comparator)
- MyArrays.java (generic sort methods)
- **Data Files:** random integer numbers: randomIntegers_10K.txt, randomIntegers_50K.txt, randomIntegers_100K.txt, randomIntegers_500K.txt, randomIntegers_1M.txt

Summary

- Performance of algorithms:
  - Empirical: measure runtime
  - Analytical: asymptotic complexity
- Sorting
  - Generic sorting: one single implementation is able to sort arrays with data of any type
  - Comparable
  - Comparator
  - Arrays.sort

Bibliography



- Sorting with implementations in Java:
  - [Sedgewick] – chap 2 (Sorting)

- Java Generics Trail in Java Tutorials https://dev.java/learn/generics/

- [Coursera] Algorithms Part I from Princeton University:
  - Algorithms, Part I - Elementary Sorts - Week 5 | Coursera
  - Algorithms, Part I - Mergesort - Week 6 | Coursera