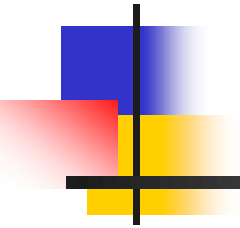
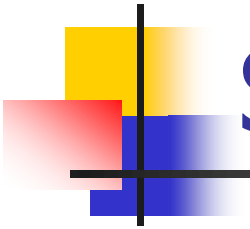
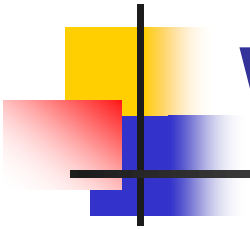


Algorithms and Data Structures



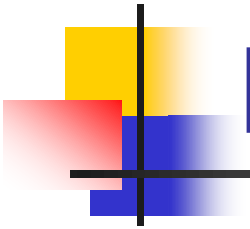
Syllabus

- Introduction (Ch.1)
- Getting Started (Ch.2)
- Growth of Functions (Ch.3)
- Divide-and-Conquer (Ch.4)
- Heapsort & Quicksort (Ch.6-7)
- Elementary Data Structures (Ch.10)
- Hash Tables (Ch.11)
- Binary Search Trees (Ch.12)
- Red-Black Trees (Ch.13)
- Dynamic Programming (Ch.15)
- Greedy Algorithms (Ch.16)
- Amortized Analysis (Ch.17)
- B-Trees (Ch.18)
- Elementary Graph Algorithms (Ch.22)
- Minimum Spanning Trees (Ch.23)
- Single-Source Shortest Paths (Ch.24)
- Computational Geometry (Ch.33)
- NP-Completeness (Ch.34)



What is it all about?

- Solving problems
  - Get me from home to work
  - Balance my checkbook
  - Simulate a jet engine
  - Graduate from SPU
- Using a computer to help solve problems
  - Designing programs (architecture, algorithms)
  - Writing programs
  - Verifying programs
  - Documenting programs



Data Structures and Algorithms

- Algorithm
  - Outline, the essence of a computational procedure, step-by-step instructions
- Program – an implementation of an algorithm in some programming language
- Data structure
  - **Organization** of data needed to solve the problem

Overall Picture

Data Structure and Algorithm Design Goals

Implementation Goals

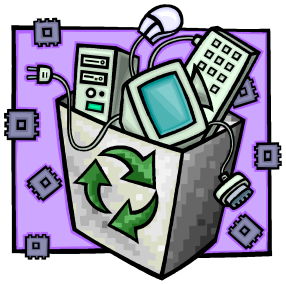
Correctness

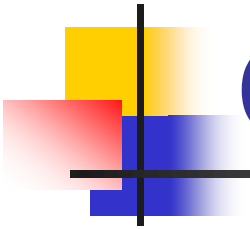
Efficiency

Robustness

Adaptability

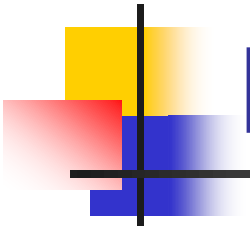
Reusability





Overall Picture (2)

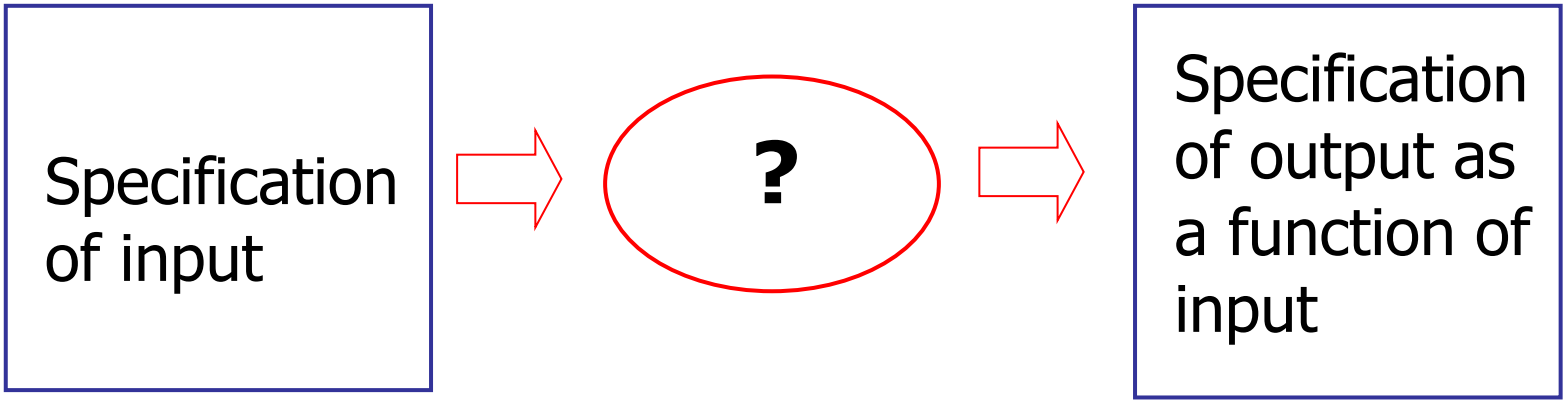
- This course is **not** about:
  - Programming languages
  - Computer architecture
  - Software architecture
  - Software design and implementation principles
    - Issues concerning small and large scale programming
- We will only touch upon the theory of complexity and computability



History

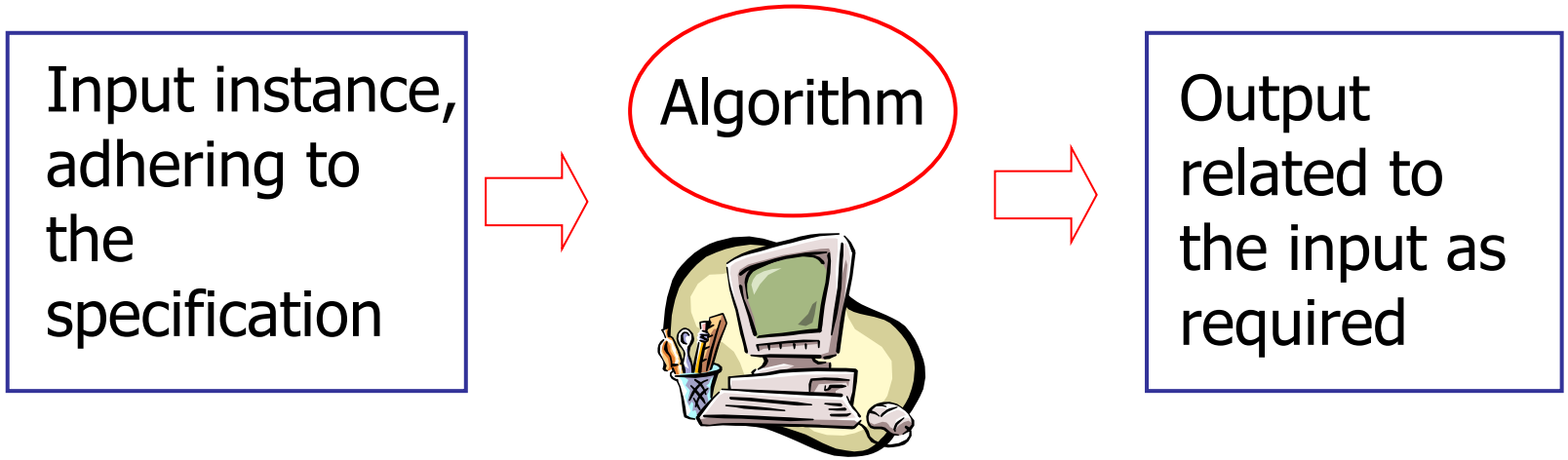
- Name: Persian mathematician Mohammed al-Khowarizmi, in Latin became Algorismus
- First algorithm: Euclidean Algorithm, greatest common divisor, 400-300 B.C.
- 19th century – Charles Babbage, Ada Lovelace.
- 20th century – Alan Turing, Alonzo Church, John von Neumann

Algorithmic problem



- Infinite number of input *instances* satisfying the specification. For example:
  - A sorted, non-decreasing sequence of natural numbers. The sequence is of non-zero, finite length:
    - 1, 20, 908, 909, 100000, 1000000000.
    - 3.

Algorithmic Solution



- Algorithm describes actions on the input instance
- Infinitely many correct algorithms for the same algorithmic problem

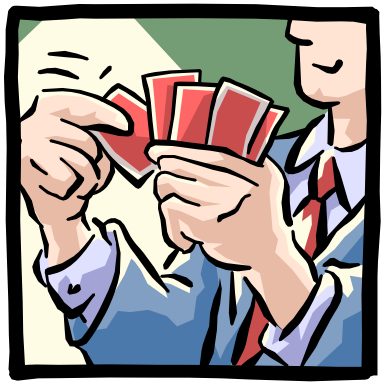
Example: Sorting

INPUT

sequence of numbers

$a_1, a_2, a_3, \ldots, a_n$

2 5 4 10 7



OUTPUT

a permutation of the sequence of numbers

$b_1, b_2, b_3, \ldots, b_n$

2 4 5 7 10

Correctness

For any given input the algorithm halts with the output:

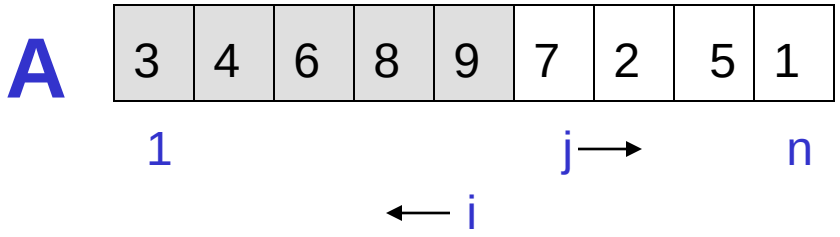
- $b_1 < b_2 < b_3 < \ldots < b_n$
- $b_1, b_2, b_3, \ldots, b_n$ is a permutation of $a_1, a_2, a_3, \ldots, a_n$

Running time

Depends on

- number of elements (n)
- how (partially) sorted they are
- algorithm

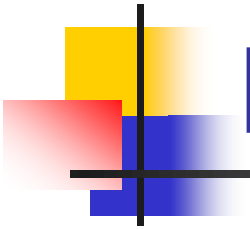
Insertion Sort



Strategy

- Start "empty handed"
- Insert a card in the right position of the already sorted hand
- Continue until all cards are inserted/sorted

```
for j=2 to length(A)
    do key=A[j]
    "insert A[j] into the
    sorted sequence A[1..j-1]"
       i=j-1
       while i>0 and A[i]>key
          do A[i+1]=A[i]
             i--
       A[i+1]:=key
```



Designing Algorithms

- Several techniques/patterns for designing algorithms exist
- Incremental approach: builds the solution one component at a time
  - E.g., iterative looping
- Divide-and-conquer approach: breaks original problem into several smaller instances of the same problem
  - E.g., recursive functions

Algorithms Representation (1)

we prefer not to get bogged down in such formalism, and so we rely on our informal analysis to show that the second property holds for the outer loop.

Termination: Finally, we examine what happens when the loop terminates. The condition causing the **for** loop to terminate is that $j > A.length = n$. Because each loop iteration increases j by 1, we must have $j = n + 1$ at that time. Substituting $n + 1$ for j in the wording of loop invariant, we have that the subarray $A[1..n]$ consists of the elements originally in $A[1..n]$, but in sorted order. Observing that the subarray $A[1..n]$ is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.

We shall use this method of loop invariants to show correctness later in this chapter and in other chapters as well.

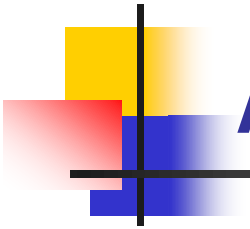
Pseudocode conventions

We use the following conventions in our pseudocode.

- Indentation indicates block structure. For example, the body of the **for** loop that begins on line 1 consists of lines 2–8, and the body of the **while** loop that begins

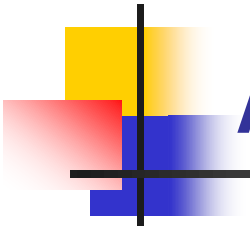
Algorithms Representation (2)

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```



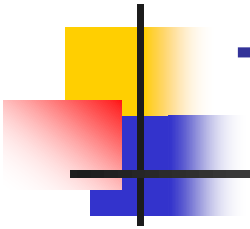
Analysis of Algorithms

- Efficiency:
  - Running time
  - Space used
- Efficiency as a function of input size:
  - Number of data elements (numbers, points)
  - A number of bits in an input number



Ad Hoc Algorithms

- Ad hoc is a Latin phrase which, literraly, means “for this”
- It generally signifies a solution designed for a specific problem or task, non-generalizable, and which cannot be adapted to other purposes



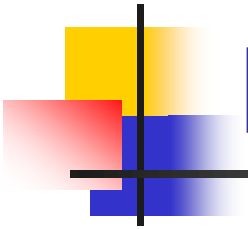
The RAM model

- Very important to choose the level of detail.
- The RAM model:
  - Instructions (each taking constant time):
    - Arithmetic (add, subtract, multiply, etc.)
    - Data movement (assign)
    - Control (branch, subroutine call, return)
  - Data types – integers and floats

Analysis of Insertion Sort

- Time to compute the **running time** as a function of the **input size**

	cost	**times**
for j=2 **to** *length*(A)	c_1	n
do key=A[j]	c_2	n-1
"insert A[j] into the sorted sequence A[1..j-1]"	0	n-1
i=j-1	c_3	n-1
while i>0 **and** A[i]>key	c_4	$\sum_{j=2}^{n} t_j$
do A[i+1]=A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i--	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[i+1]:=key	c_7	n-1

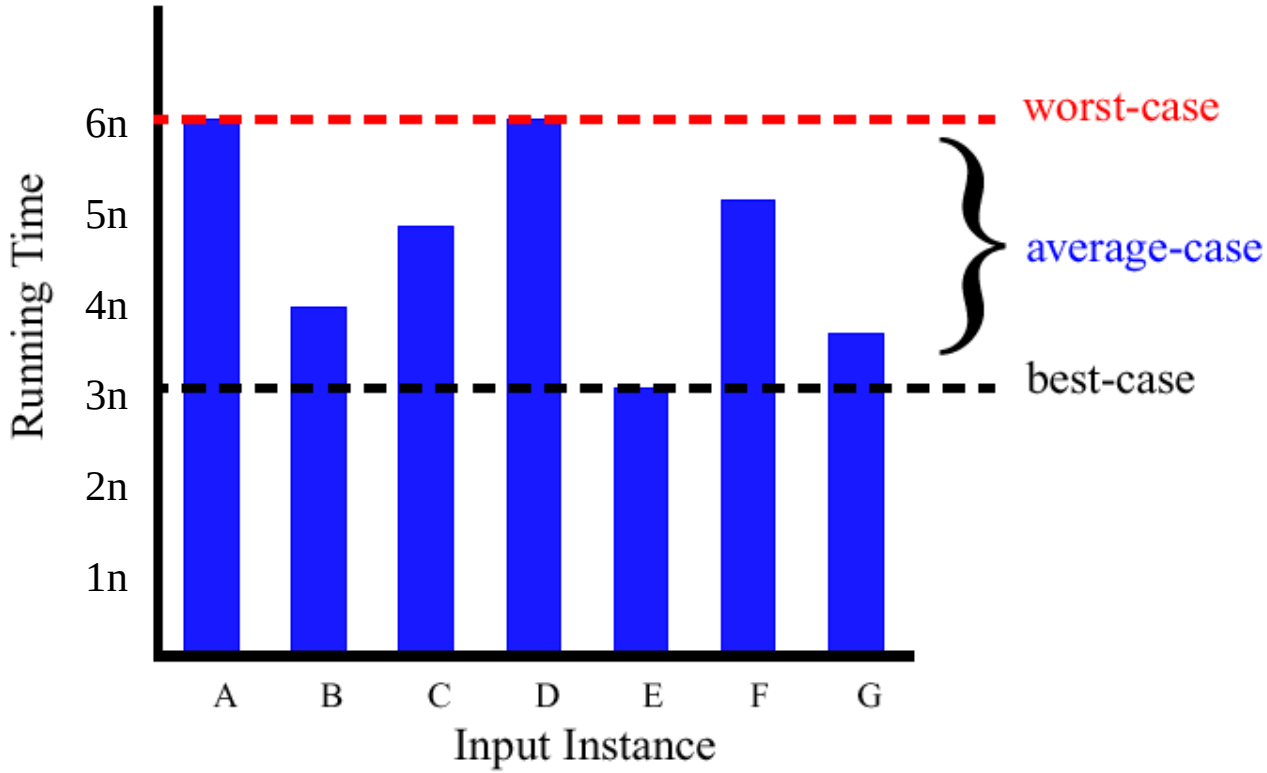


Best/Worst/Average Case

- **Best case**: elements already sorted → $t_j=1$, running time = $f(n)$, i.e., *linear* time.
- **Worst case**: elements are sorted in inverse order
  → $t_j=j$, running time = $f(n^2)$, i.e., *quadratic* time
- **Average case**: $t_j=j/2$, running time = $f(n^2)$, i.e., *quadratic* time

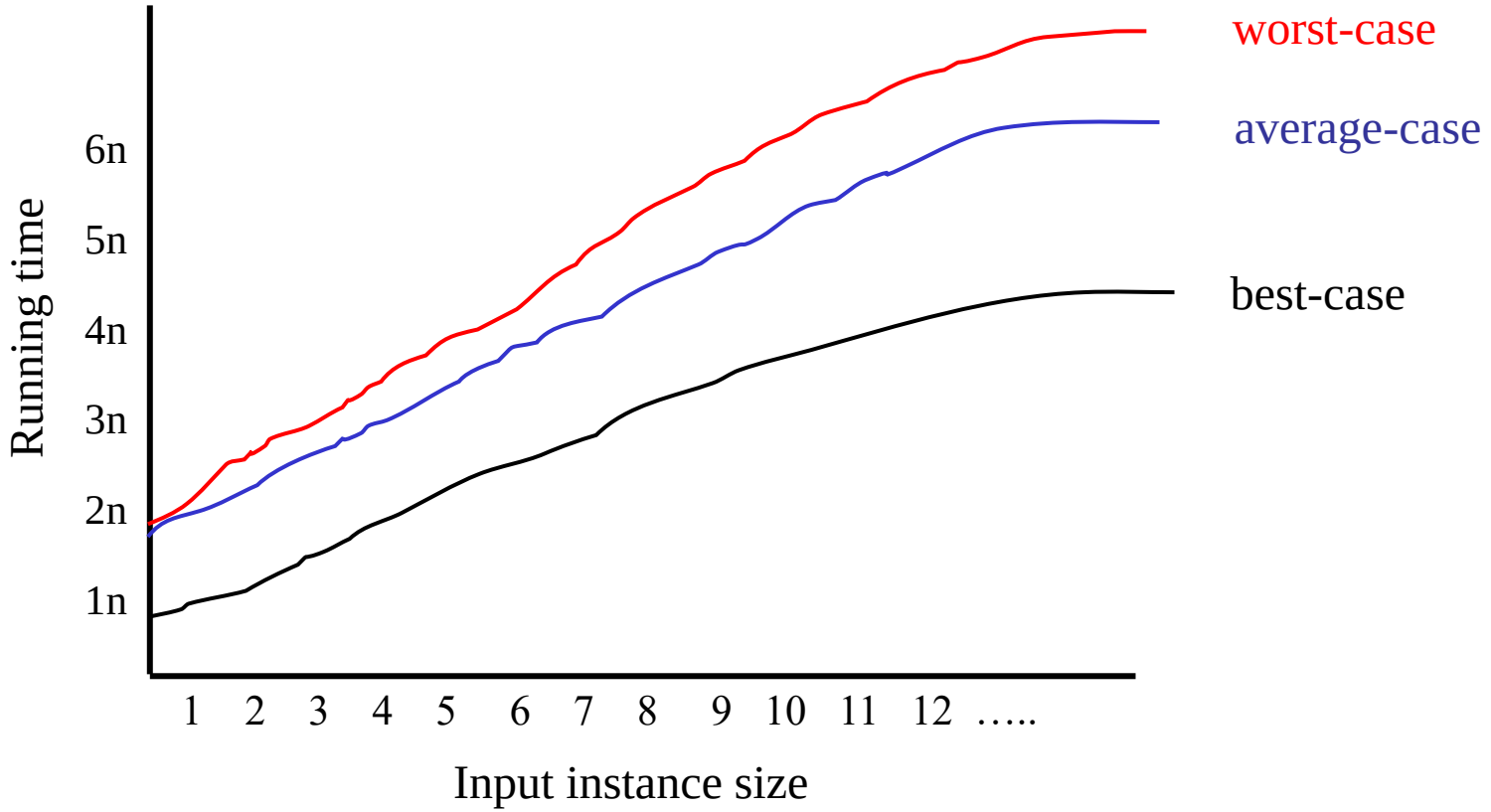
Best/Worst/Average Case (2)

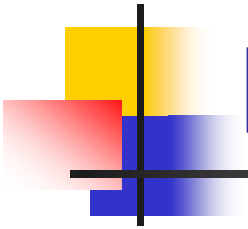
- For a specific size of input *n*, investigate running times for different input instances:



Best/Worst/Average Case (3)

- For inputs of all sizes:

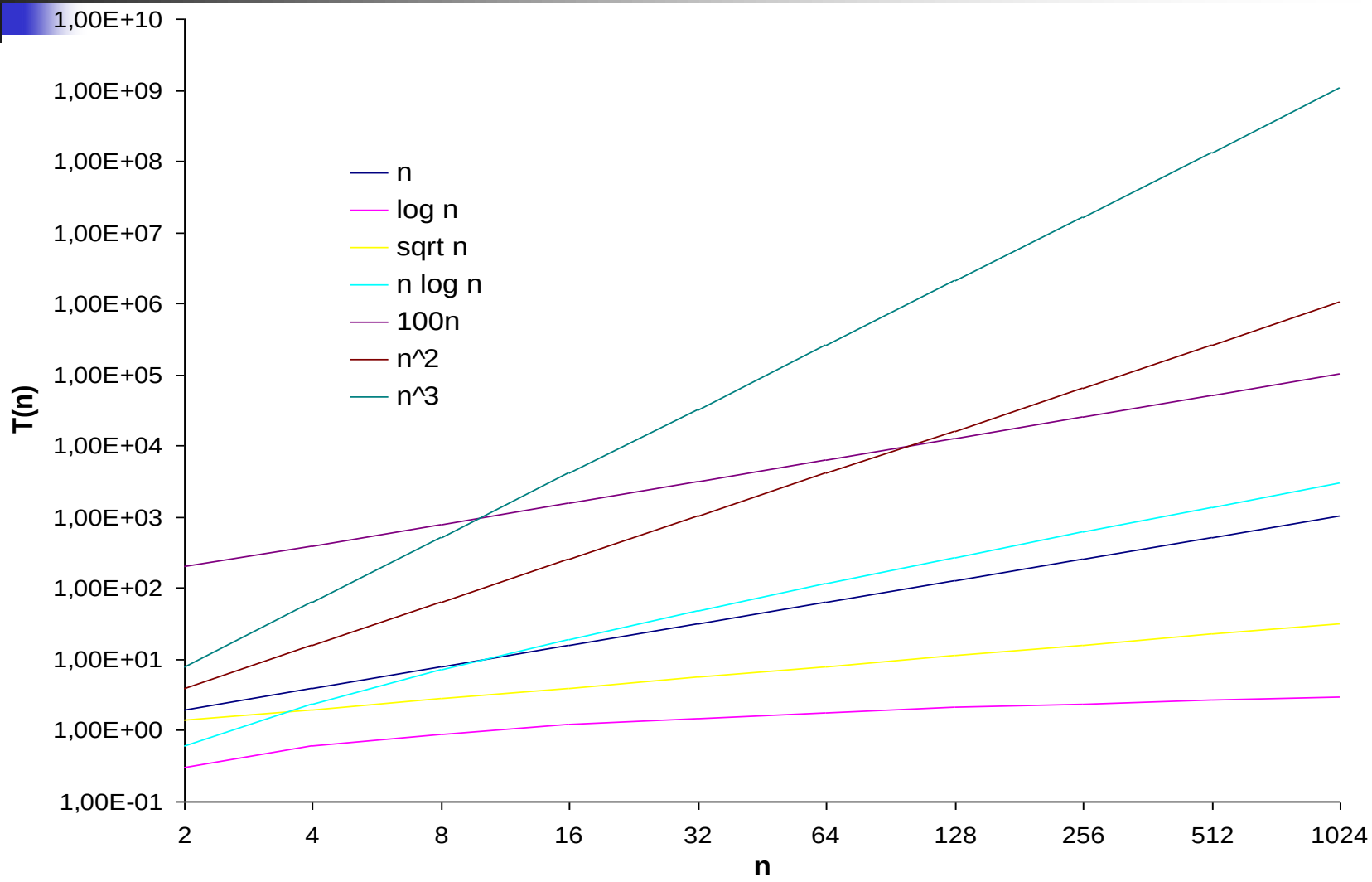




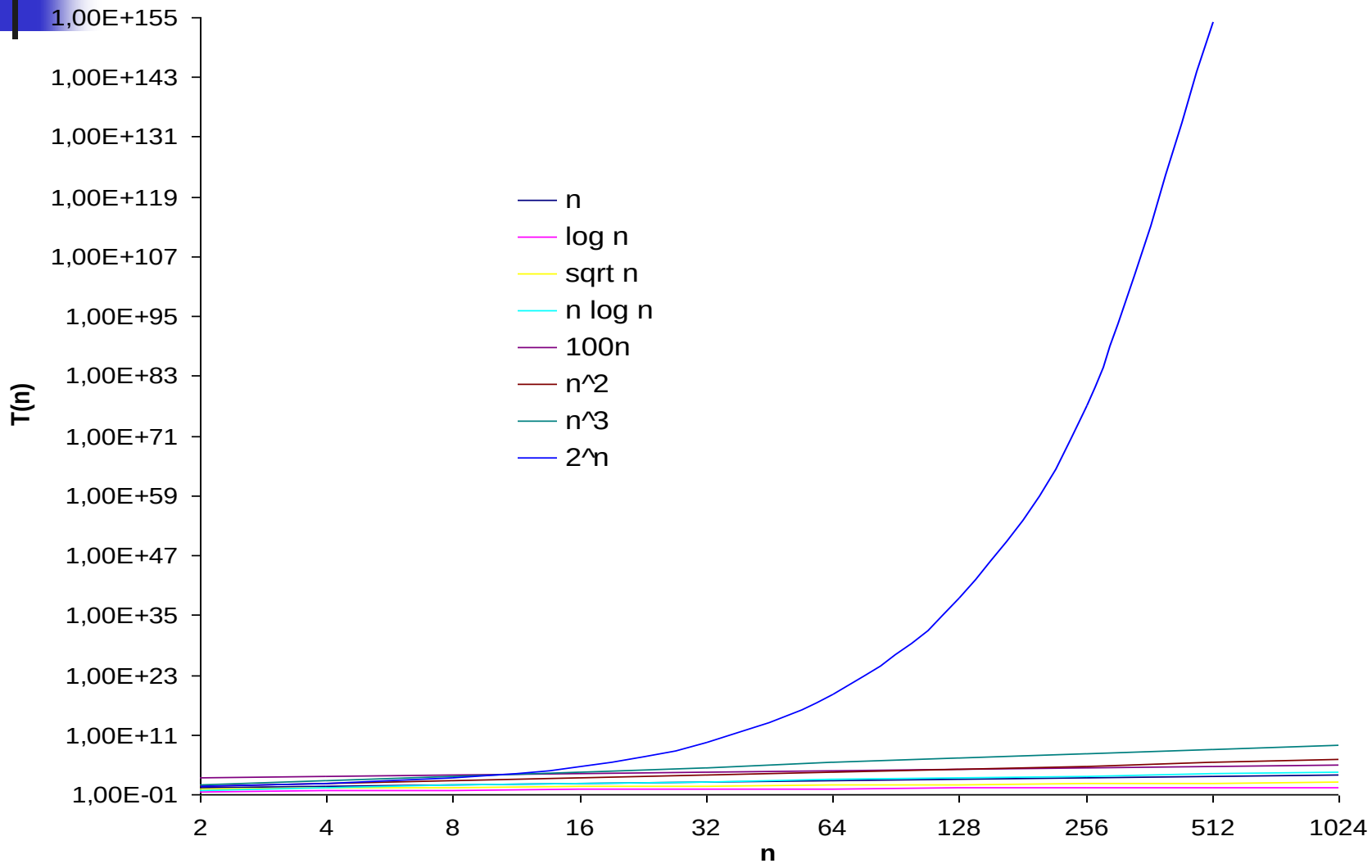
Best/Worst/Average Case (4)

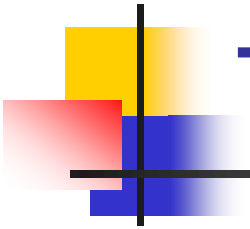
- **Worst case** is usually used:
  - It is an upper-bound and in certain application domains (e.g., air traffic control, surgery) knowing the **worst-case** time complexity is of crucial importance
  - For some algorithms **worst case** occurs fairly often
  - The **average case** is often as bad as the **worst case**
  - Finding the **average case** can be very difficult

Growth Functions



Growth Functions (2)





That's it?

- Is **insertion sort** the best approach to sorting?
- Alternative strategy based on divide and conquer
- MergeSort
  - sorting the numbers <4, 1, 3, 9> is split into
  - sorting <4, 1> and <3, 9> and
  - merging the results
  - Running time *f(n log n)*

Example 2: Searching

INPUT

- sequence of numbers (database)
- a single number (query)

$a_1, a_2, a_3, \ldots, a_n;\ q$

2 5 4 10 7; 5

2 5 4 10 7; 9

OUTPUT

- an index of the found number or *NIL*

j

2

NIL

Searching (2)

```
j=1
while j<=length(A) and A[j]!=q
   do j++
if j<=length(A) then return j
else return NIL
```

- Worst-case running time: *f(n)*, average-case: *f(n/2)*
- We can't do better. This is a *lower bound* for the problem of searching in an arbitrary sequence.

Example 3: Searching

INPUT

- sorted non-descending sequence of numbers (database)
- a single number (query)

$a_1, a_2, a_3, \ldots, a_n;\ q$

2 4 5 7 10; 5

2 4 5 7 10; 9

OUTPUT

- an index of the found number or *NIL*

j

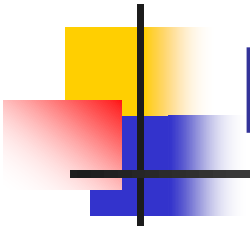
3

NIL

Binary search

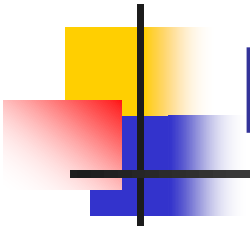
- Idea: Divide and conquer, one of the key design techniques

```
left=1
right=length(A)
do
   j=(left+right)/2
   if A[j]==q then return j
   else if A[j]>q then right=j-1
   else left=j+1
while left<=right
return NIL
```



Binary search – analysis

- How many times the loop is executed:
  - With each execution its length is cut in half
  - How many times do you have to cut *n* in half to get 1?
  - *lg n*



Next lecture

- Correctness of algorithms
- Asymptotic analysis, big *O* notation.