Lecture 1: Asymptotics, Recurrences, Elementary Sorting

Instructor: Saravanan Thirumuruganathan
Outline

1. Introduction to Asymptotic Analysis
   - Rate of growth of functions
   - Comparing and bounding functions: $O, \Theta, \Omega$
   - Specifying running time through recurrences
   - Solving recurrences

2. Elementary Sorting Algorithms
   - Bubble, Insertion and Selection sort
   - Stability of sorting algorithms
In-Class Quizzes

- **URL**: http://m.socrative.com/
- **Room Name**: 4f2bb99e
Time Complexity:
- Quantifies amount of time an algorithm needs to complete as a function of input size

Space Complexity:
- Quantifies amount of space an algorithm needs to complete as a function of input size

Function: Input size Vs {Time, Space}
Best Case Complexity:

- of an algorithm is the function that determines the minimum number of steps taken on any problem instance of size $n$

Worst Case Complexity:

- ... maximum ...

Average Case Complexity:

- ... average ...

Function: Input size Vs \{Time, Space\}
Rate of Growth of Functions

Growth Function $T(n)$

- Input is positive integers $n = 1, 2, 3, \ldots$
- Asymptotically positive (returns positive numbers for large $n$)
- How does $T(n)$ grow when $n$ grows?
- $n$ is size of input
- $T(n)$ is the amount of time it takes for an algorithm to solve some problem
Rate of Growth of Functions

© The McGraw-Hill Companies, Inc. all rights reserved.

The graph shows the growth rates of various functions, including $n!$, $2^n$, $n^2$, $n \log n$, $n$, and $\log n$, comparing them for values of $n$ ranging from 2 to 8.
Question:

- You have a machine that can do million operations per second.
- Your algorithm requires $n^2$ steps
- Suppose size of input is 1 million
- How long does the algorithm takes for this input?
Answer:

- Algorithm will take $(1M)^2$ operations
- Machine can do $1M$ operations per second
- Running time $= \frac{(1M)^2}{1M} = 1M$ seconds
- $1M$ seconds $= \frac{1M}{60\times60\times24} = \text{Approximately 12 days}$
Why does it matter?

Running time of different algorithms for various input sizes

<table>
<thead>
<tr>
<th>n</th>
<th>( n )</th>
<th>( n \log_2 n )</th>
<th>( n^2 )</th>
<th>( n^3 )</th>
<th>( 1.5^n )</th>
<th>( 2^n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>30</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>18 min</td>
</tr>
<tr>
<td>50</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>11 min</td>
<td>36 years</td>
</tr>
<tr>
<td>100</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>12,892 years</td>
<td>10^{17} years</td>
</tr>
<tr>
<td>1,000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>10,000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>100,000</td>
<td>&lt; 1 sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>

\(^1\)Table 2.1 from K&T Algorithm Design. Very long means it takes more than \(10^{25}\) years.
Why does it matter?

- The “Big Data” era
- Can Google/Facebook/… use it?
This is how functions look in the real world!\textsuperscript{2}

\textsuperscript{2}Skiena Lecture notes
Solution: Analyze Asymptotic Behavior

- Analyze the asymptotic behavior of algorithms
- What happens to \( f(n) \) when \( n \to \infty \)?

<table>
<thead>
<tr>
<th></th>
<th>( T(n) = 1000n )</th>
<th>( T(n) = n^2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n = 10 )</td>
<td>10( K )</td>
<td>100</td>
</tr>
<tr>
<td>( n = 100 )</td>
<td>100( K )</td>
<td>10( K )</td>
</tr>
<tr>
<td>( n = 1000 )</td>
<td>1( M )</td>
<td>1( M )</td>
</tr>
<tr>
<td>( n = 10K )</td>
<td>10( M )</td>
<td>100( M )</td>
</tr>
<tr>
<td>( n = 100K )</td>
<td>100( M )</td>
<td>10( B )</td>
</tr>
</tbody>
</table>
Solution: Bound the functions

- Identify known functions (such as $n$, $n^2$, $n^3$, $2^n$, ...) that can “bound” $T(n)$
- How to bound? - asymptotic upper, lower and tight bounds
- Find a function $f(n)$ such that $T(n)$ is proportional to $f(n)$
- Why proportional (as against equal)?
- Ignore aspects such as programming language, programmer capability, compiler optimization, machine specification etc
$O$, $\Omega$, $\Theta^3$

$3$CLRS book
**Upper bounds:** $T(n)$ is $O(f(n))$ if there exists constants $c > 0$ and $n_0 \geq 0$ such that $T(n) \leq c \cdot f(n)$ for all $n \geq n_0$
**Big-O Notation**

**Upper bounds:** \( T(n) \) is \( O(f(n)) \) if there exists **constants** \( c > 0 \) and \( n_0 \geq 0 \) such that \( T(n) \leq c \cdot f(n) \) for all \( n \geq n_0 \)

**Example:** \( T(n) = 32n^2 + 17n + 1 \). Is \( T(n) \) in \( O(n^2) \)?

- Yes! Use \( c = 50, \ n_0 = 1 \)
- Simple Proof:

\[
T(n) \leq 32n^2 + 17n + 1 \\
\leq 32n^2 + 17n^2 + 1n^2 \\
\leq 50n^2 \\
\leq cn^2 \\
c = 50 \text{ and } n_0 = 1
\]

Note: Not necessary to find the smallest \( c \) or \( n_0 \)
Example: \( T(n) = 32n^2 - 17n + 1 \). Is \( T(n) \) in \( O(n^2) \)?
Big-O Notation

Example: \( T(n) = 32n^2 - 17n + 1 \). Is \( T(n) \) in \( O(n^2) \)?

- Yes! Use \( c = 50 \), \( n_0 = 1 \)
- Simple Proof:

\[
T(n) \leq 32n^2 - 17n + 1 \\
\leq 32n^2 + 17n + 1 \\
\leq 32n^2 + 17n^2 + 1n^2 \\
\leq 50n^2 \\
\leq cn^2
\]

\( c = 50 \) and \( n_0 = 1 \)
Example: \( T(n) = 32n^2 - 17n + 1 \). Is \( T(n) \) in \( O(n^3) \)?
Example: \( T(n) = 32n^2 - 17n + 1 \). Is \( T(n) \) in \( O(n^3) \)?

- Yes! Use \( c = 50, \ n_0 = 1 \)
- Simple Proof:

\[
T(n) \leq 32n^2 - 17n + 1 \\
\leq 32n^2 + 17n + 1 \\
\leq 32n^2 + 17n^2 + 1n^2 \\
\leq 50n^2 \\
\leq 50n^3 \\
\leq cn^3 \\
c = 50 \text{ and } n_0 = 1
\]
Example: $T(n) = 32n^2 + 17n + 1$. Is $T(n)$ in $O(n)$?
Example: \( T(n) = 32n^2 + 17n + 1 \). Is \( T(n) \) in \( O(n) \)?

- No!
- Proof by contradiction

\[
32n^2 + 17n + 1 \leq c \cdot n
\]

\[
32n + 17 + \frac{1}{n} \leq c
\]

\[
32n \leq c \quad \text{(ignore constants for now)}
\]

\[
n \leq c \quad \text{(ignore constants for now)}
\]

This inequality does not hold for \( n = c + 1 \)!
Set Theoretic Perspective

- $O(f(n))$ is the set of all functions $T(n)$ where there exist positive constants $c, n_0$ such that $0 \leq T(n) \leq c \cdot f(n)$ for all $n \geq n_0$
- Example: $O(n^2) = \{ n^2, \ldots, 32n^2 + 17n + 1, 32n^2 - 17n + 1, \ldots, n, 2n, \ldots \}$
- Notation: $T(n) = O(f(n))$ or $T(n) \in O(f(n))$
Limit based Perspective

- \( T(n) \) is \( O(f(n)) \) if \( \limsup_{n \to \infty} \frac{T(n)}{f(n)} < \infty \)
- Example: \( 32n^2 + 17n + 1 \) is \( O(n^2) \)

\[
\limsup_{n \to \infty} \frac{T(n)}{f(n)} = \frac{32n^2 + 17n + 1}{n^2}
= 32 + \frac{17}{n} + \frac{1}{n^2}
= 32 < \infty
\]
Lower bounds: $T(n)$ is $\Omega(f(n))$ if there exists constants $c > 0$ and $n_0 \geq 0$ such that $T(n) \geq c \cdot f(n)$ for all $n \geq n_0$.

---

$^5$From K&T: Algorithm Design
Big-Omega Notation

**Lower bounds:** \( T(n) \) is \( \Omega(f(n)) \) if there exists constants \( c > 0 \) and \( n_0 \geq 0 \) such that \( T(n) \geq c \cdot f(n) \) for all \( n \geq n_0 \)

**Example:** \( T(n) = 32n^2 + 17n + 1 \). Is \( T(n) \) in \( \Omega(n^2) \)?

- Yes! Use \( c = 32 \), \( n_0 = 1 \)
- Simple Proof:

\[
T(n) \geq 32n^2 + 17n + 1 \\
\geq 32n^2 \\
\geq cn^2 \\
c = 32 \text{ and } n_0 = 1
\]
Big-Theta Notation

**Tight bounds:** \( T(n) \) is \( \Theta(f(n)) \) if there exists **constants** \( c_1 > 0, c_2 > 0 \) and \( n_0 \geq 0 \) such that \( c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n) \) for all \( n \geq n_0 \)

---

\(^6\)From K&T: Algorithm Design
**Big-Theta Notation**

**Tight bounds:** $T(n)$ is $\Theta(f(n))$ if there exists constants $c_1 > 0$, $c_2 > 0$ and $n_0 \geq 0$ such that $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$ for all $n \geq n_0$

**Example:** $T(n) = 32n^2 + 17n + 1$. Is $T(n)$ in $\Omega(n^2)$?

- Yes! Use $c_1 = 32$, $c_2 = 50$ and $n_0 = 1$
- Combine proofs from before

**Theorem:** For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
Limit based Definitions

Let \( \limsup_{n \to \infty} \frac{T(n)}{f(n)} = c \)

- If \( c < \infty \) then \( T(n) \) is \( O(f(n)) \) (typically \( c \) is zero)
- If \( c > 0 \) then \( T(n) \) is \( \Theta(f(n)) \) (also \( O(f(n)) \) and \( \Omega(f(n)) \))
- If \( c = \infty \) then \( T(n) \) is \( \Omega(f(n)) \)
Some Big-O tips

- Big-O is one of the most useful things you will learn in this class!
- Big-O ignores *constant* factors through $c$
  - Algorithm implemented in Python might need a larger $c$ than one implemented in C++
- Big-O ignores small inputs through $n_0$
  - Simply set a large value of $n_0$
- Suppose $T(n)$ is $O(f(n))$. Typically, $T(n)$ is messy while $f(n)$ is simple
  - $T(n) = 32n^2 + 17n + 1$, $f(n) = n^2$
- Big-O hides constant factors. Some times using an algorithm with worser Big-O might still be a good idea (e.g. sorting, finding medians)
## Survey of Running Times

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>Constant time</td>
<td>Function that returns a constant (say 42)</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>Logarithmic</td>
<td>Binary Search</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>Linear</td>
<td>Finding Max of an array</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>Linearithmic</td>
<td>Sorting (for e.g. Mergesort)</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>Quadratic</td>
<td>Selection sort</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>Cubic</td>
<td>Floyd-Warshall</td>
</tr>
<tr>
<td>$O(n^k)$</td>
<td>Polynomial</td>
<td>Subset-sum with $k$ elements</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>Exponential</td>
<td>Subset-sum with no cardinality constraints</td>
</tr>
</tbody>
</table>
Dominance Rankings

- $n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \log n \gg 1$
- Exponential algorithms are useless even at $n \geq 50$
- Quadratic algorithms at around $n \geq 1M$
- $O(n \log n)$ at around $n \geq 1B$

\textsuperscript{7}Skiena lecture notes
Closer Look at $T(n)$

- So far we assumed someone gave us $T(n)$
- What is $n$? (Program Analysis)
- How do we get $T(n)$? (Recurrences)
for $i=1$ to $n$
{
    constant time operations
}

for $i=1$ to $n$
{
    for $j=1$ to $n$
    {
        constant time operations
    }
}
Recurrences

- Typically programs are lot more complex than that
- Recurrences occur in recursion and divide and conquer paradigms
- Specify running time as a function of $n$ and running time over inputs of smaller sizes
- Examples:
  - $\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$
  - $T(n) = 2T\left(\frac{n}{2}\right) + n$
  - $T(n) = T(n - 1) + n$
Solving Recurrences

- Unrolling
- Guess and prove by induction (aka Substitution)
- Recursion tree
- Master method
Let $T(n) = T(n-1) + n$. Base case: $T(1) = 1$

\[
T(n) = T(n-1) + n \\
= n + T(n-1) \\
= n + n - 1 + T(n-2) \\
= n + n - 1 + n - 2 + T(n-3) \\
= n + n - 1 + n - 2 + n - 3 + \ldots + 4 + 3 + 2 + 1 \\
= \frac{n(n+1)}{2} \\
= 0.5n^2 + 0.5n \\
= O(n^2)
\]
Solve $T(n) = 2T(n-1)$. Base case: $T(1) = 1$
Quiz!

Solve \( T(n) = 2T(n - 1) \). Base case: \( T(1) = 1 \)

\[
T(n) = 2T(n - 1) \\
= 2(2T(n - 2)) \\
= 2(2(2T(n - 3))) \\
= 2^3 T(n - 3) \\
= 2^i \ldots 2T(n - i) \\
= 2^n \\
= O(2^n)
\]
Recursion Tree

\[ T(n) = T\left(\frac{2n}{3}\right) + T\left(\frac{n}{3}\right) + cn \]

Total: \( O(n \log n) \)

\[ \text{From CLRS} \]
Logarithms

- Logarithm is an *inverse* exponential function
- $b^x = n$ implies $x = \log_b n$
- If $b = 2$, logarithms reflect how many times we can double something until we get $n$ or halve something till we get 1
- Example: $2^4 = 16$, $\log_2 16 = \lg 16 = 4$
- Example: You need $\lg 256 = 8$ bits to represent $[0, 255]$
- Identities:
  - $\log_b(xy) = \log_b(x) + \log_b(y)$
  - $\log_b a = \frac{\log_c a}{\log_c b}$
  - $\log_b b = 1$ and $\log_b 1 = 0$

---

Skiena Lectures

---

9 Skiena Lectures
Master Method

- A “black box” method to solve recurrences that occur from Divide and Conquer algorithms
- Divide, Conquer and Combine steps
- \( T(n) = aT\left(\frac{n}{b}\right) + f(n) \)
- Assumes that all sub-problems are of equal size
- \( a \) - number of sub-problems (equivalently, \#recursive calls)
- \( b \) - the rate at which the problem shrinks
- Note: \( a \) and \( b \) must be constants (it cannot be for e.g. \( \sqrt{n} \))
- \( f(n) \) - complexity of the combine step
Master Theorem: Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants (for e.g. they cannot be $\sqrt{n}$). Let $T(n)$ be defined on the non-negative integers by recurrence as

$$T(n) = aT\left(\frac{n}{b}\right) + n^d$$

Then $T(n)$ has the following asymptotic bounds:

$$T(n) = \begin{cases} 
O(n^d \log n) & \text{if } a = b^d \\
O(n^d) & \text{if } a < b^d \\
O(n^{\log_b a}) & \text{if } a > b^d 
\end{cases}$$
$T(n) = 2T\left(\frac{n}{2}\right) + n$

- $a = 2$, $b = 2$ and $d = 1$ (as $n = n^1$)
- $b^d = 2^1 = 2 = a$
- By case 1, $T(n) = O(n^d \log n) = O(n \log n)$

\[^{10}\text{From Tim Roughgarden’s notes}\]
$T(n) = 2T\left(\frac{n}{2}\right) + n^2$

- $a = 2$, $b = 2$ and $d = 2$
- $b^d = 2^2 = 4 > a$
- By case 2, $T(n) = O(n^d) = O(n^2)$

---

\(^{11}\)From Tim Roughgarden’s notes
Master Method - Examples \(^{12}\)

\[ T(n) = 4T\left(\frac{n}{2}\right) + n \]

- \(a = 4\), \(b = 2\) and \(d = 1\) (as \(n = n^1\))
- \(b^d = 2^1 = 2 < a\)
- By case 3, \(T(n) = O(n^{\log_b a}) = O(n^{\log_2 4}) = O(n^2)\)

\(^{12}\text{From Tim Roughgarden’s notes}\)
Solve $T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$. Let’s solve it step by step!
Solve \( T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2 \)

- \( a = 8, \ b = 2 \) and \( d = 2 \)
- \( b^d = 2^2 < a \)
- Falls in Case 3 of Master Theorem
- \( O(n^{\log_b a}) = O(n^{\log_2 8}) = O(n^3) \)

\(^{14}\text{From Wikipedia}\)
Sorting Problem

- **Input:** A sequence of $n$ numbers $\langle a_1, a_2, \ldots, a_n \rangle$
- **Output:** A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$
- **Example:** $\langle 4, 2, 1, 3, 5 \rangle$ to $\langle 1, 2, 3, 4, 5 \rangle$
- Assume distinct values (doesn't affect correctness or analysis)
Applications of Sorting

1. Direct applications
   - Sorting a list of names in dictionary
   - Sorting search results based on Google’s ranking algorithm

2. Problems made simpler after sorting
   - Finding median, frequency distribution
   - Finding duplicates
   - Binary search
   - Closest pair of points

3. Non-obvious applications
   - Data compression (e.g. Huffman encoding)
   - Computer Graphics (e.g Convex hulls)
   - Many many more!

$^{15}$From Slides of Kevin Wayne
Comparison based sorting
- Time complexity measured in terms of comparisons
- Elementary algorithms: Bubble, Selection and Insertion sort
- Mergesort and Quicksort

Non-comparison based sorting
- Bucket, Counting and Radix sort
Facets of Sorting Algorithms

- Best, Average and Worst case time complexity
- Worst case space complexity
- **In-Place:** Transforms the input data structure with only constant additional space
- **Stability:** The relative order of items with same key values. E.g. \( \langle 100_1, 400, 100_2, 200 \rangle \rightarrow \langle 100_1, 100_2, 200, 400 \rangle \)
- **Adaptive:** Can it leverage the “sortedness” of input?
Basic Idea:
- Compare adjacent elements
- Swap them if they are in wrong order
- Repeat till entire array is sorted
Bubble Sort

http://www.cs.miami.edu/~ogihara/csc220/slides/Ch08.pdf
Bubble Sort

Pseudocode:

BubbleSort(A):
    for i = 1 to A.length - 1
        for j = A.length downto i+1
                swap(A[j-1], A[j])

Loop Invariant: First \( i - 1 \) elements are in sorted order

Better Implementation: Count swaps within an iteration. Terminate if no swaps.
Bubble Sort Properties:

- Best case time complexity: $O(n)$
- Worst case time complexity: $O(n^2)$
- Adaptive: Yes
- In-Place: Yes
- Stability: Yes
Selection Sort

- **Basic Idea:**
  - Find smallest element and exchange it with A[1]
  - Find second smallest element and exchange it with A[2]
  - Repeat process till entire array is sorted
Selection Sort

1. Selection Sort

88 25 14 92 64

14 25 88 92 64

14 25 88 92 64

14 25 64 92 88

88 <-> 14

25 <-> 25

88 <-> 64

88 <-> 92

14 25 88 92 64

14 25 64 92 88

14 25 64 88 92

http://www.cs.miami.edu/~ogihara/csc220/slides/Ch08.pdf
Selection Sort

Pseudocode:

SelectionSort(A):
    for i = 1 to A.length
        k = i
        for j = i+1 to A.length
            if A[j] < A[k]
                k = j
        swap(A[i], A[k])

Loop Invariant: First i – 1 elements are in sorted order
Selection Sort Properties:

- Best case time complexity: $O(n^2)$
- Worst case time complexity: $O(n^2)$
- Adaptive: No
- In-Place: Yes
- Stability: No (but can be made to one with some effort)
Insertion Sort

- Best of the elementary sorting algorithms
- **Basic Idea:**
  - Start with an empty sorted array
  - Pick an element and insert into the right place
  - Repeat till all elements are handled
Insertion Sort\textsuperscript{18}

\begin{figure}
\centering
\includegraphics[width=0.8\textwidth]{insertion-sort-diagram.png}
\caption{Insertion Sort Algorithm}
\end{figure}

\textsuperscript{18}http://www.cs.miami.edu/~ogihara/csc220/slides/Ch08.pdf
Pseudocode:

InsertionSort(A):
    for i = 2 to A.length
        key = A[i]
        j = i - 1
        while j > 0 and A[j] > key
            j = j - 1
        A[j+1] = key

Loop Invariant: At start of i-th iteration, subarray A[1..i-1] consists of elements originally in A[1..i-1] but in sorted order
Quiz!

Insertion Sort Properties:

- Best case time complexity: $O(n)$
- Worst case time complexity: $O(n^2)$
- Adaptive: Yes
- In-Place: Yes
- Stability: Yes
Suppose you have an array with identical elements. Which sort would take the least time?
Suppose you have an array with identical elements. Which sort would take the least time?

- Insertion sort
- Bubble sort
Suppose you have an array that is $k$-sorted - i.e. each element is at most $k$ away from its target position. For example, $\langle 1, 3, 0, 2 \rangle$ is a $2$—sorted array.
Suppose you have an array that is $k$-sorted - i.e. each element is at most $k$ away from its target position. For example, $\langle 1, 3, 0, 2 \rangle$ is a 2–sorted array.

- Insertion sort
Summary

Major Topics:
- Asymptotics, $O$, $\Omega$, $\Theta$, Recurrences, Master method
- Sorting, facets of Sorting, elementary Sorting algorithms