Data Structures and Algorithms

Jörg Endrullis

Vrije Universiteit Amsterdam
Department of Computer Science
Section Theoretical Computer Science

2009–2010

http://www.few.vu.nl/~tcs/ds/
Lectures:

▶ Tuesday 11:00 – 12:45
  (weeks 36–42 in WN-KC159)

▶ Wednesday 9:00 – 10:45
  (weeks 36–41 in WN-Q105, 42 in WN-Q112)

Exercise classes:

▶ Thursday 11:00 – 12:45 (weeks 36–42 in WN-M607(50))

▶ Thursday 13:30 – 15:15 (weeks 36–42 in WN-M623(50))

▶ Thursday 15:30 – 17:15 (weeks 36–42 in WN-C121)

Exam on 19.10.2009 at 8:45 – 11:30.

Retake exam on 12.01.2010 at 18:30 – 21:15.
Voortentamen (pre-exam):

- **Tuesday 29 September, 11:00 – 12:45 in WN-KC159**
- The participation is **not obligatory**, but recommended.
- The result **can influence the final grade only positively**:
  - Let $V$ be the pre-exam grade, and $T$ the exam grade. The final grade is calculated by:
    \[
    \max(T, \frac{2T + V}{3})
    \]

Thus if the final exam grade $T$ is higher than the pre-exam grade $V$, then only the final grade $T$ counts. But if the pre-exam $V$ is higher then it counts with $\frac{1}{3}$. 
Contact Persons

- **lecture:**
  - Jörg Endrullis
    - joerg@few.vu.nl

- **exercise classes:**
  - Dennis Andriessen
    - dae400@few.vu.nl
  - Michel Degenhardt
    - ldt200@few.vu.nl
  - Atze van der Ploeg
    - atzeus@gmail.com
Definition

An algorithm is a list of instructions for completing a task.

- Algorithms are central to computer science.
- Important aspects when designing algorithms:
  - correctness
  - termination
  - efficiency
Evaluation of Algorithms

- Algorithms with equal functionality may have huge differences in efficiency (complexity).
- Important measures:
  - time complexity
  - space (memory) complexity
- Time and memory usage increase with size of the input:
Time Complexity

Running time of programs depends on various factors:

- input for the program
- compiler
- speed of the computer (CPU, memory)
- time complexity of the used algorithm

Average case often difficult to determine.

We focus on worst case running time:

- easier to analyse
- usually the crucial measure for applications
Methods for Analysing Time Complexity

**Experiments** (measurements on a certain machine):
- requires implementation
- comparisons require equal hardware and software
- experiments may miss out important inputs

Calculation of complexity **for idealised computer model**:
- requires exact counting
- computer model: Random Access Machine (RAM)

**Asymptotic** complexity estimation depending on input size $n$:
- allows for approximations
- logarithmic ($\log_2 n$), linear ($n$), $\ldots$, exponential ($a^n$), $\ldots$
A Random Access Machine is a CPU connected to a memory:

- potentially unbounded number of memory cells
- each memory cell can store an arbitrary number

- primitive operations are executed in constant time
- memory cells can be accessed with one primitive operation
We use Pseudo-code to describe algorithms:

- programming-like, high-level description
- independent from specific programming language
- primitive operations:
  - assigning a value to a variable
  - calling a method
  - arithmetic operations, comparing two numbers
  - array access
  - returning from a method
Counting Primitive Operations: \textit{arrayMax}

We analyse the \textbf{worst case complexity} of \textit{arrayMax}.

\textbf{Algorithm} \textit{arrayMax}(A, n):

\begin{enumerate}
\item \textbf{Input}: An array storing $n \geq 1$ integers.
\item \textbf{Output}: The maximum element in $A$.
\end{enumerate}

\begin{verbatim}
currentMax = A[0] for i = 1 to n - 1 do 
  if currentMax < A[i] then
    currentMax = A[i]
done
return currentMax
\end{verbatim}

Hence we have a \textbf{worst case time complexity}:

\begin{align*}
T(n) &= 2 + 1 + n + (n - 1) \cdot 6 + 1 \\
&= 7 \cdot n - 2
\end{align*}
We analyse the **best case complexity** of *arrayMax*. 

**Algorithm** *arrayMax*(A, n): 

**Input:** An array storing \( n \geq 1 \) integers. 

**Output:** The maximum element in A.

\[
\text{currentMax} = A[0] \quad 1 + 1
\]

for \( i = 1 \) to \( n - 1 \) do 

\[
\begin{align*}
\text{if currentMax} & < A[i] \text{ then} \\
\text{currentMax} & = A[i] \\
\end{align*}
\]

1 + 1

done 

1 + 1

return \( \text{currentMax} \) 

1

Hence we have a **best case time complexity**:

\[
T(n) = 2 + 1 + n + (n - 1) \cdot 4 + 1
\]

\[
= 5 \cdot n
\]
Important Growth Functions

Examples of growth functions, ordered by speed of growth:

- \( \log n \) — logarithmic growth
- \( n \) — linear growth
- \( n \cdot \log n \) — \((n \cdot \log n)\)-growth
- \( n^2 \) — quadratic growth
- \( n^3 \) — cubic growth
- \( a^n \) — exponential growth \((a > 1)\)

We recall the definition of logarithm:

\[
\log_a b = c \quad \text{such that} \quad a^c = b
\]
## Speed of Growth

<table>
<thead>
<tr>
<th></th>
<th>$n$</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>$10^4$</th>
<th>$10^5$</th>
<th>$10^6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\log_2 n$</td>
<td>3</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>17</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>$n$</td>
<td>10</td>
<td>100</td>
<td>1000</td>
<td>$10^4$</td>
<td>$10^5$</td>
<td>$10^6$</td>
<td></td>
</tr>
<tr>
<td>$n \cdot \log n$</td>
<td>30</td>
<td>700</td>
<td>13000</td>
<td>$10^5$</td>
<td>$10^6$</td>
<td>$10^7$</td>
<td></td>
</tr>
<tr>
<td>$n^2$</td>
<td>100</td>
<td>10000</td>
<td>$1^6$</td>
<td>$10^8$</td>
<td>$10^{10}$</td>
<td>$10^{12}$</td>
<td></td>
</tr>
<tr>
<td>$n^3$</td>
<td>1000</td>
<td>$1^6$</td>
<td>$1^9$</td>
<td>$10^{12}$</td>
<td>$10^{15}$</td>
<td>$10^{18}$</td>
<td></td>
</tr>
<tr>
<td>$2^n$</td>
<td>1024</td>
<td>$10^{30}$</td>
<td>$10^{300}$</td>
<td>$10^{3000}$</td>
<td>$10^{30000}$</td>
<td>$10^{300000}$</td>
<td></td>
</tr>
</tbody>
</table>

Note that $\log_2 n$ growth very slow, whereas $2^n$ growth explosive!
Time depending on Problem Size

Computation time assuming that 1 step takes $1\mu s$ (0.000001s).

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>$10^4$</th>
<th>$10^5$</th>
<th>$10^6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\log n$</td>
<td>$&lt;$</td>
<td>$&lt;$</td>
<td>$&lt;$</td>
<td>$&lt;$</td>
<td>$&lt;$</td>
<td>0.00002s</td>
</tr>
<tr>
<td>$n$</td>
<td>$&lt;$</td>
<td>$&lt;$</td>
<td>0.001s</td>
<td>0.01s</td>
<td>0.1s</td>
<td>1s</td>
</tr>
<tr>
<td>$n \cdot \log n$</td>
<td>$&lt;$</td>
<td>$&lt;$</td>
<td>0.013s</td>
<td>0.1s</td>
<td>1s</td>
<td>10s</td>
</tr>
<tr>
<td>$n^2$</td>
<td>$&lt;$</td>
<td>0.01s</td>
<td>1s</td>
<td>100s</td>
<td>3h</td>
<td>1000h</td>
</tr>
<tr>
<td>$n^3$</td>
<td>0.001s</td>
<td>1s</td>
<td>1000s</td>
<td>$1000h$</td>
<td>100y</td>
<td>$10^5y$</td>
</tr>
<tr>
<td>$2^n$</td>
<td>0.001s</td>
<td>$10^{23}y$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
<td>$&gt;$</td>
</tr>
</tbody>
</table>

Here $<$ means ‘fast’ ($< 0.001s$), and $>$ ‘more than $10^{300}$ years’.

A problem for which there exist only exponential algorithms are usually considered intractable.
Search in Arrays

We analyse the **worst case complexity** of *search*.

**Algorithm** *search*(\(A, n, x\)):

- **Input:** An array storing \(n \geq 1\) integers.
- **Output:** Returns **true** if \(A\) contains \(x\) and **false**, otherwise.

```plaintext
for i = 0 to n - 1 do
    if A[i] == x then
        return true
    done
return false
```

Hence we have a **worst case time complexity**:

\[
T(n) = 1 + (n + 1) + n \cdot 4 + 1 = 5 \cdot n + 3
\]
Search in Arrays

We analyse the **best case complexity** of *search*.

**Algorithm** `search(A, n, x)`:

- **Input:** An array storing $n \geq 1$ integers.
- **Output:** Returns `true` if $A$ contains $x$ and `false`, otherwise.

```plaintext
for i = 0 to n − 1 do
    if A[i] == x then
        return true
    done
return false
```

Hence we have a **best case time complexity**:

$$T(n) = 5$$
**Algorithm** *binSearch*(A, n, x):

**Input:** An array storing n integers in ascending order.

**Output:** Returns **true** if A contains x and **false**, otherwise.

\[ low = 0 \]
\[ high = n - 1 \]

while \( low \leq high \) do

\[ mid = \lfloor (low + high)/2 \rfloor \]
\[ y = A[mid] \]

if \( x < y \) then \( high = mid - 1 \)

if \( x == y \) then return **true**

if \( x > y \) then \( low = mid + 1 \)

done

return **false**

---

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>11</th>
<th>13</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>( low )</td>
<td>( mid )</td>
<td>( high )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( x = 8 \)
\( y = 7 \)
Algorithm \textit{binSearch}(A, n, x):

\textbf{Input:} An array storing \(n\) integers in ascending order.
\textbf{Output:} Returns \textbf{true} if \(A\) contains \(x\) and \textbf{false}, otherwise.

\(low = 0\)
\(high = n - 1\)

\textbf{while} \(low \leq high\) \textbf{do}

\hspace{1em} \(mid = \lfloor (low + high)/2 \rfloor\)
\hspace{1em} \(y = A[mid]\)
\hspace{1em} \textbf{if} \(x < y\) \textbf{then} \(high = mid - 1\)
\hspace{1em} \textbf{if} \(x == y\) \textbf{then} \textbf{return} \textbf{true}
\hspace{1em} \textbf{if} \(x > y\) \textbf{then} \(low = mid + 1\)

\textbf{done}

\textbf{return} false

\[
\begin{array}{cccccccc}
1 & 2 & 2 & 4 & 6 & 7 & 8 & 11 & 13 & 15 & 16 \\
\end{array}
\]

\(x = 8\)
\(y = 13\)
Search in Sorted Arrays: Binary Search

Algorithm `binSearch(A, n, x)`:  
**Input:** An array storing \( n \) integers in ascending order.  
**Output:** Returns `true` if \( A \) contains \( x \) and `false`, otherwise.

\[
\begin{align*}
\text{low} &= 0 \\
\text{high} &= n - 1 \\
\textbf{while} \ \text{low} \leq \text{high} \ \textbf{do} \\
\quad &\text{mid} = \lfloor (\text{low} + \text{high})/2 \rfloor \\
\quad &y = A[\text{mid}] \\
\quad &\textbf{if} \ x < y \ \textbf{then} \ \text{high} = \text{mid} - 1 \\
\quad &\textbf{if} \ x == y \ \textbf{then} \ \text{return} \ \text{true} \\
\quad &\textbf{if} \ x > y \ \textbf{then} \ \text{low} = \text{mid} + 1 \\
\textbf{done} \\
\textbf{return} \ \text{false}
\end{align*}
\]

\[
\begin{array}{cccccccccc}
1 & 2 & 2 & 4 & 6 & 7 & 8 & 11 & 13 & 15 & 16 \\
\end{array}
\]

\( x = 8 \)

\( y = 8 \)
Search in Sorted Arrays: Binary Search

We analyse the **worst case complexity** of `binSearch`.

\[
\begin{align*}
\text{low} &= 0 & 1 \\
\text{high} &= n - 1 & 2 \\
\text{while } & \text{low } \leq \text{high} \text{ do} & \text{number of loops } \cdot (1+ \\
& \quad \ldots & \quad \ldots) \\
\text{done} \\
\text{return false}
\end{align*}
\]

The analyse the maximal number of loops \( L \) for minimal lists \( A \):

<table>
<thead>
<tr>
<th>( L )</th>
<th>( A )</th>
<th>( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1]</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>[1 2]</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>[1 2 3 4]</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>[1 2 3 4 5 6 7 8]</td>
<td>9</td>
</tr>
</tbody>
</table>

For a list \( A \) of length \( n \) we need \( (1 + \log_2 n) \) loops in worst case.
Search in Sorted Arrays: Binary Search

We analyse the **worst case complexity** of `binSearch`.

\[
\begin{align*}
low & = 0 & 1 \\
high & = n - 1 & 2 \\
\textbf{while } & low \leq high & 1 + (1 + \log_2 n) \cdot (1 + \\
& \text{do} & 4 \\
& \quad \text{mid} & \lfloor (low + high)/2 \rfloor & 2 \\
& \quad y & A[mid] & 2 \\
& \quad \textbf{if } & x < y \quad \textbf{then } & high = mid - 1 & 1 \text{ (false)} \\
& \quad & \textbf{if } x == y \quad \textbf{then } & \textbf{return true} & 1 \text{ (false)} \\
& \quad & \textbf{if } x > y \quad \textbf{then } & low = mid + 1 & 3 \text{ (true)} \\
& \text{done} & & \\
& \textbf{return} & \text{false} & 1 \\
\end{align*}
\]

Hence we have a **worst case time complexity**:

\[
T(n) = 5 + (1 + \log_2 n) \cdot 12 \\
= 17 + 12 \cdot \log_2 n
\]
Search in Sorted Arrays: Binary Search

We analyse the **best case complexity** of `binSearch`.

```
low = 0
high = n - 1
while low <= high do
    mid = ⌊(low + high)/2⌋
y = A[mid]
if x < y then high = mid - 1
if x == y then return true
if x > y then low = mid + 1
done
return false
```

Hence we have a **best case time complexity**:

\[ T(n) = 13 \]
Assumption (Random Access Machine):

- primitive operations are executed in constant time

Allows for analysis of worst, best and average case complexity.
(average analysis requires probability distribution for the inputs)

Disadvantages

- exact counting is cumbersome and error-prone
- in the real world:
  - different operations take different time
  - computers have different CPU/memory speeds
Estimating Running Time

Assume we have an algorithm that performs (input size $n$):
- $n^2$ additions, and
- $n$ multiplications.

Assume that we have computers $A$ and $B$ for which:
- $A$ needs $3s$ per addition and $7s$ per multiplication,
- $B$ needs $1s$ per addition and $10s$ per multiplication.

Then the algorithm runs on $A$ with time complexity:

$$T(n) = 3 \cdot n^2 + 7 \cdot n$$

and on computer $B$ with:

$$T(n) = 1 \cdot n^2 + 10 \cdot n$$

The constant factors may differ on different computers.
Estimating Running Time

Let

- $A$ be an algorithm,
- $T(n)$ the time complexity for Random Access Machines,
- $T_C(n)$ the time complexity on a (real) computer $C$.

For every computer $C$ there exist factors $a, b > 0$ such that:

$$a \cdot T(n) \leq T_C(n) \leq b \cdot T(n)$$

We can choose:

- $a =$ time for the fastest primitive operation on $C$
- $b =$ time for the slowest primitive operation on $C$

Thus idealized computation is precise up to constant factors:

- changing hardware/software affects only constant factors
The Role of Constant Factors

Problem size that can be solved in 1 hour:

<table>
<thead>
<tr>
<th></th>
<th>current computer</th>
<th>100× faster</th>
<th>10000× faster</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$N_1$</td>
<td>$100 \cdot N_1$</td>
<td>$10000 \cdot N_1$</td>
</tr>
<tr>
<td>$n^2$</td>
<td>$N_2$</td>
<td>$10 \cdot N_2$</td>
<td>$100 \cdot N_2$</td>
</tr>
<tr>
<td>$n^3$</td>
<td>$N_3$</td>
<td>$4.6 \cdot N_3$</td>
<td>$21.5 \cdot N_3$</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$N_4$</td>
<td>$N_4 + 7$</td>
<td>$N_4 + 13$</td>
</tr>
</tbody>
</table>

A 10000× faster computer is equal to:

- normal computer with 10000$h$ time, or
- 10000 times smaller constant factors in $T(n)$: $\frac{1}{10000} \cdot T(n)$.

The growth rate of running time (e.g. $n$, $n^2$, $n^3$, $2^n$,...) is more important than constant factors.
Methods for Analysing Time Complexity

**Experiments** (measurements on a certain machine):
- requires implementation
- comparisons require equal hardware and software
- experiments may miss out important inputs

**Calculation** of complexity for idealised computer model:
- requires exact counting
- computer model: Random Access Machine (RAM)

**Asymptotic complexity estimation** depending on input size $n$:
- allows for approximations
- linear ($n$), quadratic ($n^2$), exponential ($a^n$), ...
Asymptotic estimation

Magnitude of growth of complexity depending on input size \( n \).
Mostly used: upper bounds big-Oh-notation (worst case).

**Definition (Big-Oh-notation)**

Given functions \( f(n) \) and \( g(n) \), we say that \( f(n) \) is \( O(g(n)) \), denoted \( f(n) \in O(g(n)) \), if there exist \( c, n_0 \) such that:

\[
\forall n \geq n_0 : f(n) \leq c \cdot g(n)
\]

In words: the growth rate of \( f(n) \) is \( \leq \) to the growth rate of \( g(n) \).

**Example (\( 2n + 1 \in O(n) \))**

We need to find \( c, n_0 \) such that for all \( n \geq n_0 \) we have

\[
2n + 1 \leq c \cdot n
\]

We choose \( c = 3, n_0 = 1 \). Then \( 3 \cdot n = 2n + n \geq 2n + 1 \).
Examples

**Example \((7n − 2 ∈ O(n))\)**

We need to find \(c, n_0\) such that for all \(n ≥ n_0\) we have

\[
7n − 2 ≤ c \cdot n
\]

We choose \(c = 7, n_0 = 1\).

**Example \((3n^3 + 5n^2 + 2 ∈ O(n^3))\)**

We need to find \(c, n_0\) such that for all \(n ≥ n_0\) we have

\[
3n^3 + 5n^2 + 2 ≤ c \cdot n^3
\]

We choose \(c = 4, n_0 = 7\), then

\[
4 \cdot n^3 = 3n^3 + n^3 ≥ 3n^3 + 7n^2
\]

\[
= 3n^3 + 5n^2 + 2n^2 ≥ 3n^3 + 5 \cdot n^2 + 2
\]
Examples

**Example (7 ∈ O(1))**

We need to find \( c, n_0 \) such that for all \( n \geq n_0 \) we have

\[
7 \leq c \cdot 1
\]

Take \( c = 7, n_0 = 1 \).

In general: \( O(1) \) means ‘constant time’.

**Example (n^2 \notin O(n))**

Assume there would exist \( c, n_0 \) such that for all \( n \geq n_0 \) we have

\[
n^2 \leq c \cdot n
\]

Take an arbitrary \( n \geq c + 1 \), then

\[
n^2 \geq (c + 1) \cdot n > c \cdot n
\]

This contradicts \( n^2 \leq c \cdot n \).
Big-Oh Rules

- If \( f(n) \) is a polynomial of degree \( d \), then \( f(n) \in O(n^d) \).
- More general, we can drop:
  - constant factors, and
  - lower order terms: (\( \succ \) means higher order/growth rate)

\[
3^n \succ 2^n \succ n^5 \succ n^2 \succ n \cdot \log_2 n \succ n \succ \log_2 n \succ \log_2 \log_2 n
\]

Example

\[
5n^6 + 3n^2 + 2n^7 + n \in O(n^7)
\]

Example

\[
2^{200} + 3 \cdot 2^n + 50n^3 + \log_2 n \in O(2^n)
\]
Asymptotic estimation: \textit{arrayMax}

We analyse the \textbf{worst case complexity} of \textit{arrayMax}.

Algorithm \textit{arrayMax}(A, n):
\begin{itemize}
  \item \textbf{Input:} An array storing \(n \geq 1\) integers.
  \item \textbf{Output:} The maximum element in \(A\).
\end{itemize}

\begin{verbatim}
currentMax = A[0]
for i = 1 to n - 1 do
  if currentMax < A[i] then
    currentMax = A[i]

done
return currentMax
\end{verbatim}

Hence we have a (worst case) \textbf{time complexity}:
\[
T(n) \in O(1 + (n - 1) \cdot 1 + 1)
\]
\[
= O(n)
\]
Asymptotic estimation: \textit{prefixAverage}

We analyse the \textbf{complexity} of \textit{prefixAverage}.

\textbf{Algorithm} \textit{prefixAverage}(\textit{A}, \textit{n}):

\textbf{Input:} An array storing \( n \geq 1 \) integers.
\textbf{Output:} An array \textit{B} of length \( n \) such that for all \( i < n \):

\[ B[i] = \frac{1}{i+1} \cdot (A[0] + A[1] + \ldots + A[i]) \]

\( B = \) new array of length \( n \)
for \( i = 0 \) to \( n - 1 \) do
\hspace{1em} \textit{sum} = 0
\hspace{1em} for \( j = 0 \) to \( i \) do
\hspace{2em} \textit{sum} = \textit{sum} + \textit{A}[\textit{j}]
\hspace{1em} done
\hspace{1em} \textit{B}[\textit{i}] = \textit{sum}/(\textit{i} + 1)
\hspace{1em} done
return \( \textit{B} \)

The (worst case) \textbf{time complexity} is: \( T(n) \in O(n^2) \)
We analyse the **complexity** of `prefixAverage2`.

**Algorithm** `prefixAverage2(A, n)`:  
**Input:** An array storing $n \geq 1$ integers.  
**Output:** An array $B$ of length $n$ such that for all $i < n$:

$$B[i] = \frac{1}{i+1} \cdot (A[0] + A[1] + \ldots + A[i])$$

$n$

1

1

1

$$B = \text{new array of length } n$$

$sum = 0$

1

$n \cdot 1$

1

1

1

The (worst case) **time complexity** is:

$$T(n) \in O(n)$$
Efficiency for Small Problem Size

Asymptotic complexity (big-$O$-notation) speaks about big $n$:

▶ For small $n$, an algorithm with good asymptotic complexity can be slower than one with bad complexity.

**Example**

We have

▶ $1000 \cdot n \in O(n)$ is asymptotically better than $n^2 \in O(n^2)$

but for $n = 10$:

▶ $1000 \cdot n = 10000$ is much slower than $n^2 = 100$
Relatives of Big-Oh

Lower bounds: big-Omega-notation.

Definition (Big-Omega-notation)

\( f(n) \) is \( \Omega(g(n)) \), denoted \( f(n) \in \Omega(g(n)) \), if there exist \( c, n_0 \) s.t.:

\[
\forall n \geq n_0 : f(n) \geq c \cdot g(n)
\]

Exact bound (lower and upper): big-Theta-notation.

Definition (Big-Theta-notation)

\( f(n) \) is \( \Theta(g(n)) \), denoted \( f(n) \in \Theta(g(n)) \) if:

\[
f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n))
\]
### Definition

An **abstract data type** (ADT) is an abstraction of a data type. An ADT specifies:

- data stored
- operations on the data
- error conditions associated with operations

The choice of data types is important for efficient algorithms.
The Stack ADT

- stores arbitrary objects: **last-in first-out principle (LIFO)**
- main operations:
  - `push(o)`: insert the object `o` at the top of the stack
  - `pop()`: removes & returns the object on the top of the stack. An error occurs (exception is thrown) if the stack is empty.

- auxiliary operations:
  - `isEmpty()`: indicates whether the stack is empty
  - `size()`: returns number of elements in the stack
  - `top()`: returns the top element without removing it. An error occurs if the stack is empty.
Applications of Stacks

Direct applications:
- page-visited history in a web browser
- undo-sequence in a text editor
- chain of method calls in the Java Virtual Machine

Indirect applications:
- auxiliary data structure for algorithms
- component of other data structures
Stack in the Java Virtual Machine (JVM)

JVM keeps track of the chain of active methods with a stack:

- stores local variables and program counter

```java
public class Example {
    public static void main() {
        int i = 5;
        foo(i);
    }

    public static void foo(int j) {
        int k;
        k = j + 1;
        bar(k);
    }

    public static void bar(int m) {
        ...  
    }
}
```

- Frame for bar
  - PC = 1
  - m = 6

- Frame for foo
  - PC = 3
  - j = 5
  - k = 6

- Frame for main
  - PC = 2
  - i = 5
we fix the maximal size $N$ of the stack in advance

elements are added to the array from left to right

variable $t$ points to the top of the stack

initially $t = 0$, the stack is empty
Array-based Implementation of Stacks

\textbf{size()}: \\
\textbf{return} \ t + 1

\textbf{push(o)}: \\
\textbf{if} \ \textbf{size()} == \ N \ \textbf{then} \\
\quad \textbf{throw} \ \text{FullStackException}
\textbf{else} \\
\quad t = t + 1
\quad S[t] = o

(throws an exception if the stack is full)
Array-based Implementation of Stacks

pop():
    if size() == 0 then
        throw EmptyStackException
    else
        o = S[t]
        t = t - 1
    return o

(throws an exception if the stack is empty)
Array-based Implementation of Stacks, Summary

Performance:
- every operation runs in $O(1)$

Limitations:
- maximum size of the stack must be chosen a priori
- trying to add more than $N$ elements causes an exception

These limitations do not hold for stacks in general:
- only for this specific implementation
The Queue ADT

- stores arbitrary objects: **first-in first-out principle (FIFO)**
- main operations:
  - enqueue(o): insert the object o at the end of the queue
  - dequeue(): removes & returns the object at the beginning of the queue. An error occurs if the queue is empty.

- auxiliary operations:
  - isEmpty(): indicates whether the queue is empty
  - size(): returns number of elements in the queue
  - front(): returns the first element without removing it. An error occurs if the queue is empty.
Applications of Queues

Direct applications:
- waiting lists (bureaucracy)
- access to shared resources (CPU, printer, ...)

Indirect applications:
- auxiliary data structure for algorithms
- component of other data structures
we fix the maximal size $N - 1$ of the queue in advance

uses array of size $N$ in circular fashion:

- variable $f$ points to the front of the queue
- variable $r$ points one behind the rear of the queue (that is, array location $r$ is kept empty)
Array-based Implementation of Queues

size():
    return \((N + r - f) \mod N\)

isEmpty():
    return size() == 0
Array-based Implementation of Queues

enqueue(o):
    if size() == N - 1 then
        throw FullQueueException
    else
        Q[r] = o
        r = (r + 1) mod N
    (throws an exception if the queue is full)
Array-based Implementation of Queues

dequeue():

if size() == 0 then
    throw EmptyQueueException
else
    o = Q[f]
    f = (f + 1) mod N
return o

(throws an exception if the queue is empty)
What is a Tree?

- stores elements in a hierarchical structure
- consists of nodes in parent-child relation
- children are ordered (we speak about ordered trees)

Applications of trees:
- file systems, databases
- organization charts
Tree Terminology

- **Root**: node without parent (A)
- **Inner node**: at least one child (A, B, C, D, F)
- **Leaf (external node)**: no children (H, I, E, J, G)
- **Depth of a node**: length of the path to the root
- **Height of the tree**: maximum depth of any node (3)

![Tree Diagram]

- **Ancestors**:
  - A is parent of C
  - B is grandparent of H
- **Descendant**:
  - C is child of A
  - H is grandchild of B
- **Substree**:
  - node + all descendants
The Tree ADT

- **Accessor operations:**
  - `root()`: returns root of the tree
  - `children(v)`: returns the list of children of node `v`
  - `parent(v)`: returns parent of node `v`. An error occurs (exception is thrown) if `v` is root.

- **Generic operations:**
  - `size()`: returns number of nodes in the tree
  - `isEmpty()`: indicates whether the tree is empty
  - `elements()`: returns set of all the nodes of the tree
  - `positions()`: returns set of all positions of the tree.

- **Query methods:**
  - `isInternal(v)`: indicates whether `v` is an inner node
  - `isExternal(v)`: indicates whether `v` is a leaf
  - `isRoot(v)`: indicates whether `v` is the root node
A traversal visits the nodes of a tree in systematic manner.

In a **preorder traversal**:
- a node is visited before its descendants.

```plaintext
preOrder(v):
  visit(v)
  for each child w of v do
    preOrder(w)
```

Preorder Traversal

- A
- B
- C
- D
- E
- F
- G
- H
- I
- J
In a **postorder traversal**:  
- a node is visited after its descendants.

**postOrder**(v):
  
  for each child w of v do
  
  postOrder(w)

  visit(v)
A **binary tree** is a tree with the property:
- each inner node has exactly two children

We call the children of inner nodes **left** and **right child**.

Applications of binary trees:
- arithmetic expressions
- decision processes
- searching
Binary trees can represent arithmetic expressions:
▶ inner nodes are operators
▶ leafs are operands

Example: \((3 \times (a - 2)) + (b/4)\)

(postorder traversal can be used to evaluate an expression)
Binary Decision Trees

Binary tree can represent a decision process:

- inner nodes are questions with yes/no answers
- leafs are decisions

Example:

```
Question A?
  yes
  Question B?
  yes
  Result 1
  no
  Question C?
  yes
  Result 2
  no
  Result 3
no
  Question D?
  yes
  Result 4
  no
  Result 5
```
Properties of Binary Trees

Notation:
- \( n \) number of nodes
- \( i \) number of internal nodes
- \( e \) number of leafs
- \( h \) height

Properties:
- \( e = i + 1 \)
- \( n = 2e - 1 \)
- \( h \leq i \)
- \( h \leq (n - 1)/2 \)
- \( e \leq 2^h \)
- \( h \geq \log_2 e \)
- \( h \geq \log_2(n + 1) - 1 \)
Binary Tree ADT

- Inherits all methods from Tree ADT.
- Additional methods:
  - `leftChild(v)`: returns the left child of `v`
  - `rightChild(v)`: returns the right child of `v`
  - `sibling(v)`: returns the sibling of `v`

(exceptions are thrown if left/right child or sibling do not exist)
In a **inorder traversal**:

- a node is visited after its left and before its right subtree.

**Application:** drawing binary trees

- \( x(v) = \) inorder rank \( v \)
- \( y(v) = \) depth of \( v \)

**inOrder(\( v \)):**

\[
\text{if } \text{isInternal}(v) \text{ then } \\
\text{inOrder(leftChild}(v)) \\
\text{visit}(v) \\
\text{if } \text{isInternal}(v) \text{ then } \\
\text{inOrder(rightChild}(v))
\]
A specialization of the inorder traversal for printing expressions:

\[
\text{printExpression}(v): \\
\text{if isInternal}(v) \text{ then} \\
\quad \text{print('(')} \\
\quad \text{inOrder(leftChild}(v)) \\
\quad \text{print}(v.\text{element}()) \\
\text{if isInternal}(v) \text{ then} \\
\quad \text{inOrder(rightChild}(v)) \\
\quad \text{print(')')}
\]

Example: \(\text{printExpression}(v)\) of the above tree yields

\[
((3 \times (a - 2)) + (b/4))
\]
Postorder traversal for evaluating arithmetic expressions:

eval(ν):
    if isExternal(ν) then
        return ν.element()
    else
        x = eval(leftChild(ν))
        y = eval(rightChild(ν))
        ◊ = operator stored at ν
        return x ◊ y
Euler Tour Traversal

- Generic traversal of a binary tree:
  - preorder, postorder, inorder are special cases
- Walk around the tree, each node is visited tree times:
  - on the left (preorder)
  - from below (inorder)
  - on the right (postorder)
A Vector stores a list of elements:

- Access via rank/index.

Accessor methods:
- `elementAtRank(r)`

Update methods:
- `replaceAtRank(r, o)`
- `insertAtRank(r, o)`
- `removeAtRank(r)`

Generic methods:
- `size()`
- `isEmpty()`

Here `r` is of type integer, `n, m` are nodes, `o` is an object (data).
List ADT

A List consists of a sequence of nodes:

▶ The nodes store arbitrary objects.
▶ Access via before/after relation between nodes.

element 1  element 2  element 3  element 4

Accessor methods:
  first(), last(), before(n), after(n)
Query methods:
  isFirst(n), isLast(n)
Generic methods:
  size(), isEmpty()
Update methods:
  replaceElement(n, o),
  swapElements(n, m),
  insertBefore(n, o),
  insertAfter(n, o),
  insertFirst(o),
  insertLast(o),
  remove(n)

Here \( r \) is of type integer, \( n, m \) are nodes, \( o \) is an object (data).
Sequence ADT

- Sequence ADT is the union of Vector and List ADT:
  - inherits all methods
- Additional methods:
  - \texttt{atRank}(r): returns the node at rank $r$
  - \texttt{rankOf}(n): returns the rank of node $n$
- Elements can be accessed by:
  - rank/index, or
  - navigation between nodes

Remarks

The distinction between Vector, List and Sequence is artificial:
- every element in a list naturally has a rank

Important are the different access methods:
- access via rank/index, or navigation between nodes
Singly Linked List

- A singly linked list provides an implementation of List ADT.
- Each node stores:
  - element (data)
  - link to the next node
- Variables \textit{first} and optional \textit{last} point to the first/last node.
- Optional variable \textit{size} stores the size of the list.
Singly Linked List: size

If we store the size on the variable \( \text{size} \), then:

```python
size():
    return size
```

Running time: \( O(1) \).

If we do not store the size on a variable, then:

```python
size():
    s = 0
    m = first
    while m != ∅ do
        s = s + 1
        m = m.next
    done
    return s
```

Running time: \( O(n) \).
Singly Linked List: insertAfter

insertAfter(n, o):
\[
x = \text{new node with element } o
\]
\[
x.next = n.next
\]
\[
n.next = x
\]
\[
\text{if } n == \text{last then } \text{last} = x
\]
\[
\text{size} = \text{size} + 1
\]

Running time: \(O(1)\).
Singly Linked List: insertFirst

insertFirst(o):

\[
x = \text{new node with element } o
\]

\[
x.next = \text{first}
\]

\[
\text{first} = x
\]

\[
\text{if } \text{last} == \emptyset \text{ then } \text{last} = x
\]

\[
\text{size} = \text{size} + 1
\]

Running time: \(O(1)\).
Singly Linked List: `insertBefore`

`insertBefore(n, o):`

```
if n == first then
    insertFirst(o)
else
    m = first
    while m.next != n do
        m = m.next
    done
    insertAfter(m, o)
```

(error check \( m \neq \emptyset \) has been left out for simplicity)

We need to find the node \( m \) before \( n \):

- requires a search through the list

Running time: \( O(n) \) where \( n \) the number of elements in the list.
(worst case we have to search through the whole list)
<table>
<thead>
<tr>
<th>Operation</th>
<th>Worst case Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, isEmpty</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>first, last, after</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>before</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>replaceElement, swapElements</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertFirst, insertLast</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertAfter</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertBefore</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>atRank, rankOf, elemAtRank</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>replaceAtRank</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>insertAtRank, removeAtRank</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

1. *size* needs $O(n)$ if we do not store the size on a variable.
2. *last* and *insertLast* need $O(n)$ if we have no variable *last*.
3. *remove*($n$) runs in best case in $O(1)$ if $n == first$. 
We can implement a stack with a singly linked list:

- top element is stored at the first node

Each stack operation runs in $O(1)$ time:

- `push(o)`:
  
  `insertFirst(o)`

- `pop()`:
  
  $o = \text{first()}.element$
  
  `remove(\text{first()})`

  `return o`
Queue with a Singly Linked List

We can implement a queue with a singly linked list:

- front element is stored at the first node
- rear element is stored at the last node

Each queue operation runs in $O(1)$ time:

- $\text{enqueue}(o)$:
  
  \[ \text{insertLast}(o) \]

- $\text{dequeue}()$:
  
  \[
  o = \text{first().element} \\
  \text{remove}(\text{first()}) \\
  \text{return } o
  \]
A doubly linked list provides an implementation of List ADT.

Each node stores:
- element (data)
- link to the next node
- link to the previous node

Special header and trailer nodes.

Variable size storing the size of the list.
Doubly Linked List: insertAfter

`insertAfter(n, o)`:

- `m = n.next`
- `x = new node with element o`
- `x.prev = n`
- `x.next = m`
- `n.next = x`
- `m.prev = x`
- `size = size + 1`
Doubly Linked List: insertBefore

insertBefore\((n, o)\):
insertAfter\((n.prev, o)\)
Doubly Linked List: remove

remove(n):

\[ p = n.prev \]
\[ q = n.next \]
\[ p.next = q \]
\[ q.prev = p \]
\[ size = size - 1 \]
### Doubly Linked List: Performance

<table>
<thead>
<tr>
<th>Operation</th>
<th>Worst case Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, isEmpty</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>first, last, after</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>before</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>replaceElement, swapElements</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertFirst, insertLast</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertAfter</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertBefore</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>atRank, rankOf, elemAtRank</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>replaceAtRank</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>insertAtRank, removeAtRank</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Now all operations of List ADT are $O(1)$:

- only operations accessing via index/rank are $O(n)$
**Amortization** is a tool for understanding algorithm complexity if steps have widely varying performance:

- analyses running time of a series of operations
- takes into account interaction between different operations

The **amortized running time** of an operation in a series of operations is the worst-case running time of the series divided by the number of operations.

Example:

- $N$ operations in 1s,
- 1 operation in $Ns$

Amortized running time:

$$\frac{N \cdot 1 + N}{N + 1} \leq 2$$

That is: $O(1)$ per operation.
Amortization Techniques: Accounting Method

The **accounting method** uses a scheme of credits and debits to keep track of the running time of operations in a sequence:

- we pay one cyber-dollar for a constant computation time

We charge for every operation some amount of cyber-dollars:

- this amount is the amortized running time of this operation

When an operation is executed we need enough cyber-dollars to pay for its running time.

We charge:

- 2 cyber-dollars per operation

The first $N$ operations consume 1 dollar each. For the last operation we have $N$ dollars saved.

**Amortized $O(1)$ per operation.**
Clearable Table supports operations:
- \( \text{add}(o) \) for adding an object to the table
- \( \text{clear}() \) for emptying the table

Implementation using an array:
- \( \text{add}(o) \in O(1) \)
- \( \text{clear}() \in O(m) \) where \( m \) is number of elements in the table
  (\( \text{clear}() \) removes one by one every entry in the table)
Amortization, Example: Clearable Table

We charge 2 per operation:

- add consumes 1; thus we save 1 per add.

Whenever \( m \) elements are in the list, we have saved \( m \) dollars:

- we use the \( m \) dollars to pay for clear \( \in O(m) \)

Thus amortized costs per operation are 2 dollars, that is, \( O(1) \).
Array-based Implementation of Lists

Uses array in circular fashion (as for queues):

- variable $f$ points to first position
- variable $l$ points one behind last position

Adding elements runs in $O(1)$ as long as the array is not full.

When the array is full:

- we need to allocate a larger array $B$
- copy all elements from $A$ to $B$
- set $A = B$ (that is, from then on work further with $B$)
Array-based Implementation: Constant Increase

Each time the array is full we increase the size by \( k \) elements:

- creating \( B \) of size \( n + k \) and copying \( n \) elements is \( O(n) \)

Example: \( k = 3 \)

![Diagram showing time vs. operations for array resizing]

Every \( k \)-th insert operation we need to resize the array.

Worst-case complexity for a sequence of \( m \) insert operations:

\[
\sum_{i=1}^{\lfloor m/k \rfloor} k \cdot i \in O(n^2)
\]

Average costs for each operation in the sequence: \( O(n) \).
Array-based Implementation: Doubling the Size

Each time the array is full we double the size:

- creating $B$ of size $2n$ and copying $n$ elements is $O(n)$
Each time the array is full we double the size:
  ▶ **insert** costs $O(1)$ if the array is not full
  ▶ **insert** costs $O(n)$ if the array is full (for resize $n \mapsto 2n$)

We charge 3 cyber dollars per **insert**.

After doubling the size $n \mapsto 2n$:
  ▶ we have at least $n$ times *insert*’s without resize
  ▶ we save $2n$ cyber-dollars before the next resize

Thus we have $2n$ cyber-dollars for the next resize $2n \mapsto 4n$.

Hence the amortized costs for insert is 3: $O(1)$.

(can also be used for fast queue and stack implementations)
The performance under assumption of the doubling strategy.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Amortized Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, isEmpty</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>first, last, after</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>before</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>replaceElement, swapElements</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertFirst, insertLast</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertAfter</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>insertBefore</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>atRank, rankOf, elemAtRank</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>replaceAtRank</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertAtRank, removeAtRank</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Comparison of the Amortized Complexities

<table>
<thead>
<tr>
<th>Operation</th>
<th>Singly</th>
<th>Doubly</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, isEmpty</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>first, last, after</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>before</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>replaceElement, swapElements</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertFirst, insertLast</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertAfter</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>insertBefore</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>atRank, rankOf, elemAtRank</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>replaceAtRank</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertAtRank, removeAtRank</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

- singly linked lists (Singly)
- doubly linked lists (Doubly)
- array-based implementation with doubling strategy (Array)
Iterators

- Iterator allows to traverse the elements in a list or set.
- Iterator ADT provides the following methods:
  - `object()`: returns the current object
  - `hasNext()`: indicates whether there are more elements
  - `nextObject()`: goes to the next object and returns it
A priority queue stores a list of items:

- the items are pairs \((key, element)\)
- the \textit{key} represents the priority: \textit{smaller key} = \textit{higher priority}

Main methods:

- \texttt{insertItem}(k, o): inserts element \(o\) with key \(k\)
- \texttt{removeMin}(): removes item \((k, o)\) with smallest key and returns its element \(o\)

Additional methods:

- \texttt{minKey}(): returns but does not remove the smallest key
- \texttt{minElement}(): returns but does not remove the element with the smallest key
- \texttt{size}(), \texttt{isEmpty}()
Applications of Priority Queues

Direct applications:
- bandwidth management in routers
- task scheduling (execution of tasks in priority order)

Indirect applications:
- auxiliary data structure for algorithms, e.g.:
  - shortest path in graphs
- component of other data structures
Priority Queues: Order on the Keys

- Keys can be arbitrary objects with a total order.
- Two distinct items in a queue may have the same key.

A relation $\leq$ is called order if for all $x, y, z$:

- reflexivity: $x \leq x$
- transitivity: $x \leq y$ and $y \leq z$ implies $x \leq z$
- antisymmetry: $x \leq y$ and $y \leq x$ implies $x = y$

Implementation of the order: external class or function

- Comparator ADT
  - isLessThan($x, y$), isLessOrEqualTo($x, y$)
  - isEqual($x, y$)
  - isGreaterThan($x, y$), isGreaterOrEqualTo($x, y$)
  - isComparable($x$)
We can use a priority queue for sorting as follows:

- Insert all elements $e$ stepwise with $\text{insertItem}(e, e)$
- Remove the elements in sorted order using $\text{removeMin}()$

**Algorithm** $\text{PriorityQueueSort}(A, C)$:

**Input:** List $A$, Comparator $C$

**Output:** List $A$ sorted in ascending order

1. $P = \text{new priority queue with comparator } C$
2. **while** $\neg A.\text{isEmpty}()$ **do**
   1. $e = A.\text{remove}(A.\text{first}())$
   2. $P.\text{insertItem}(e, e)$
3. **while** $\neg P.\text{isEmpty}()$ **do**
   1. $A.\text{insertLast}(P.\text{removeMin}())$

Running time depends on:

- implementation of the priority queue
List-based Priority Queues

Implementation with an **unsorted List**:

![List with keys 5, 7, 3, 4, 1]

- **Performance:**
  - *insertItem* is $O(1)$ since we can insert in front or end
  - *removeMin, minKey* and *minElement* are $O(n)$ since we have to search the whole list for the smallest key

Implementation with an **sorted List**:

![List with keys 1, 3, 4, 5, 7]

- **Performance:**
  - *insertItem* is $O(n)$
    - for singly/doubly linked list $O(n)$ for finding the insert position
    - for array-based list $O(n)$ for *insertAtRank*
  - *removeMin, minKey* and *minElement* are $O(1)$ since the smallest key is in front of the list
Given is a sequence of pairs

\[(k_1, e_1), (k_2, e_2), \ldots, (k_n, e_n)\]

of elements \(e_i\) with keys \(k_i\) and an order \(\leq\) on the keys.

We search a permutation \(\pi\) of the pairs such that the keys \(k_{\pi(1)} \leq k_{\pi(2)} \leq \ldots \leq k_{\pi(n)}\) are in ascending order.
Selection-sort takes an unsorted list \( A \) and sorts as follows:

- search the smallest element \( A \) and swap it with the first
- afterwards continue sorting the remainder of \( A \)
Selection-Sort: Properties

- Time complexity (best, average and worst-case) $O(n^2)$:
  
  $$n + (n - 1) + \ldots + 1 = \frac{n^2 + n}{2} \in O(n^2)$$

  (caused by searching the minimal element)

- Selection-sort is an in-place sorting algorithm.

In-Place Algorithm

An algorithm is **in-place** if apart from space for input data only a constant amount of space is used: space complexity $O(1)$. 
Stable Sorting Algorithms

A sorting algorithm is called **stable** if the order of items with equal key is preserved.

**Example: not stable**

B, C exchanged order, although they have equal keys

**Example: stable**
Applications of stable sorting:

- preserving original order of elements with equal key

For example:

- we have an alphabetically sorted list of names
- we want to sort by date of birth while keeping alphabetical order for persons with same birthday

Selection-sort is stable if

- we always select the first (leftmost) minimal element
Selection-sort takes an unsorted list $A$ and sorts as follows:

- distinguishes a sorted and unsorted part of $A$
- in each step we remove an element from the unsorted part, and insert it at the correct position in the sorted part
Insertion-Sort: Complexity

- Time complexity worst-case $O(n^2)$:
  
  $$1 + 2 + \ldots + n = \frac{n^2 + n}{2} \in O(n^2)$$

  (searching insertion position together with inserting)

- Time complexity best-case $O(n)$:
  
  - if list is already sorted, and
  - we start searching insertion position from the end

- More general: time complexity $O(n \cdot (n - d + 1))$
  
  - if the first $d$ elements are already sorted

- Insertion-sort is an in-place sorting algorithm:
  
  - space complexity $O(1)$
Insertion-Sort: Properties

- Simple implementation.
- Efficient for:
  - small lists
  - big lists of which a large prefix is already sorted
- Insertion-sort is stable if:
  - we always pick the first element from the unsorted part
  - we always insert behind all elements with equal keys
- Insertion-sort can be used online:
  - does not need all data at once,
  - can sort a list while receiving it
A heap is a binary tree storing keys at its inner nodes such that:

- if $A$ is a parent of $B$, then $\text{key}(A) \leq \text{key}(B)$
- the heap is a complete binary tree: let $h$ be the heap height
  - for $i = 0, \ldots, h - 1$ there are $2^i$ nodes at depth $i$
  - at depth $h - 1$ the inner nodes are left of the external nodes

We call the rightmost inner node at depth $h - 1$ the ‘last node’.
**Theorem**

A heap storing $n$ keys height $O(\log_2 n)$.

**Proof.**

A heap of height $h$ contains $2^i$ nodes at every depth $i = 0, \ldots, h - 2$ and at least one node at depth $h - 1$. Thus $n \geq 1 + 2 + 4 + \ldots + 2^{h-2} + 1 = 2^{h-1}$. Hence $h \leq 1 + \log_2 n$. \qed
We can use a heap to implement a priority queue:

- inner nodes store \((key, element)\) pair
- variable \(last\) points to the last node

For convenience, in the sequel, we only show the keys.
Heaps: Insertion

The insertion of key \( k \) into the heap consists of 3 steps:

- Find the insertion node \( z \) (the new last node).
- Expand \( z \) to an internal node and store \( k \) at \( z \).
- Restore the heap-order property (see following slides).

Example: insertion of key 1 (without restoring heap-order)
Heaps: Insertion, Upheap

After insertion of $k$ the heap-order may be violated.

We restore the heap-order using the `upheap` algorithm:

- we swap $k$ upwards along the path to the root as long as the parent of $k$ has a larger key

Time complexity is $O(\log_2 n)$ since the heap height is $O(\log_2 n)$. 

![Heap diagram](image-url)
After insertion of $k$ the heap-order may be violated. We restore the heap-order using the **upheap** algorithm:

- we swap $k$ upwards along the path to the root as long as the parent of $k$ has a larger key

Time complexity is $O(\log_2 n)$ since the heap height is $O(\log_2 n)$.

Now the heap-order property is restored.
An algorithm for finding the insertion position (new last node):
▶ start from the current last node
▶ while the current node is a right child, go to the parent node
▶ if the current node is a left child, go to the right child
▶ while the current node has a left child, go to the left child

Time complexity is $O(\log_2 n)$ since the heap height is $O(\log_2 n)$. (we walk at most at most once completely up and down again)
The removal of the root consists of 3 steps:

- Replace the root key with the key of the last node $w$.
- Compress $w$ and its children into a leaf.
- Restore the heap-order property (see following slides).

Example: removal of the root (without restoring heap-order)
Replaces the root key by $k$ may violate the heap-order. We restore the heap-order using the `downheap` algorithm:

- we swap $k$ with its smallest child as long as a child of $k$ has a smaller key.

Time complexity is $O(\log_2 n)$ since the heap height is $O(\log_2 n)$. 
Replacements the root key by $k$ may violate the heap-order.

We restore the heap-order using the downheap algorithm:

- we swap $k$ with its smallest child as long as a child of $k$ has a smaller key

Time complexity is $O(\log_2 n)$ since the heap height is $O(\log_2 n)$.

Now the heap-order property is restored. The new last node can be found similar to finding the insertion position (but now walk against the clock direction).
Heaps: Removal, Finding the New Last Node

After the removal we have to find the new last node:

- start from the old last node (which has been remove)
- while the current node is a left child, go to the parent node
- if the current node is a right child, go to the left child
- while the current node has an right child which is not a leaf, go to the right child

Time complexity is $O(\log_2 n)$ since the heap height is $O(\log_2 n)$.
(we walk at most at most once completely up and down again)
Heap-Sort

We implement a priority queue by means of a heap:
- \texttt{insertItem}(k, e) corresponds to adding \((k, e)\) to the heap
- \texttt{removeMin()} corresponds to removing the root of the heap

Performance:
- \texttt{insertItem}(k, e), and \texttt{removeMin()} run in \(O(\log_2 n)\) time
- \texttt{size()}, \texttt{isEmpty()}, \texttt{minKey()}, and \texttt{minElement()} are \(O(1)\)

Heap-sort is \(O(n \log_2 n)\)

Using a heap-based priority queue we can sort a list of \(n\) elements in \(O(n \cdot \log_2 n)\) time (\(n\) times insert + \(n\) times removal).

Thus heap-sort is much faster than quadratic sorting algorithms (e.g. selection sort).
We can represent a heap with $n$ keys by a vector of size $n + 1$:

- The root node has rank 1 (cell at rank 0 is not used).
- For a node at rank $i$:
  - the left child is at rank $2i$
  - the right child is at rank $2i + 1$
  - the parent (if $i > 1$) is located at rank $\lfloor i/2 \rfloor$
- Leaf nodes and links between the nodes are not stored explicitly.
We can represent a heap with $n$ keys by a vector of size $n + 1$:

- The last element in the heap has rank $n$, thus:
  - `insertItem` corresponds to inserting at rank $n + 1$
  - `removeMin` corresponds to removing at rank $n$
- Yields in-place heap-sort (space complexity $O(1)$):
  - uses a max-heap (largest element on top)
Merging two Heaps

We are given two heaps $h_1, h_2$ and a key $k$:

- create a new heap with root $k$ and $h_1, h_2$ as children
- we perform downheap to restore the heap-order
Merging two Heaps

We are given two heaps $h_1, h_2$ and a key $k$:

- create a new heap with root $k$ and $h_1, h_2$ as children
- we perform downheap to restore the heap-order
Bottom-up Heap Construction

We have $n$ keys and want to construct a heap from them.

Possibility one:
- start from empty heap and use $n$ times insert
- needs $O(n \log_2 n)$ time

Possibility two: bottom-up heap construction
- for simplicity we assume $n = 2^h - 1$ (for some $h$)
- take $2^{h-1}$ elements and turn them into heaps of size 1
- for phase $i = 1, \ldots, \log_2 n$:
  - merge the heaps of size $2^i - 1$ to heaps of size $2^{i+1} - 1$
We construct a heap from the following $2^4 - 1 = 15$ elements:

16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10
We construct a heap from the following $2^4 - 1 = 15$ elements:

16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10
Bottom-up Heap Construction, Example

We construct a heap from the following $2^4 - 1 = 15$ elements:

$$16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10$$
We construct a heap from the following $2^4 - 1 = 15$ elements:

16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10
We construct a heap from the following $2^4 - 1 = 15$ elements:

$16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10$
We construct a heap from the following $2^4 - 1 = 15$ elements:

$16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10$
We construct a heap from the following $2^4 - 1 = 15$ elements:

16, 15, 4, 12, 6, 9, 23, 20, 25, 5, 11, 27, 7, 8, 10

We are ready: this is the final heap.
Bottom-up Heap Construction, Performance

Visualization of the worst-case of the construction:

- displays the longest possible heapdown paths (may not be the actual path, but maximal length)
- each edge is traversed at most once
- we have $2n$ edges hence the time complexity is $O(n)$
- faster than $n$ successive insertions
Dictionary ADT models:

- a searchable collection of key-element items.
- search via the key

Main operations are: searching, inserting, deleting items

- **findElement**(*k*): returns the element with key *k* if the dictionary contains such an element (otherwise returns special element `No_Such_Key`)
- **insertItem**(*k*, *e*): inserts (*k*, *e*) into the dictionary
- **removeElement**(*k*): like **findElement**(*k*) but additionally removes the item if present
- **size**(), **isEmpty**()
- **keys**(), **elements**()
A log file is a dictionary implemented based on an unsorted list:

- **Performance:**
  - `insertItem` is $O(1)$ (can insert anywhere, e.g. end)
  - `findElement`, and `removeElement` are $O(n)$
    (have to search the whole sequence in worst case)

- Efficient for small dictionaries or if insertions are much more frequent than search and removal.
  (e.g. access log of a workstation)
Ordered Dictionaries:

- Keys are assumed to come from a total order.
- New operations:
  - closestKeyBefore($k$)
  - closestElementBefore($k$)
  - closestKeyAfter($k$)
  - closestElementAfter($k$)
Binary Search

Binary search performs $\text{findElement}(k)$ on sorted arrays:

- each step the search space is halved
- time complexity is $O(\log_2 n)$
- for pseudo code and complexity analysis see lecture 1

Example: $\text{findElement}(7)$

```
0 1 3 5 7 8 9 11 15 16 18
```

$low$ $high$
Binary search performs \texttt{findElement}(k) on sorted arrays:

- each step the search space is halved
- time complexity is $O(\log_2 n)$
- for pseudo code and complexity analysis see lecture 1

Example: \texttt{findElement}(7)
Binary search performs \texttt{findElement}(k) on sorted arrays:

- each step the search space is halved
- time complexity is $O(\log_2 n)$
- for pseudo code and complexity analysis see lecture 1

Example: \texttt{findElement}(7)

```
0 1 3 5 7 8 9 11 15 16 18
```

```
0 1 3 5 7 8 9 11 15 16 18
```

```
0 1 3 5 7 8 9 11 15 16 18
```
Binary search performs $\text{findElement}(k)$ on sorted arrays:

- each step the search space is halved
- time complexity is $O(\log_2 n)$
- for pseudo code and complexity analysis see lecture 1

Example: $\text{findElement}(7)$
Binary Search

Binary search performs $\text{findElement}(k)$ on sorted arrays:

- each step the search space is halved
- time complexity is $O(\log_2 n)$
- for pseudo code and complexity analysis see lecture 1

Example: $\text{findElement}(7)$

```
0  1  3  5  7  8  9  11  15  16  18
```

```
0  1  3  5  7  8  9  11  15  16  18
```

```
0  1  3  5  7  8  9  11  15  16  18
```

```
0  1  3  5  7  8  9  11  15  16  18
```

```
0  1  3  5  7  8  9  11  15  16  18
```

```
0  1  3  5  7  8  9  11  15  16  18
```

```
0  1  3  5  7  8  9  11  15  16  18
```
Binary search performs \texttt{findElement}(k) on sorted arrays:

- each step the search space is halved
- time complexity is $O(\log_2 n)$
- for pseudo code and complexity analysis see lecture 1

Example: \texttt{findElement}(7)
Binary search performs \texttt{findElement}(k) on sorted arrays:

- each step the search space is halved
- time complexity is $O(\log_2 n)$
- for pseudo code and complexity analysis see lecture 1

Example: \texttt{findElement}(7)
Binary Search

Binary search performs $$\text{findElement}(k)$$ on sorted arrays:

- each step the search space is halved
- time complexity is $$O(\log_2 n)$$
- for pseudo code and complexity analysis see lecture 1

Example: $$\text{findElement}(7)$$
A lookup table is a (ordered) dictionary based on a sorted array:

- **Performance:**
  - **findElement** takes $O(\log_2 n)$ using binary search
  - **insertItem** is $O(n)$ (shifting $n/2$ items worst case)
  - **removeElement** takes $O(n)$ (shifting $n/2$ items worst case)

- Efficient for small dictionaries or if search are much more frequent than insertion and removal. (e.g. user authentication with password)
Data Structure for Trees

A node is represented by an object storing:

- an element
- link to the parent node
- list of links to children nodes
Data Structure for Binary Trees

A node is represented by an object storing:

- an element
- link to the parent node
- left child
- right child

Binary trees can also be represented by a Vector, see heaps.
A **binary search tree** is a binary tree such that:

- The inner nodes store keys (or key-element pairs).
  (leafs are empty, and usually left out in literature)

- For every node $n$:
  - the left subtree of $n$ contains only keys $< \text{key}(n)$
  - the right subtree of $n$ contains only keys $> \text{key}(n)$

- Inorder traversal visits the keys in increasing order.
Searching for key $k$ in a binary search tree $t$ works as follows:

- if the root of $t$ is an inner node with key $k'$:
  - if $k == k'$, then return the element stored at the root
  - if $k < k'$, then search in the left subtree
  - if $k > k'$, then search in the right subtree
- if $t$ is a leaf, then return No_Such_Key (key not found)

Example: `findElement(4)`
Finding the minimal key in a binary search tree:

- walk left until the left child is a leaf

\[
\text{minKey}(n): \\
\text{if } \text{isExternal}(n) \text{ then} \\
\quad \text{return No_Such_Key} \\
\text{if } \text{isExternal}(\text{leftChild}(n)) \text{ then} \\
\quad \text{return key}(n) \\
\text{else} \\
\quad \text{return minKey(\text{leftChild}(n))}
\]

Example: for the tree below, \text{minKey()} returns 1
Finding the maximal key in a binary search tree:

- walk right until the right child is a leaf

maxKey(n):

```plaintext
if isExternal(n) then
    return No_Such_Key

if isExternal(rightChild(n)) then
    return key(n)

else
    return maxKey(rightChild(n))
```

Example: for the tree below, maxKey() returns 9
Binary Search Tree: Insertion

Insertion of key \( k \) in a binary search tree \( t \):

- search for \( k \) and remember the leaf \( w \) where we ended up (assuming that we did not find \( k \))
- insert \( k \) at node \( w \), and expand \( w \) to an inner node

Example: \texttt{insert(5)}
The algorithm `removeAboveExternal(n)` removes a node for which at least one child is a leaf (external node):

- if the left child of `n` is a leaf, replace `n` by its right subtree
- if the right child of `n` is a leaf, replace `n` by its left subtree

Example: `removeAboveExternal(9)`
Binary Search Tree: Deletion

The algorithm \texttt{remove}(n) removes a node:

- if at least one child is a leaf then \texttt{removeAboveExternal}(n)

- if both children of \texttt{n} are internal nodes, then:
  
  - find the minimal node \texttt{m} in the right subtree of \texttt{n}
  
  - replace the key of \texttt{n} by the key of \texttt{m}
  
  - remove the node \texttt{m} using \texttt{removeAboveExternal}(m)

(works also with the maximal node of the left subtree)

Example: \texttt{remove}(6)
Performance of Binary Search Trees

Binary search tree storing \( n \) keys, and height \( h \):

- the space used is \( O(n) \)
- \texttt{findElement}, \texttt{insertItem}, \texttt{removeElement} take \( O(h) \) time

The height \( h \) is \( O(n) \) in worst case and \( O(\log_2 n) \) in best case:

- \texttt{findElement}, \texttt{insertItem}, and \texttt{removeElement} take \( O(n) \) time in worst case
AVL Trees

**AVL trees** are binary search trees such that for every inner node $n$ the heights of the children differ at most by 1.

- AVL trees are often called height-balanced.

- Heights of the subtrees are displayed above the nodes.
The **balance factor** of a node is the height of its right subtree minus the height of its left subtree.

Above the nodes we display the balance factor.

- Nodes with balance factor -1, 0, or 1 are called balanced.
The height of an AVL tree storing $n$ keys is $O(\log n)$.

**Proof.**

Let $n(h)$ be the minimal number of inner nodes for height $h$.

- we have $n(1) = 1$ and $n(2) = 2$

- we know

  $$n(h) = 1 + n(h - 1) + n(h - 2)$$

  $$> 2 \cdot n(h - 2) \quad \text{since } n(h - 1) > n(h - 2)$$

  $$> 4 \cdot n(h - 4)$$

  $$> 8 \cdot n(h - 6)$$

  $$n(h) > 2^i \cdot n(h - 2i) \quad \text{by induction}$$

  $$\geq 2^{h/2}$$

Thus $h \leq 2 \cdot \log_2 n(h)$.
AVL Trees: Insertion

Insertion of a key $k$ in AVL trees works in the following steps:

- We insert $k$ as for binary search trees:
  - let the inserted node be $w$

- After insertion we might need to rebalance the tree:
  - the balance of the ancestors of $w$ may be affected
  - nodes with balance factor -2 or 2 need rebalancing

Example: `insertItem(1)` (without rebalancing)
AVL Trees: Rebalancing after Insertion

After insertion the AVL tree may need to be rebalanced:

- walk from the inserted node to the root
- we rebalance the first node with balance factor -2 or 2

There are only the following 4 cases (2 modulo symmetry):

- **Left Left**
  - Balance factor: -2
  - Tree structure:
    - A
      - B
        - C
          - h
    - A
      - B
        - C
          - h

- **Left Right**
  - Balance factor: -2
  - Tree structure:
    - A
      - B
        - C
          - h
    - A
      - B
        - C
          - h

- **Right Right**
  - Balance factor: 2
  - Tree structure:
    - A
      - B
        - C
          - h
    - A
      - B
        - C
          - h

- **Right Left**
  - Balance factor: -1
  - Tree structure:
    - A
      - B
        - C
          - h
    - A
      - B
        - C
          - h

The case ‘Left Left’ requires a right rotation:

-1
  \[
  \begin{array}{c}
  y \\
  A \\
  h + 1 \text{ inserted}
  \end{array}
  \quad \quad
  \begin{array}{c}
  x \\
  B \\
  h
  \end{array}
  \quad \quad
  \begin{array}{c}
  y \\
  C \\
  h
  \end{array}
  \quad \quad
  \begin{array}{c}
  A \\
  h + 1 \text{ inserted}
  \end{array}
  \quad \quad
  \begin{array}{c}
  y \\
  B \\
  h
  \end{array}
  \quad \quad
  \begin{array}{c}
  x \\
  C \\
  h
  \end{array}
\]
Example: Case Left Left

Example: \texttt{insertItem}(0)

```
rotate right 2, 4
```
AVL Trees: Rebalancing, Case Right Right

The case ‘Right Right’ requires a left rotation:
(this case is symmetric to the case ‘Left Left’)

```
A
  ↓
  h

B
  ↓
  h

C
  ↓
  h + 1
inserted

x
  ↓
y
  ↓
  1

y

rotate left

0

A
  ↓
  h

B
  ↓
  h

C
  ↓
h + 1
inserted

x
  ↓
y
  ↓
  0
```
Example: Case Right Right

Example: `insertItem(7)`

```
Example: Case Right Right

Example: insertItem(7)

```

```
rotate right 3, 5

```

```
insert

```

```
rotate right 3, 5

```
AVL Trees: Rebalancing, Case Left Right

The case ‘Left Right’ requires a left and a right rotation:

(insertion in C or z instead of B works exactly the same)
Example: Case Left Right

Example: \textit{insertItem}(5)

rotate left 4, 6

rotate right 6, 7
AVL Trees: Rebalancing, Case Right Left

The case ‘Right Left’ requires a right and a left rotation:

\[
\begin{align*}
A & \xrightarrow{h} B \quad & \text{inserted} \\
\downarrow & & \quad \downarrow \\
2 & \xrightarrow{h-1} 1 \\
\uparrow & & \quad \uparrow \\
x & \text{rotate right } y, z \quad & \text{rotate left } x, z \\
\end{align*}
\]

\[
\begin{align*}
A & \xrightarrow{h} B \quad & \text{inserted} \\
\downarrow & & \quad \downarrow \\
2 & \xrightarrow{h-1} 0 \\
\uparrow & & \quad \uparrow \\
x & \text{rotate right } y, z \quad & \text{rotate left } x, z \\
\end{align*}
\]

(this case is symmetric to the case ‘Left Right’
Example: Case Right Left

Example: \texttt{insertItem}(7)

rotate right 7, 8

rotate left 5, 7
AVL Trees: Rebalancing after Deletion

After deletion the AVL tree may need to be rebalanced:

- walk from the inserted node to the root
- we rebalance all nodes with balance factor -2 or 2

There are only the following 6 cases (2 of which are new):

- **Left Left**
  - Balance factor: -2
  - Node B deleted
  - Node C rebalanced

- **Left Right**
  - Balance factor: -2
  - Node B deleted
  - Node C rebalanced

- **Left**
  - Balance factor: 0
  - Node C deleted

- **Right Right**
  - Balance factor: 2
  - Node B deleted
  - Node C rebalanced

- **Right Left**
  - Balance factor: 2
  - Node B deleted
  - Node C rebalanced

- **Right**
  - Balance factor: 0
  - Node C deleted

AVL Trees: Rebalancing, Case Left

The case ‘Left’ requires a right rotation:
Example: Case Left

Example: \textit{remove}(9)

rotate right 4, 7

remove
AVL Trees: Rebalancing, Case Right

The case ‘Right’ requires a left rotation:
Example: Case Right

Example: $\text{remove}(4)$

1. **Remove 4** from the tree:
   - Original tree:
     - Node 4
     - Nodes 6, 8, 7, 9
   - After removal:
     - Node 6
     - Nodes 0, 8, 0

2. **Rotate left 6, 8**:
   - Original tree:
     - Node 6
     - Nodes 0, 8, 0
   - After rotation:
     - Node 1
     - Nodes 6, 0, 7, 9
AVL Trees: Performance

A single rotation (restructure) is $O(1)$:
- assuming a linked-structure binary tree

Insertion runs in $O(\log n)$:
- initial find is $O(\log n)$
- rebalancing and update of heights is $O(\log n)$
  (at most 2 rotations are needed, no further rebalancing)

Deletion runs in $O(\log n)$:
- initial find is $O(\log n)$
- rebalancing and update of heights is $O(\log n)$
  - rebalancing may decrease the height of the subtree, thus further rebalancing above the node may be necessary

Lookup (find) is $O(\log n)$ (height of the tree is $O(\log n)$).
Show with pictures step for step how item 9 is removed.

Indicate in every picture:
- which node is not balanced, and
- which nodes are involved in rotation.
Dictionary methods:

<table>
<thead>
<tr>
<th></th>
<th>search</th>
<th>insert</th>
<th>remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log File</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Lookup Table</td>
<td>$O(\log_2 n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>AVL Tree</td>
<td>$O(\log_2 n)$</td>
<td>$O(\log_2 n)$</td>
<td>$O(\log_2 n)$</td>
</tr>
</tbody>
</table>

Ordered dictionary methods:

<table>
<thead>
<tr>
<th></th>
<th>closestAfter</th>
<th>closestBefore</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log File</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Lookup Table</td>
<td>$O(\log_2 n)$</td>
<td>$O(\log_2 n)$</td>
</tr>
<tr>
<td>AVL Tree</td>
<td>$O(\log_2 n)$</td>
<td>$O(\log_2 n)$</td>
</tr>
</tbody>
</table>

- Log File corresponds to unsorted list.
- Lookup Table corresponds to unsorted list.
A hash function $h$ maps keys of a given type to integers in a fixed interval $[0, \ldots, N - 1]$. We call $h(x)$ hash value of $x$.

Examples:

- $h(x) = x \mod N$ is a hash function for integer keys
- $h((x, y)) = (5 \cdot x + 7 \cdot y) \mod N$ is a hash function for pairs of integers

<table>
<thead>
<tr>
<th>key</th>
<th>element</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>14</td>
</tr>
</tbody>
</table>

A hash table consists of:
- hash function $h$
- an array (called table) of size $N$

The idea is to store item $(k, e)$ at index $h(k)$. 
Hash Tables: Example 1

Example: phone book with table size $N = 5$
- hash function $h(w) = (\text{length of the word } w) \mod 5$

- Alice
- John
- Sue

- Ideal case: one access for $\text{find}(k)$ (that is, $O(1)$).
- Problem: collisions
  - Where to store Joe (collides with Sue)?
- This is an example of a bad hash function:
  - Lots of collisions even if we make the table size $N$ larger.
Hash Tables: Example 2

A dictionary based on a hash table for:
- items (social security number, name)
- 700 persons in the database

We choose a hash table of size $N = 1000$ with:
- hash function $h(x) =$ last three digits of $x$

- (025-611-001, Mr. X)
- (987-067-002, Brad Pit)
- (431-763-997, Alan Turing)
- (007-007-999, James Bond)
Collisions

Collisions occur when different elements are mapped to the same cell:

- Keys $k_1, k_2$ with $h(k_1) = h(k_2)$ are said to collide.

Different possibilities of handing collisions:

- chaining,
- linear probing,
- double hashing, ...
Collisions continued

Usual setting:

- The set of keys is much larger than the available memory.
- Hence collisions are unavoidable.

How probable are collisions:

- We have a party with $p$ persons. What is the probability that at least 2 persons have birthday the same day ($N = 365$).

- Probability for no collision:

  \[
  q(p, N) = \frac{N}{N} \cdot \frac{N-1}{N} \cdot \ldots \cdot \frac{N-p+1}{N} = \frac{(N-1) \cdot (N-2) \cdot \ldots \cdot (N-p+1)}{N^{p-1}}
  \]

- Already for $p \geq 23$ the probability for collisions is $> 0.5$. 
Hashing: Efficiency Factors

The efficiency of hashing depends on various factors:

- hash function
- type of the keys: integers, strings, ...
- distribution of the actually used keys
- occupancy of the hash table (how full is the hash table)
- method of collision handling

The load factor $\alpha$ of a hash table is the ratio $n/N$, that is, the number of elements in the table divided by size of the table.

High load factor $\alpha \geq 0.85$ has negative effect on efficiency:

- lots of collisions
- low efficiency due to collision overhead
What is a good Hash Function?

Hash functions should have the following properties:

▶ Fast computation of the hash value ($O(1)$).

▶ Hash values should be distributed (nearly) uniformly:
  
  ▶ Every has value (cell in the hash table) has equal probability.
  
  ▶ This should hold even if keys are non-uniformly distributed.

The goal of a hash function is:

▶ ‘disperse’ the keys in an apparently random way

Example (Hash Function for Strings in Python)

We display python hash values modulo 997:

\[
\begin{align*}
  h(‘a’) &= 535  \\
  h(‘b’) &= 80  \\
  h(‘c’) &= 618  \\
  h(‘d’) &= 163  \\
  h(‘ab’) &= 354  \\
  h(‘ba’) &= 979
\end{align*}
\]

At least at first glance they look random.
Hash function is usually specified as composition of:

- **hash code map**: $h_1 : \text{keys} \rightarrow \text{integers}$
- **compression map**: $h_2 : \text{integers} \rightarrow [0, \ldots, N - 1]$

The hash code map is applied before the compression map:

$h(x) = h_2(h_1(x))$ is the composed hash function

The compression map usually is of the form $h_2(x) = x \mod N$:

- The actual work is done by the hash code map.
- What are good $N$ to choose? ... see following slides
We revisit the example (social security number, name):

- hash function $h(x) = x$ as number mod 1000

Assume the last digit is always 0 or 1 indicating male/female.

Then 80% of the cells in the table stay unused! Bad hash!
A better hash function for ‘social security number’:

- hash function \( h(x) = x \) as number \( \mod 997 \)
- e.g. \( h(025 - 611 - 000) = 025611000 \mod 997 = 409 \)

Why 997? Because 997 is a prime number!

- Let the hash function be of the form \( h(x) = x \mod N \).
- Assume the keys are distributed in equidistance \( \Delta < N \):
  \[
  k_i = z + i \cdot \Delta
  \]

We get a collision if:

\[
 k_i \mod N = k_j \mod N
\]

\[\iff \quad z + i \cdot \Delta \mod N = z + j \cdot \Delta \mod N \]

\[\iff \quad i = j + m \cdot N \quad (\text{for some } m \in \mathbb{Z}) \]

Thus a prime maximizes the distance of keys with collisions!
Hash Code Maps

What if the keys are not integers?

- **Integer cast**: interpret the bits of the key as integer.

\[
\begin{align*}
\text{a} & : 0001 \\
\text{b} & : 0010 \\
\text{c} & : 0011 \\
\end{align*}
\]

\[001001000111 = 291\]

What if keys are longer than 32/64 bit Integers?

- **Component sum**:
  - partition the bits of the key into parts of fixed length
  - combine the components to one integer using sum
    (other combinations are possible, e.g. bitwise xor, . . .)

\[
\begin{align*}
1001010 & | 0010111 & | 0110000 \\
1001010 + 0010111 + 0110000 &= 74 + 23 + 48 = 145
\end{align*}
\]
Other possible hash code maps:

- **Polynomial accumulation:**
  - partition the bits of the key into parts of fixed length
    \[ a_0 a_1 a_2 \ldots a_n \]
  - take as hash value the value of the polynom:
    \[ a_0 + a_1 \cdot z + a_2 \cdot z^2 \ldots a_n \cdot z^n \]
  - especially suitable for strings (e.g. \( z = 33 \) has at most 6 collisions for 50,000 english words)

- **Mid-square method:**
  - pick \( m \) bits from the middle of \( x^2 \)

- **Random method:**
  - take \( x \) as seed for random number generator
Collision Handling: Chaining

Chaining: each cell of the hash table points to a linked list of elements that are mapped to this cell.

- colliding items are stored outside of the table
- simple but requires additional memory outside of the table

Example: keys = birthdays, elements = names

- hash function: \( h(x) = (\text{month of birth}) \mod 5 \)

```
<table>
<thead>
<tr>
<th>cell</th>
<th>elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
</tr>
<tr>
<td>1</td>
<td>(01.01., Sue) → ∅</td>
</tr>
<tr>
<td>2</td>
<td>∅</td>
</tr>
<tr>
<td>3</td>
<td>(12.03., John) → (16.08., Madonna) → ∅</td>
</tr>
<tr>
<td>4</td>
<td>∅</td>
</tr>
</tbody>
</table>
```

Worst-case: everything in one cell, that is, linear list.
Collision Handling: Linear Probing

Open addressing:
> the colliding items are placed in a different cell of the table

Linear probing:
> colliding items stored in the next (circularly) available cell
> testing if cells are free is called ‘probing’

Example: $h(x) = x \mod 13$
> we insert: 18, 41, 22, 44, 59, 32, 31, 73

Colliding items might lump together causing new collisions.
Linear Probing: Search

Searching for a key \( k \) (\text{findElement}(k)) works as follows:

- Start at cell \( h(k) \), and probe consecutive locations until:
  - an item with key \( k \) is found, or
  - an empty cell is found, or
  - all \( N \) cells have been probed unsuccessfully.

\text{findElement}(k):
\[
i = h(k) \\
p = 0 \\
\text{while } p < N \text{ do} \\
\quad c = A[i] \\
\quad \text{if } c == \emptyset \text{ then return } \text{No\_Such\_Key} \\
\quad \text{if } c.key == k \text{ then return } c.element \\
\quad i = (i + 1) \mod N \\
\quad p = p + 1 \\
\text{return } \text{No\_Such\_Key}
\]
Deletion $\text{remove}(k)$ is expensive:

- Removing 15, all consecutive elements have to be moved:

To avoid the moving we introduce a special element $\text{Available}$:

- Instead of deleting, we replace items by $\text{Available}$ (A).

- From time to time we need to ‘clean up’:
  - remove all $\text{Available}$ and reorder items
Linear Probing: Inserting

Inserting \texttt{insertItem}(k, o):

- Start at cell $h(k)$, probe consecutive elements until:
  - empty or \textcolor{red}{Available} cell is found, then store item here, or
  - all $N$ cells have been probed (table full, throw exception)

Example: \texttt{insert}(3) in the above table yields ($h(x) = x \mod 13$)

Important: for \texttt{findElement} cells with \textcolor{red}{Available} are treated as filled, that is, the search continues.
Linear Probing: Possible Extensions

Disadvantages of linear probing:
▶ Colliding items lump together, causing:
   ▶ longer sequences of probes
   ▶ reduced performance

Possible improvements/ modifications:
▶ instead of probing successive elements, compute the $i$-th probing index $h_i$ depending on $i$ and $k$:

$$h_i(k) = h(k) + f(i, k)$$

Examples:
▶ Fixed increment $c$: $h_i(k) = h(k) + c \cdot i$.
▶ Changing directions: $h_i(k) = h(k) + c \cdot i \cdot (-1)^i$.
▶ Double hashing: $h_i(k) = h(k) + i \cdot h'(k)$.
Double Hashing

Double hashing uses a secondary hash function \( d(k) \):

- Handles collisions by placing items in the first available cell
  \[ h(k) + j \cdot d(k) \]
  for \( j = 0, 1, \ldots, N - 1 \).

- The function \( d(k) \) always be > 0 and < \( N \).

- The size of the table \( N \) should be a prime.
Double Hashing: Example

We use double hashing with:

- \( N = 13 \)
- \( h(k) = k \mod 13 \)
- \( d(k) = 7 - (k \mod 7) \)

<table>
<thead>
<tr>
<th>( k )</th>
<th>( h(k) )</th>
<th>( d(k) )</th>
<th>Probes</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>41</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>22</td>
<td>9</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>44</td>
<td>5</td>
<td>5</td>
<td>5, 10</td>
</tr>
<tr>
<td>59</td>
<td>7</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>31</td>
<td>5</td>
<td>4</td>
<td>5, 9, 0</td>
</tr>
<tr>
<td>73</td>
<td>8</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
Performance of Hashing

In worst case insertion, lookup and removal take $O(n)$ time:
- occurs when all keys collide (end up in one cell)

The load factor $\alpha = \frac{n}{N}$ affects the performance:
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes is:

$$\frac{1}{1 - \alpha}$$

![Graph showing the relationship between $\alpha$ and $f(x)$]
Performance of Hashing

In worst case insertion, lookup and removal take $O(n)$ time:
- occurs when all keys collide (end up in one cell)

The load factor $\alpha = n/N$ affects the performance:
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes is:
  $$\frac{1}{1 - \alpha}$$

In practice hashing is very fast as long as $\alpha < 0.85$:
- $O(1)$ expected running time for all Dictionary ADT methods

Applications of hash tables:
- small databases
- compilers
- browser caches
Universal Hashing

No hash function is good in general:

- there always exist keys that are mapped to the same value

Hence no single hash function $h$ can be proven to be good.

However, we can consider a set of hash functions $H$.
(assume that keys are from the interval $[0, M - 1]$)

We say that $H$ is universal (good) if for all keys $0 \leq i \neq j < M$:

$$\text{probability}(h(i) = h(j)) \leq \frac{1}{N}$$

for $h$ randomly selected from $H$. 
The following set of hash functions $H$ is universal:

- Choose a prime $p$ between $M$ and $2 \cdot M$.
- Let $H$ consist of the functions
  \[ h(k) = ((a \cdot k + b) \mod p) \mod N \]
  for $0 < a < p$ and $0 \leq b < p$.

**Proof Sketch.**

Let $0 \leq i \neq j < M$. For every $i' \neq j' < p$ there exist unique $a, b$ such that $i' = a \cdot i + b \mod p$ and $j' = a \cdot i + b \mod p$. Thus every pair $(i', j')$ with $i' \neq j'$ has equal probability. Consequently the probability for $i' \mod N = j' \mod N$ is $\leq \frac{1}{N}$. \qed
Comparison AVL Trees vs. Hash Tables

Dictionary methods:

<table>
<thead>
<tr>
<th></th>
<th>search</th>
<th>insert</th>
<th>remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVL Tree</td>
<td>$O(\log_2 n)$</td>
<td>$O(\log_2 n)$</td>
<td>$O(\log_2 n)$</td>
</tr>
<tr>
<td>Hash Table</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

1 expected running time of hash tables, worst-case is $O(n)$.

Ordered dictionary methods:

<table>
<thead>
<tr>
<th></th>
<th>closestAfter</th>
<th>closestBefore</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVL Tree</td>
<td>$O(\log_2 n)$</td>
<td>$O(\log_2 n)$</td>
</tr>
<tr>
<td>Hash Table</td>
<td>$O(n + N)$</td>
<td>$O(n + N)$</td>
</tr>
</tbody>
</table>

Examples, when to use AVL trees instead of hash tables:

1. if you need to be sure about worst-case performance
2. if keys are imprecise (e.g. measurements), e.g. find the closest key to 3.24: closestTo(3.72)
Sorting Algorithms

We have already seen:
- Selection-sort
- Insertion-sort
- Heap-sort

We will see:
- Bubble-sort
- Merge-sort
- Quick-sort

We will show that:
- $O(n \cdot \log n)$ is optimal for comparison based sorting.
The basic idea of bubble-sort is as follows:

- exchange neighboring elements that are in wrong order
- stops when no elements were exchanged

**bubbleSort(A):**

```plaintext
n = length(A)
swapped = true
while swapped == true do
    swapped = false
    for i = 0 to n - 2 do
        if A[i] > A[i + 1] then
            swap(A[i], A[i + 1])
            swapped = true
    done
n = n - 1
done
```
Bubble-Sort: Example

Unsorted Part

Sorted Part
Bubble-Sort: Properties

Time complexity:

- worst-case:
  \[(n - 1) + \ldots + 1 = \frac{(n - 1)^2 + n - 1}{2} \in O(n^2)\]
  (caused by sorting an inverse sorted list)

- best-case: \(O(n)\)

Bubble-sort is:

- slow
- in-place
Divide-and-Conquer is a general algorithm design paradigm:

- **Divide**: divide the input $S$ into two disjoint subsets $S_1$, $S_2$
- **Recur**: recursively solve the subproblems $S_1$, $S_2$
- **Conquer**: combine solutions for $S_1$, $S_2$ to a solution for $S$ (the base case of the recursion are problems of size 0 or 1)

Example: merge-sort

```
7 2 | 9 4 ↦ 2 4 7 9

7 2 ↦ 2 7
7 ↦ 7

9 4 ↦ 4 9
9 ↦ 9
4 ↦ 4
```

- $|$ indicates the splitting point
- $\mapsto$ indicates merging of the sub-solutions
Merge-Sort

Merge-sort of a list $S$ with $n$ elements works as follows:

- **Divide:** divide $S$ into two lists $S_1$, $S_2$ of $\approx n/2$ elements
- **Recur:** recursively sort $S_1$, $S_2$
- **Conquer:** merge $S_1$ and $S_2$ into a sorting of $S$

**Algorithm** $\text{mergeSort}(S, C)$:

**Input:** a list $S$ of $n$ elements and a comparator $C$

**Output:** the list $S$ sorted according to $C$

\[
\text{if } \text{size}(S) > 1 \text{ then } \\
(S_1, S_2) = \text{partition } S \text{ into size } \lfloor n/2 \rfloor \text{ and } \lceil n/2 \rceil \\
\text{mergeSort}(S_1, C) \\
\text{mergeSort}(S_2, C) \\
S = \text{merge}(S_1, S_2, C)
\]
Merging two Sorted Sequences

Algorithm \text{merge}(A, B, C):

\textbf{Input:} sorted lists \(A, B\)

\textbf{Output:} sorted lists containing the elements of \(A\) and \(B\)

\(S = \) empty list

\textbf{while} \(\neg A.\text{isEmtpy}()\) or \(\neg B.\text{isEmtpy}()\) \textbf{do}

\hspace{1em} \textbf{if} \(A.\text{first}().\text{element} < B.\text{first}().\text{element}\) \textbf{then}

\hspace{2em} \(S.\text{insertLast}(A.\text{remove}(A.\text{first}()))\)

\hspace{1em} \textbf{else}

\hspace{2em} \(S.\text{insertLast}(B.\text{remove}(B.\text{first}()))\)

\textbf{done}

\textbf{while} \(\neg A.\text{isEmtpy}()\) \textbf{do} \(S.\text{insertLast}(A.\text{remove}(A.\text{first}()))\)

\textbf{while} \(\neg B.\text{isEmtpy}()\) \textbf{do} \(S.\text{insertLast}(B.\text{remove}(B.\text{first}()))\)

Performance:

- Merging two sorted lists of length about \(n/2\) is \(O(n)\) time.
  (for singly linked lists, double linked lists, and arrays)
Merge-sort: Example

Divide (split)

Conquer (merge)
An execution of merge-sort can be displayed in a binary tree:

- each node represents recursive call and stores:
  - unsorted sequence before execution, its partition
  - sorted sequence after execution
- leaves are calls on subsequences of size 0 or 1
Merge-Sort: Example Execution

Finished merge-sort tree.
Merge-Sort: Running Time

The height $h$ of the merge-sort tree is $O(\log_2 n)$:
- each recursive call splits the sequence in half

The work all nodes together at depth $i$ is $O(n)$:
- partitioning, and merging of $2^i$ sequences of $n/2^i$
- $2^{i+1} \leq n$ recursive calls

<table>
<thead>
<tr>
<th>depth</th>
<th>nodes</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$n$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>$n/2$</td>
</tr>
<tr>
<td>$i$</td>
<td>$2^i$</td>
<td>$n/2^i$</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

Thus the worst-case running time is $O(n \cdot \log_2 n)$. 
Quick-Sort

Quick-sort of a list $S$ with $n$ elements works as follows:

- **Divide:** pick random element $x$ (pivot) from $S$ and split $S$ in:
  - $L$ elements less than $x$
  - $E$ elements equal than $x$
  - $G$ elements greater than $x$

  ![List](image)

- **Recur:** recursively sort $L$, and $G$

  ![List](image)

- **Conquer:** join $L$, $E$, and $G$

  ![List](image)
Quick-Sort: The Partitioning

The partitioning runs in $O(n)$ time:

- we traverse $S$ and compare every element $y$ with $x$
- depending on the comparison insert $y$ in $L$, $E$ or $G$

Algorithm $\text{partition}(S, p)$:

Input: a list $S$ of $n$ elements, and position $p$ of the pivot
Output: list $L$, $E$, $G$ of lists less, equal or greater than pivot

$L, E, G = \text{empty lists}$
$x = S.\text{elementAtRank}(p)$

while $\neg S.\text{isEmpty}()$ do
  $y = S.\text{remove}(S.\text{first}())$
  if $y < x$ then $L.\text{insertLast}(y)$
  if $y == x$ then $E.\text{insertLast}(y)$
  if $y > x$ then $G.\text{insertLast}(y)$

done

return $L$, $E$, $G$
An execution of quick-sort can be displayed in a binary tree:

- each node represents recursive call and stores:
  - unsorted sequence before execution, and its pivot
  - sorted sequence after execution
- leaves are calls on subsequences of size 0 or 1
Quick-Sort: Example

8 2 9 3 1 5 7 6 4 ↦ 1 2 3 4 5 6 7 8 9

2 3 1 5 4 ↦ 1 2 3 4 5

2 3 5 4 ↦ 2 3 4 5

2 ↦ 2

5 4 ↦ 4 5

4 ↦ 4
Quick-Sort: Worst-Case Running Time

The worst-case running time occurs when:

- the pivot is always the minimal or maximal element
- then one $L$ and $G$ has size $n - 1$, the other size 0

Then the running time is $O(n^2)$:

$$n + (n - 1) + (n - 2) + \ldots + 1 \in O(n^2)$$
Consider a recursive call on a list of size $m$:

- **Good call**: if both $L$ and $G$ are each less than $\frac{3}{4} \cdot s$ size
- **Bad call**: one of $L$ and $G$ is greater than $\frac{3}{4} \cdot s$ size

A good call has probability $\frac{1}{2}$:

- half of the pivots give rise to good calls
Quick-Sort: Average Running Time

For a node at depth $i$, we expect (average):

- $i/2$ ancestors are good calls
- the size of the sequence is $\leq (3/4)^{i/2} \cdot n$

As a consequence:

- for a node at depth $2 \cdot \log_{3/4} n$ the expected input size is 1
- the expected height of the quick-sort tree is $O(\log n)$

The amount of work at depth $i$ is $O(n)$.
Thus the expected (average) running time is $O(n \cdot \log n)$. 
In-Place Quick-Sort

Quick-Sort can be sorted in-place (but then non-stable):

Algorithm inPlaceQuickSort(A, l, r):
   Input: list A, indices l and r
   Output: list A where elements from index l to r are sorted

   if l ≥ r then return
   p = A[r]  (take rightmost element as pivot)
   l' = l  and  r' = r
   while l' ≤ r' do
      while l' ≤ r' and A[l'] ≤ p do l' = l' + 1  (find > p)
      while l' ≤ r' and A[r'] ≥ p do r' = r' - 1  (find < p)
      if l' < r' then swap(A[l'], A[r'])  (swap < p with > p)
   done
   swap(A[r], A[l'])  (put pivot into the right place)
   inPlaceQuickSort(A, l, l' - 1)  (sort left part)
   inPlaceQuickSort(A, l' + 1, r)  (sort right part)

Considered in-place although recursion needs O(log n) space.
In-Place Quick-Sort: Example

Unsorted Part

Sorted Part

Pivot
Many sorting algorithms are comparison based:

- sort by comparing pairs of objects
- Examples: selection-sort, insertion-sort, bubble-sort, heap-sort, merge-sort, quick-sort, ... 

No comparison based sorting algorithm can be faster than \( \Omega(n \cdot \log n) \) time (worst-case).

We will prove this lower bound on the next slides...
We will only count comparisons (sufficient for lower bound):

▶ Assume input is a permutation of the numbers 1, 2, \ldots, n.

▶ Every execution corresponds to a path in the decision tree:

Algorithm itself maybe does not have this tree structure, but this is the maximal information gained by the algorithm.
Every leaf corresponds to exactly one input permutation:

- application of the same swapping steps on two different input permutations, e.g. \( \ldots,6,\ldots,7,\ldots \) and \( \ldots,7,\ldots,6,\ldots \), yields different results (not both results can be sorted)
The height of the tree is a lower bound on the running time:

- There are \( n! = n \cdot (n - 1) \cdots 1 \) permutations of 1, 2, \ldots, \( n \).
- Thus the height of the tree is at least: \( \log_2(n!) \).
Hence any comparison-based sorting algorithm takes at least

\[ \log_2(n!) \geq \log_2\left(\frac{n}{2}\right)^{n/2} \]

\[ = \frac{n}{2} \log_2 \frac{n}{2} \]

\[ \in \Omega\left(n \cdot \log n\right) \]

time in worst-case.
## Summary of Comparison-Based Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection-sort</td>
<td>$O(n^2)$</td>
<td>▶ slow (but good for small lists)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ in-place, stable</td>
</tr>
<tr>
<td>insertion-sort</td>
<td>$O(n^2)$</td>
<td>▶ insertion-sort good for online sorting and nearly sorted lists</td>
</tr>
<tr>
<td>bubble-sort</td>
<td>$O(n^2)$</td>
<td>▶ in-place, not stable, fast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ good for large inputs ($1K − 1M$)</td>
</tr>
<tr>
<td>heap-sort</td>
<td>$O(n \cdot \log_2 n)$</td>
<td>▶ fast, stable, usually not in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ sequential data access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▶ good for large inputs ($&gt; 1M$)</td>
</tr>
<tr>
<td>merge-sort</td>
<td>$O(n \cdot \log_2 n)$</td>
<td>▶ in-place, randomized, not stable</td>
</tr>
<tr>
<td>quick-sort</td>
<td>$O(n \cdot \log_2 n)$ (expected)</td>
<td>▶ fastest, good for huge inputs</td>
</tr>
</tbody>
</table>

Quick-sort usually performs fastest, although worst-case $O(n^2)$. 
## Sorting: Comparison of Runtime

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>25.000 sorted</th>
<th>100.000 sorted</th>
<th>25.000 not sorted</th>
<th>100.000 not sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection-sort</td>
<td>1.1</td>
<td>19.4</td>
<td>1.1</td>
<td>19.5</td>
</tr>
<tr>
<td>insertion-sort</td>
<td>0</td>
<td>0</td>
<td>1.1</td>
<td>19.6</td>
</tr>
<tr>
<td>bubble-sort</td>
<td>0</td>
<td>0</td>
<td>5.5</td>
<td>89.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>5 million sorted</th>
<th>20 million sorted</th>
<th>5 million not sorted</th>
<th>20 million not sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>insertion-sort</td>
<td>0.03</td>
<td>0.13</td>
<td>timeout</td>
<td>timeout</td>
</tr>
<tr>
<td>heap-sort</td>
<td>3.6</td>
<td>15.6</td>
<td>8.3</td>
<td>42.2</td>
</tr>
<tr>
<td>merge-sort</td>
<td>2.5</td>
<td>10.5</td>
<td>3.7</td>
<td>16.1</td>
</tr>
<tr>
<td>quick-sort</td>
<td>0.5</td>
<td>2.2</td>
<td>2.0</td>
<td>8.7</td>
</tr>
</tbody>
</table>

Source: Gumm, Sommer *Einführung in die Informatik.*
Let $S$ be a list of $n$ key-element items with keys in $[0, N - 1]$. Bucket-sort uses the keys as indices into auxiliary array $B$:

- the elements of $B$ are lists, so-called buckets
- Phase 1:
  - empty $S$ by moving each item $(k, e)$ into its bucket $B[k]$
- Phase 2:
  - for $i = 0, \ldots, N - 1$ move the items of $B[k]$ to the end of $S$

Performance:

- phase 1 takes $O(n)$ time
- phase 2 takes $O(n + N)$ time

Thus bucket-sort is $O(n + N)$. 
Bucket-Sort: Example

- **key range** [0, 9]

![Bucket-Sort Example Diagram]

- **Phase 1: filling the buckets**

- **Phase 2: emptying the buckets into the list**
Bucket-Sort: Properties and Extensions

The keys are used as indices for an array, thus:

- keys should be numbers from $[0, N - 1]$
- no external comparator

Bucket-sort is a stable sorting algorithm.

Extensions:

- can be extended to an arbitrary (fixed) finite set of keys $D$ (e.g. the names of the 50 U.S. states)
- sort $D$ and compute the rank $\text{rankOf}(k)$ of each element
- put item $(k, e)$ into bucket $B[\text{rankOf}(k)]$

Bucket-sort runs in $O(n + N)$ time:

- very efficient if keys come from a small intervall $[0, N - 1]$ (or in the extended version from a small set $D$)
Lexicographic Order

A $d$-tuple is a sequence of $d$ keys $(k_1, k_2, \ldots, k_d)$:
- $k_i$ is called the $i$-th dimension of the tuple

Example: $(2, 5, 1)$ as point in 3-dimensional space

The lexicographic order of $d$ tuples is recursively defined:

$$(x_1, x_2, \ldots, x_d) < (y_1, y_2, \ldots, y_d)$$

$$\iff x_1 < y_1 \lor (x_1 = y_1 \land (x_2, \ldots, x_d) < (y_2, \ldots, y_d))$$

That is, the tuples are first compared by dimension 1, then 2, \ldots
Lexicographic-Sort

Lexicographic-sort sorts a list of $d$-tuples in lexicographic order:

- Let $C_i$ be comparator comparing tuples by $i$-th dimension.
- Let \texttt{stableSort} be a stable sorting algorithm.

Lexicographic-sort executes $d$-times \texttt{stableSort}, thus:

- let $T(n)$ be the running time of \texttt{stableSort}
- then lexicographic-sort runs in $O(d \cdot T(n))$

Algorithm \texttt{lexicographicSort}(\textit{S}): 
\textbf{Input:} a list \textit{S} of $d$-tuples 
\textbf{Output:} list \textit{S} sorted in lexicographic order 

\begin{verbatim}
for i = d downto 1 do
    stableSort(\textit{S}, \textit{C}_i)
end for
\end{verbatim}
Lexicographic-Sort: Example
Number representations

We can write numbers in different numeral systems, e.g.:

- \(43_{10}\), that is, 43 in decimal system (base 10)
- \(101011_2\), that is, 43 in binary system (base 2)
- \(1121_3\), that is, 43 represented base 3

For every base \(b \geq 2\) and every number \(m\) there exist unique digits \(0 \leq d_0, \ldots, d_l < b\) such that:

\[
m = d_l \cdot b^l + d_{l-1} \cdot b^{l-1} + \ldots + d_1 \cdot b^1 + d_0 \cdot b^0
\]

and if \(l > 0\) then \(d_l \neq 0\).

Example

\[
\begin{align*}
43 &= 43_{10} \\
&= 4 \cdot 10^1 + 3 \cdot 10^0 \\
&= 101011_2 \\
&= 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
&= 1121_3 \\
&= 1 \cdot 3^3 + 1 \cdot 3^2 + 2 \cdot 3^1 + 1 \cdot 3^0
\end{align*}
\]
Radix-Sort

Radix-sort is specialization of lexicographic-sort:
▶ uses bucket-sort as stable sorting algorithm
▶ is applicable if tuples consists of integers from $[0, N - 1]$
▶ runs in $O(d \cdot (n + N))$ time

Sorting integers of fixed bit-length $d$ in linear time:
▶ consider a list of $n$ $d$-bit integers $x_{d-1}x_{d-2}\ldots x_0$ (base 2)
▶ thus each integer is a $d$-tuple $(x_{d-1}, x_{d-2}, \ldots, x_0)$
▶ apply radix sort with $N = 2$
▶ the runtime is $O(d \cdot n)$

For example, we can sort 32-bit integers in linear time.
We sort the following list of 4-bit integers:
Exercise C-4.14

Suppose we are given a sequence $S$ of $n$ elements each of which is an integer from $[0, n^2 - 1]$. Describe a simple method for sorting $S$ in $O(n)$ time.

- Each number from $[0, n^2 - 1]$ can be represented by a two digit number in the number system with base $n$.

\[ (n - 1) \cdot n + (n - 1) = n^2 - 1 \]

- Conversion of each element into base-$n$ is $O(1)$. ($O(n)$ for the whole list).

- Then use radix-sort to sort in $O(2 \cdot n)$, that is, $O(n)$ time.
The Selection Problem

The selection problem:

- given an integer $k$ and a list $x_1, \ldots, x_n$ of $n$ elements
- find the $k$-th smallest element in the list

Example: the 3rd smallest element of the following list is 6

7 4 9 6 2

An $O(n \cdot \log n)$ solution:

- sort the list ($O(n \cdot \log n)$)
- pick the $k$-th element of the sorted list ($O(1)$)

2 4 6 7 9

Can we find the $k$-th smallest element faster?
Quick-Select

Quick-select of the $k$-th smallest element in the list $S$:

- based on prune-and-search paradigm

- **Prune**: pick random element $x$ (pivot) from $S$ and split $S$ in:
  - $L$ elements $< x$, $E$ elements $==$ $x$, $G$ elements $> x$

- partitioning into $L$, $E$ and $G$ works precisely as for quick-sort

- **Search**:
  - if $k \leq |L|$ then return $\text{quickSelect}(k, L)$
  - if $|L| < k \leq |L| + |E|$ then return $x$
  - if $k > |L| + |E|$ then return $\text{quickSelect}(k - |L| - |E|, G)$
Quick-select can be displayed by a sequence of nodes:

- each node represents recursive call and stores: $k$, the sequence, and the pivot element

- $k = 5$, $S = (7 4 9 3 2 6 5 1 8)$
- $k = 2$, $S = (7 4 9 6 5 8)$
- $k = 2$, $S = (7 4 6 5)$
- $k = 1$, $S = (7 6 5)$
- found 5
Quick-Select: Running Time

The worst-case running is $O(n^2)$ time:

- if the pivot is always the minimal or maximal element

The expected running time is $O(n)$ (compare with quick-sort):

- with probability 0.5 the recursive call is good: $(3/4)n$ size

- $T(n) \leq b \cdot a \cdot n + T\left(\frac{3}{4}n\right)$
  - $a$ is the time steps for partitioning per element
  - $b$ is the expected number of calls until a good call
  - $b = 2$ (average number of coins to toss until head shows up)

- Let $m = 2 \cdot a \cdot n$, then
  
  $T(n) \leq 2 \cdot a \cdot n + T\left((3/4)n\right)$
  
  \[
  \leq 2 \cdot a \cdot n + 2 \cdot a \cdot (3/4) \cdot n + 2 \cdot a \cdot (3/4)^2 \cdot n \ldots
  \]
  
  $= 8 \cdot a \cdot n \in O(n)$ (geometric series)
Quick-Select: Median of Medians

We can do selection in $O(n)$ worst-case time.

Idea: recursively use select itself to find a good pivot:

- divide $S$ into $n/5$ sets of 5 elements
- find a median in each set (baby median)
- recursively use select to find the median of the medians

The minimal size of $L$ and $G$ is $0.3 \cdot n$. 
Quick-Select: Median of Medians

We know:

- The minimal size of $L$ and $G$ is $0.3 \cdot n$.
- Thus the maximal size of $L$ and $G$ is $0.7 \cdot n$.

Let $b \in \mathbb{N}$ such that:

- partitioning of a list of size $n$ takes at most $b \cdot n$ time,
- finding the baby medians takes at most $b \cdot n$ time,
- the base case $n \leq 1$ takes at most $b$ time.

We derive a recurrence equation for the time complexity:

$$T(n) = \begin{cases} 
  b & \text{if } n \leq 1 \\
  T(0.7 \cdot n) + T(0.2 \cdot n) + 2 \cdot b \cdot n & \text{if } n > 1
\end{cases}$$

We will see how to solve recurrence equations...
We have considered algorithms for solving special problems. Now we consider a few fundamental techniques:

- Devide-and-Conquer
- The Greedy Method
- Dynamic Programming
Divide-and-Conquer is a general algorithm design paradigm:

- **Divide**: divide input $S$ into $k \geq 2$ disjoint subsets $S_1, \ldots, S_k$
- **Recur**: recursively solve the subproblems $S_1, \ldots, S_k$
- **Conquer**: combine solutions for $S_1, \ldots, S_k$ to solution for $S$ (the base case of the recursion are problems of size 0 or 1)

We have already seen examples:

- merge sort
- quick sort
- bottom-up heap construction

Focus now: analysing time complexity by recurrence equations.
We consider 3 methods for solving recurrence equation:

- Iterative substitution method,
- Recursion tree method,
- Guess-and-test method, and
- Master method.
Iterative Substitution

Iterative substitution technique works as follows:

- iteratively apply the recurrence equations to itself
- hope to find a pattern

Example

\[
T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
T(n-1) + 2 & \text{if } n > 1
\end{cases}
\]

We start with \( T(n) \) and apply the recursive case:

\[
T(n) = T(n-1) + 2 \\
= T(n-2) + 4 \\
= T(n-k) + 2 \cdot k
\]

For \( k = n - 1 \) we reach the base case \( T(n-k) = 1 \) thus:

\[
T(n) = 1 + 2 \cdot (n - 1) = 2 \cdot n - 1
\]
Merge-Sort Review

Merge-sort of a list $S$ with $n$ elements works as follows:

- **Divide:** divide $S$ into two lists $S_1$, $S_2$ of $\approx n/2$ elements
- **Recur:** recursively sort $S_1$, $S_2$
- **Conquer:** merge $S_1$ and $S_2$ into a sorting of $S$

Let $b \in \mathbb{N}$ such that:

- merging two lists of size $n/2$ takes at most $b \cdot n$ time, and
- the base case $n \leq 1$ takes at most $b$ time

We obtain the following recurrence equation for merge-sort:

$$T(n) = \begin{cases} 
b & \text{if } n \leq 1 \\
2 \cdot T(n/2) + b \cdot n & \text{if } n > 1 
\end{cases}$$

We search a **closed solution** for the equation, that is:
- $T(n) = \ldots$ where $T(n)$ does not occur in the right side
Example: Merge-Sort

\[ T(n) = \begin{cases} 
  b & \text{if } n \leq 1 \\
  2 \cdot T(n/2) + b \cdot n & \text{if } n > 1 
\end{cases} \]

We assume that \( n \) is a power of 2: \( n = 2^k \) (that is, \( k = \log_2 n \))

- allowed since we are interested in asymptotic behaviour

We start with \( T(n) \) and apply the recursive case:

\[
T(n) = 2 \cdot T(n/2) + b \cdot n \\
= 2 \cdot (2 \cdot T(n/2^2) + b \cdot (n/2)) + b \cdot n \\
= 2^2 \cdot T(n/2^2) + 2 \cdot b \cdot n \\
= 2^3 \cdot T(n/2^3) + 3 \cdot b \cdot n \\
= 2^k \cdot T(n/2^k) + k \cdot b \cdot n \\
= n \cdot b + (\log_2 n) \cdot b \cdot n 
\]

Thus \( T(n) = b \cdot (n + n \cdot \log_2 n) \in O(n \log_2 n) \).
The Recursion Tree Method

The recursion tree method is a visual approach:

- draw the recursion tree and hope to find a pattern

\[
T(n) = \begin{cases} 
  b & \text{if } n \leq 2 \\
  3 \cdot T(n/3) + b \cdot n & \text{if } n > 2 
\end{cases}
\]

For a node with input size \( k \): work at this node is \( b \cdot k \).

<table>
<thead>
<tr>
<th>depth</th>
<th>nodes</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>( n )</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>( n/3 )</td>
</tr>
<tr>
<td>( i )</td>
<td>( 3^i )</td>
<td>( n/3^i )</td>
</tr>
</tbody>
</table>

Thus the work at depth \( i \) is \( 3^i \cdot b \cdot n/3^i = b \cdot n \).

The height of the tree is \( \log_3 n \). Thus \( T(n) \) is \( O(n \cdot \log_3 n) \).
Guess-and-Test Method

The guess-and-test method works as follows:

- we guess a solution (or an upper bound)
- we prove that the solution is true by induction

Example

\[ T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
T(n/2) + 2 \cdot n & \text{if } n > 1 
\end{cases} \]

Guess: \( T(n) \leq 2 \cdot n \)

- for \( n = 1 \) it holds \( T(n) = 1 \leq 2 \cdot 1 \)
- for \( n > 1 \) we have:
  \[
  T(n) = T(n/2) + 2 \cdot n \leq 2 \cdot n/2 + 2 \cdot n = 3 \cdot n
  \]

Wrong guess: we cannot make \( 3 \cdot n \) smaller or equal to \( 2 \cdot n \).
Example, continued

\[ T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
T(n/2) + 2 \cdot n & \text{if } n > 1 
\end{cases} \]

New guess: \( T(n) \leq 4 \cdot n \)

- for \( n = 1 \) it holds \( T(n) = 1 \leq 4 \cdot 1 \)
- for \( n > 1 \) we have:
  \[ T(n) = T(n/2) + 2 \cdot n \leq 4 \cdot n/2 + 2 \cdot n \quad \text{(by induction hypothesis)} \]
  \[ = 4 \cdot n \]

This time the guess was good: \( 4 \cdot n \leq 4 \cdot n \). Thus \( T(n) \leq 4 \cdot n \).
Example: Quick-Select with Median of Median

\[
T(n) = \begin{cases} 
    b & \text{if } n \leq 1 \\
    T(0.7 \cdot n) + T(0.2 \cdot n) + 2 \cdot b \cdot n & \text{if } n > 1 
\end{cases}
\]

Guess: \( T(n) \leq 20 \cdot b \cdot n \)

- for \( n = 1 \) it holds \( T(n) = b \leq 20 \cdot b \cdot 1 \)

- for \( n > 1 \) we have:

\[
T(n) = T(0.7 \cdot n) + T(0.2 \cdot n) + 2 \cdot b \cdot n \\
\leq 0.7 \cdot 20 \cdot b \cdot n + 0.2 \cdot 20 \cdot b \cdot n + 2 \cdot b \cdot n \quad \text{(by IH)}
\]
\[
= 0.9 \cdot 20 \cdot b \cdot n + 2 \cdot b \cdot n = 18 \cdot b \cdot n + 2 \cdot b \cdot n
\]
\[
= 20 \cdot b \cdot n
\]

Thus the guess was good.
This shows that quick-select with median of median is \( O(n) \).
Many divide-and-conquer recurrence equations have the form:

\[ T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases} \]

**Theorem (The Master Theorem)**

1. if \( f(n) \) is \( O(n^{\log_b a - \epsilon}) \), then \( T(n) \) is \( \Theta(n^{\log_b a}) \)
2. if \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \log^{k+1} n}) \)
3. if \( f(n) \) is \( \Omega(n^{\log_b a + \epsilon}) \), then \( T(n) \) is \( \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).
Master Method: Example 1

\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

Theorem (The Master Theorem)

1. if \( f(n) \) is \( O(n^{\log_b a - \varepsilon}) \), then \( T(n) \) is \( \Theta(n^{\log_b a}) \)
2. if \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \log^{k+1} n}) \)
3. if \( f(n) \) is \( \Omega(n^{\log_b a + \varepsilon}) \), then \( T(n) \) is \( \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

Example

\[ T(n) = 4 \cdot T(n/2) + n \]

Solution: \( \log_b a = 2 \), thus case 1 says \( T(n) = \Theta n^2 \).
Master Method: Example 2

\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

Theorem (The Master Theorem)

1. If \( f(n) \) is \( O(n^{\log_b a - \varepsilon}) \), then \( T(n) \) is \( \Theta(n^{\log_b a}) \)
2. If \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \log^{k+1} n}) \)
3. If \( f(n) \) is \( \Omega(n^{\log_b a + \varepsilon}) \), then \( T(n) \) is \( \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

Example

\[ T(n) = 2 \cdot T(n/2) + n \cdot \log n \]

Solution: \( \log_b a = 1 \), thus case 2 says \( T(n) = \Theta n \log^2 n \).
Master Method: Example 3

\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

**Theorem (The Master Theorem)**

1. *if* \( f(n) \) *is* \( O(n^{\log_b a - \varepsilon}) \), *then* \( T(n) \) *is* \( \Theta(n^{\log_b a}) \)
2. *if* \( f(n) \) *is* \( \Theta(n^{\log_b a \log^k n}) \), *then* \( T(n) \) *is* \( \Theta(n^{\log_b a \log^{k+1} n}) \)
3. *if* \( f(n) \) *is* \( \Omega(n^{\log_b a + \varepsilon}) \), *then* \( T(n) \) *is* \( \Theta(f(n)) \), *provided* \( af(n/b) \leq \delta f(n) \) *for some* \( \delta < 1 \).

**Example**

\[ T(n) = T(n/3) + n \cdot \log n \]

Solution: \( \log_b a = 0 \), thus case 3 says \( T(n) = \Theta(n \log n) \).
Master Method: Example 4

\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

Theorem (The Master Theorem)

1. if \( f(n) \) is \( O(n^{\log_b a - \varepsilon}) \), then \( T(n) \) is \( \Theta(n^{\log_b a}) \)
2. if \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \log^{k+1} n}) \)
3. if \( f(n) \) is \( \Omega(n^{\log_b a + \varepsilon}) \), then \( T(n) \) is \( \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

Example

\[ T(n) = 8 \cdot T(n/2) + n^2 \]

Solution: \( \log_b a = 3 \), thus case 1 says \( T(n) = \Theta n^3 \).
Master Method: Example 5

\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

**Theorem (The Master Theorem)**

1. *if* \( f(n) \) *is* \( O(n^{\log_b a - \varepsilon}) \), *then* \( T(n) \) *is* \( \Theta(n^{\log_b a}) \)

2. *if* \( f(n) \) *is* \( \Theta(n^{\log_b a \log^k n}) \), *then* \( T(n) \) *is* \( \Theta(n^{\log_b a \log^{k+1} n}) \)

3. *if* \( f(n) \) *is* \( \Omega(n^{\log_b a + \varepsilon}) \), *then* \( T(n) \) *is* \( \Theta(f(n)) \), *provided* \( af(n/b) \leq \delta f(n) \) *for some* \( \delta < 1 \).

**Example**

\[ T(n) = 9 \cdot T(n/3) + n^3 \]

Solution: \( \log_b a = 2 \), thus case 3 says \( T(n) = \Theta n^3 \).
Master Method: Example 6

\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

**Theorem (The Master Theorem)**

1. If \( f(n) \) is \( O(n^{\log_b a - \epsilon}) \), then \( T(n) \) is \( \Theta(n^{\log_b a}) \)

2. If \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \log^{k+1} n}) \)

3. If \( f(n) \) is \( \Omega(n^{\log_b a + \epsilon}) \), then \( T(n) \) is \( \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

**Example**

\[ T(n) = T(n/2) + 1 \quad \text{binary search} \]

Solution: \( \log_b a = 0 \), thus case 2 says \( T(n) = \Theta \log n \).
Master Method: Example 7

\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

**Theorem (The Master Theorem)**

1. If \( f(n) \) is \( O(n^{\log_b a - \varepsilon}) \), then \( T(n) \) is \( \Theta(n^{\log_b a}) \).
2. If \( f(n) \) is \( \Theta(n^{\log_b a \log^k n}) \), then \( T(n) \) is \( \Theta(n^{\log_b a \log^{k+1} n}) \).
3. If \( f(n) \) is \( \Omega(n^{\log_b a + \varepsilon}) \), then \( T(n) \) is \( \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

**Example**

\[ T(n) = 2 \cdot T(n/2) + \log n \quad \text{heap construction} \]

Solution: \( \log_b a = 0 \), thus case 1 says \( T(n) = \Theta(n) \).
Algorithm to multiply two $n$-bit integers $A$ and $B$:

- **Divide**: split $A$, $B$ in $n/2$ higher-order and lower-order bits

\[ A = A_h \cdot 2^{n/2} + A_\ell \]
\[ B = B_h \cdot 2^{n/2} + B_\ell \]

We can define $A \cdot B$ as follows:

\[ A \cdot B = (A_h \cdot 2^{n/2} + A_\ell) \cdot (B_h \cdot 2^{n/2} + A_\ell) \]
\[ = A_h \cdot B_h \cdot 2^n + A_h \cdot B_\ell \cdot 2^{n/2} + A_\ell \cdot B_h \cdot 2^{n/2} + A_\ell \cdot B_\ell \]

So, $T(n) = 4 \cdot T(n/2) + n$ which implies $T(n) \in O(n^2)$. Can we do better? . . .
Algorithm to multiply two $n$-bit integers $A$ and $B$:

- **Divide:** split $A$, $B$ in $n/2$ higher-order and lower-order bits
  \[
  A = A_h \cdot 2^{n/2} + A_l \\
  B = B_h \cdot 2^{n/2} + B_l
  \]

We use a different way to multiply the parts:

\[
A \cdot B = A_h \cdot B_h \cdot 2^n + (A_h \cdot B_l + A_l \cdot B_h) \cdot 2^{n/2} + A_l \cdot B_l
= A_h \cdot B_h \cdot 2^n
+ [(A_h - A_l) \cdot (B_l - B_h) + A_h \cdot B_h + A_l \cdot B_l] \cdot 2^{n/2}
+ A_l \cdot B_l
\]

Here $T(n) = 3T(n/2) + n$, hence $T(n) \in O(n^{\log_2 3}) = O(n^{1.585})$.

(note that we need to calculate $A_h \cdot B_h$ and $A_l \cdot B_l$ only once)
The Greedy Method

An optimization problem consists of
▶ a set of configurations (different choices), and
▶ an objective function (a score assigned to configurations).

We search a configuration that maximizes (or minimizes) the objective function.

The **greedy method** for solving optimization problems:
▶ tries to find the global optimum (or come close to it) by iteratively selecting the locally optimal choice.

That is, in every step we choose the currently best possible.
Problem: return coins for a given amount of money.
  ▶ configurations: coins returned and money left to return
  ▶ objective function: minimize the number of coins returned

Greedy solution: always return the largest coin possible.

Example

The coins are valued 8, 4, 1. We give change for 18:
  ▶ we pick 8, 10 left to pay
  ▶ we pick 8, 2 left to pay
  ▶ we pick 1, 1 left to pay
  ▶ we pick 1, 0 left to pay

Thus the greedy solution is 8, 8, 1, 1.
Example

The coins are valued 4, 3, 1. We give change for 6:

▶ we pick 4, 2 left to pay
▶ we pick 1, 1 left to pay
▶ we pick 1, 0 left to pay

Thus the greedy solution is 4, 1, 1.

But the optimal solution would be 3, 3:

▶ choosing currently best not always leads to global optimum

The above example does not have the greedy-choice property:

A problem has the greedy-choice property if beginning from the start configuration the sequence of locally optimal choices leads to the globally optimal solution.
The Fractional Knapsack Problem

Given: a set S of n items where each item i has:

- $b_i$ = a positive benefit
- $w_i$ = a positive weight

**Fractional Knapsack Problem**

- choose fractions $x_i \leq w_i$ of items with maximal total benefit:
  \[
  \sum_{i \in S} b_i \cdot \frac{x_i}{w_i} \quad \text{(objective to maximize)}
  \]

- with weight at most W:
  \[
  \sum_{i \in S} x_i \leq W \quad \text{(constraint)}
  \]
Example

You found a treasure with:
- 50kg jewels, value 1 Million Euros
- 1kg chewing gum, value 20 Euros
- 5kg diamonds, value 5 million Euros
- 10kg gold, value 500,000 Euros

Your backpack can carry only 20kg!

What do you take with you?
- Of course the highest value per weight:
  - 5kg diamonds (value/kg = 1 million Euro)
  - 10kg gold (value/kg = 0.05 million Euro)
  - 5kg jewels (value/kg = 0.02 million Euro)

Your backpack is filled with the maximum value 5.6 million Euro.
The Fractional Knapsack Algorithm

Greedy choice:
- keep taking item with the highest benefit to weight ratio: \( \frac{b_i}{w_i} \)
- run time: \( O(n \log n) \) (for sorting by this ratio)

\[
\text{fractionalKnapsack}(S, \vec{b}, \vec{w}, W):
\]
\[
\text{for each } i \in S \text{ do}
\]
\[
x_i = 0
\]
\[
v_i = \frac{b_i}{w_i}
\]

sort \( S \) such that elements are descending w.r.t. \( v_i \)
\[
\text{while } W > 0 \text{ do}
\]
\[
i = S.\text{remove}(S.\text{first}())
\]
\[
x_i = \min(w_i, W)
\]
\[
W = W - x_i
\]
\[
done
\]
The Fractional Knapsack Algorithm

Correctness: suppose there is a better solution \( A \).

- Then \( A \) did not always choose the highest \( v_j \).
- Thus there exist items \( i, j \) in \( A \) such that:

\[
\begin{align*}
x_i &> 0 \\
v_i &< v_j \\
x_j &< w_j
\end{align*}
\]

Let

\[
a = \min(x_i, w_j - x_j)
\]

But then we could replace amount \( a \) of item \( i \) with item \( j \), which would increase the total benefit.

Thus the solution cannot be better than the greedy choice.

The fract. knapsack problem has the greedy-choice property!
Task Scheduling

Given: a set $T$ of $n$ tasks, each having:

- a start time $s_i$
- a finish time $f_i$ (where $s_i < f_i$)

Goal: perform all tasks using a minimal number of machines

\[ \begin{align*}
  &\text{Machine 1} \\
  &\text{Machine 2} \\
  &\text{Machine 3}
\end{align*} \]
Task Scheduling Algorithm

Greedy choice:
- keep taking the task with smallest start time
- assign task to free machine if possible
  (if all machines are busy take a new machine)

\texttt{taskSchedule}(T, \vec{s}, \vec{f}):
\begin{align*}
    m &= 0 \quad \text{number of machines} \\
    \text{sort } T \text{ such that elements are ascending w.r.t. } s_i \\
    \text{while } \neg T.\text{isEmpty}() \text{ do} \\
    &\quad i = T.\text{remove}(T.\text{first}()) \\
    &\quad \text{if a machine } j \leq m \text{ has time for task } i \text{ then} \\
    &\quad \quad \text{schedule } i \text{ on machine } j \\
    &\quad \text{else} \\
    &\quad \quad m = m + 1 \\
    &\quad \quad \text{schedule } i \text{ on machine } m \\
    \text{done}
\end{align*}
Given: a set $T$ of tasks:

$[1, 4], [1, 3], [2, 5], [3, 7], [4, 7], [6, 9], [7, 8]$

(ordered by starting time)
Task Scheduling Algorithm

Correctness: suppose there is a better schedule.

- **Assume:**
  - the greedy algorithm uses \( m \) machines,
  - there exists a schedule using less than \( m \) machines.

- Let \( i \) be the first task scheduled on machine \( m \).

- Then on each of the machines \( 1, \ldots, m - 1 \) there runs a task with starting time \( \leq s_i \) and finishing time \( f_i > s_i \).

- All these tasks conflict with \( i \) and with each other.

Hence, there cannot be a schedule with less than \( m \) machines.
Further applications of greedy algorithms:
- string compression (construction of Huffman codes)
- shortest path in graphs
- minimal spanning trees
- ...
Dynamic Programming

**Dynamic programming** is an algorithm design paradigm:
- used for optimization problems
- solving complex problems by splitting into smaller parts
- optimal solutions of subproblems are combined to globally optimal solution

Sounds a bit like divide-and-conquer, but:
- not necessarily recursive
- subproblems are allowed to overlap
- a suitable definition of subproblems can be chosen

Also does not tell much... we consider examples...
Maximum Subarray Problem

Given: array $A$ of integers (can be negative).

**Algorithm** $\text{maxSubarray}(A, n)$:

- **Input:** array $A$ containing $n$ integers
- **Output:** maximum subarray sum

$$max = 0$$

for $left = 0$ to $n - 1$ do

$$sum = 0$$

for $right = left$ to $n - 1$ do

$$sum = sum + A[right]$$

if $sum > max$ then $max = sum$

done

done

return $max$

The naive algorithm is $O(n^2)$.
Maximum Subarray Algorithm

How to split into subproblems?

- let $B[r]$ be the maximum sum of a subarray ending at rank $r$

Let $B[0] = \max(A[0], 0)$.

We can compute $B[r]$ from $B[r - 1]$ as follows:

$$B[r] = \max(0, B[r - 1] + A[r])$$

That is, the maximal subarray ending at $r$ is either:

- the maximal subarray ending at $r - 1$ plus element $A[r]$, or
- the empty subarray ending at $r$.

The maximum subarray sum is the maximum of the $B[i]$’s.

This gives rise to a linear time algorithm...
Maximum Subarray Algorithm

Kadane’s algorithm for the maximum subarray problem:

**Algorithm** `maxSubarray(A, n)`:

**Input:** array $A$ containing $n$ integers

**Output:** maximum subarray sum

1. $B = \text{new array of length } n$
2. $B[0] = \max(A[0], 0)$
3. $max = B[0]$
4. **for** $r = 1$ **to** $n - 1$ **do**
   1. $B[r] = \max(0, B[r - 1] + A[r])$
   2. $max = \max(max, B[r])$
5. **done**
6. **return** $max$

This algorithm computes the maximal subarray sum in $O(n)$. 
Making Change

Problem: return coins for a given amount of money.
- configurations: coins returned and money left to return
- objective function: minimize the number of coins returned

Let $C$ be the set of coins, e.g. $C = \{1, 3, 4\}$.

How to split into subproblems?
- let $B[w]$ be the best solution for making change for $w$
  (that is the minimal number of coins for making $w$)

Let $B[0] = 0$. Compute $B[w]$ from $B[0], \ldots , B[w-1]$ as follows:

$$B[w] = 1 + \min \{B[w-c] \mid c \in C \text{ s.t. } c \leq w\}$$

That is, for the minimal number of coins for $w$ we need:
- a coin $c$ from $C$ plus the minimal number of coins for $w - c$
Let $C = \{1, 3, 4\}$, we make change for 6.

Let $B[0] = \emptyset$.

- $B[1] = 1 + \max \{B[0] \text{ (for } c = 1)\} = 1$
- $B[2] = 1 + \max \{B[1] \text{ (for } c = 1)\} = 2$
- $B[3] = 1 + \max \{B[2] \text{ (for } c = 1), B[0] + 1 \text{ (for } c = 3)\} = 1$
- $B[4] = 1 + \max \{B[3] \text{ (for } c = 1), B[1] \text{ (for } c = 3), B[0] \text{ (for } c = 4)\} = 1$

Thus we need 2 coins to make change for 6.

If we want to know which coins are needed to make change for $w$, we have to remember which coin was added in each step. (for example: coin 3 was chosen for $B[6]$, and 3 for $B[3]$)
The 0/1 Knapsack Problem

Given: a set $S$ of $n$ items where each item $i$ has:

- $b_i$ = a positive benefit
- $w_i$ = a positive weight

0/1 Knapsack Problem

- choose items $T \subseteq S$ with maximal total benefit:
  \[ \sum_{i \in T} b_i \] (objective to maximize)
- with weight at most $W$:
  \[ \sum_{i \in T} w_i \leq W \] (constraint)

Thus now each item is either accepted or rejected entirely.
You found a treasure with:

- a 7kg gold bar,
- a 5kg gold bar, and
- a 4kg gold bar.

The value of the gold coincides with the weight. Your backpack can carry only 10kg!

We cannot split the gold bars. What do you take with you?

- The greedy approach would take the 7kg gold bar.
- However, better is: 5kg, 4kg with total weight 9kg.
The 0/1 Knapsack Algorithm

How to split into subproblems?

- number items from $1, \ldots, n$
- let $S_k$ be the the set of elements $1, \ldots, k$
- let $B[k, w]$ be the best selection from $S_k$ with weight $\leq w$

Thus the $B[k, w]$ are the solutions of subproblems.

We compute $B[k, w]$ from $B[k - 1, 0], \ldots, B[k - 1, w]$ as follows:

$$B[k, w] = \begin{cases} 
B[k - 1, w], & \text{if } w_k > w \\
\max(B[k - 1, w], B[k - 1, w - w_k] + b_k) & \text{otherwise}
\end{cases}$$

That is, the best subset of $S_k$ with weight $\leq w$ is either:

- the best subset of $S_{k-1}$ with weight $\leq w$, or
- the best subset of $S_{k-1}$ with weight $\leq w - w_k$ plus item $k$
Example

We have:
- item 1 with weight 3 and benefit 9 \((b_1/w_1 = 3)\)
- item 2 with weight 2 and benefit 5 \((b_2/w_2 = 2.5)\)
- item 3 with weight 2 and benefit 5 \((b_3/w_3 = 2.5)\)

Maximal total weight is \(w = 4!\)

We run the algorithm:
- \(B[0,0] = 0, B[0,1] = 0, B[0,2] = 0, B[0,3] = 0, B[0,4] = 0\)
- \(B[1,0] = 0, B[1,1] = 0, B[1,2] = 0, B[1,3] = 9, B[1,4] = 9\)

Thus the best benefit for weight 4 is \(B[3,4] = 10\).
Let $A$, $B$ be matrices:

- let $A$ have dimension $d \times e$
- let $B$ have dimension $e \times f$

Then $C = A \cdot B$ has dimension $d \times f$.

The (naive) algorithm for computing $C = A \cdot B$:

$$C[r, c] = \sum_{i=0}^{e-1} A[r, i] \cdot B[i, c]$$

takes $O(d \cdot e \cdot f)$ time.

(we multiply every row of $A$ with every column of $B$)
Matrix Chain-Products

Matrix chain-product:

- Compute $A_0 \cdot A_1 \cdots A_{n-1}$.
- Let $A_i$ have dimension $d_i \times d_{i+1}$.

Problem: how to put parentheses?

Example:

- $B$ is $3 \times 100$ (height $\times$ width)
- $C$ is $100 \times 5$
- $D$ is $5 \times 5$

Then

- $(B \cdot C) \cdot D$ takes $1500 + 75 = 1575$ operations
- $B \cdot (C \cdot D)$ takes $2500 + 1500 = 4000$ operations
Attempt 1: Enumeration Approach

Idea of the enumeration (brute-force) approach:

- Try all possible ways to parenthesize $A_0 \cdots A_{n-1}$.
- Calculate the number of operations for each of them.
- Pick the best solution.

Running time:

- As many ways to parenthesize as binary trees with $n$ leaves.
- This is exponential, almost $4^n$.

Thus the algorithm is terribly slow.
Attempt 2: Greedy Approach

Idea: select the product that uses the fewest operations.

Example where the greedy approach fails:

- $A$ is $2 \times 1$
- $B$ is $1 \times 2$
- $C$ is $2 \times 3$

Then:

- $A \cdot B$ takes 4 operations
- $B \cdot C$ takes 6 operations

Thus the greedy method picks $(A \cdot B) \cdot C$. However:

- $(A \cdot B) \cdot C$ takes $4 + 12 = 16$ operations
- $A \cdot (B \cdot C)$ takes $6 + 6 = 12$ operations

Thus the greedy method does not yield the optimal result.
What are the subproblems?

- best parenthesization of $A_i \cdot A_{i+1} \cdots A_j$.

Let $N[i, j]$ be the number of operations for this subproblem.

The global optimum can be defined from optimal subproblems:

- Recall that $A_i$ has dimension $d_i \times d_{i+1}$.
- Thus we can define:

$$
N[i, j] = \min_{i \leq k < j} \left\{ N[i, k] + N[k + 1, j] + d_i \cdot d_{k+1} \cdot d_{j+1} \right\}
$$

Note that the subproblems are can overlap:

- e.g. $N[1, 4]$ and $N[2, 6]$ overlap
Solution: Dynamic Programming Algorithm

Matrix chain-products with running time $O(n^3)$:

- We start with $N[i, i] = 0$ (nothing to multiply there).
- Then for $s = 2, 3, \ldots$ we compute problems of size $s$.
  (that is, the problems $N[i, i + s - 1]$)

```plaintext
matrixChain(d_0, d_1, \ldots, d_{n-1}):
    for i = 0 to n - 1 do N[i, i] = 0 done
    for s = 2 to n do
        for i = 0 to n - s do
            j = i + s - 1
            N[i, j] = +\infty
            for k = i to j - 1 do
                ops = N[i, k] + N[k + 1, j] + d_i \cdot d_{k+1} \cdot d_{j+1}
                N[i, j] = \min(N[i, j], ops)
            done
        done
    done
done
done
```
Example

Let \(d_0 = 2, d_1 = 1, d_2 = 2, d_3 = 4, d_4 = 3\).

- \(A_0\) is \(2 \times 1\), \(A_1\) is \(1 \times 2\), \(A_2\) is \(2 \times 4\), \(A_3\) is \(4 \times 3\).

We run the algorithm:

- \(N[0, 0] = 0, N[1, 1] = 0, N[2, 2] = 0, N[3, 3] = 0\)
- \(N[0, 1] = 4, N[1, 2] = 8, N[2, 3] = 24\)
- \(N[0, 2] = \min(0 + 8 + 8, 4 + 0 + 16) = 16, N[1, 3] = \min(0 + 24 + 6, 8 + 0 + 12) = 20\)
- \(N[0, 4] = \min(0 + 20 + 6, 4 + 24 + 12, 16 + 0 + 24) = 26\)

Thus the optimal solution is: \(A_0 \cdot ((A_1 \cdot A_2) \cdot A_3)\).
A **graph** is a pair \((V, E)\) where:

- \(V\) is a set of **nodes**, called **vertices**
- \(E\) is a set of pairs of nodes, called **edges**

Both the nodes and edges can store elements (**labels**).

**Example:**
- Nodes: \(U, X, Z\)
- Edges: \((U, X), (X, Z), (Z, X), (Z, Z)\)
Directed edge:
- ordered pair of vertices $(u, v)$
- first vertex $x$ is the origin
- second vertex $y$ is the destination

Directed graph: all edges are directed.

Undirected edge:
- unordered pair of vertices $(u, v)$

Undirected graph: all edges are undirected.
Applications of Graphs

- Electronic circuits.
- Transportation networks:
  - street network (Google maps)
  - flight network
- Computer networks.
- ...
Graphs: Terminology

- **Start- and endpoint:**
  - $U$ is start-point of $a$
  - $V$ is end-point of $a$

- **Edges:**
  - $a$ is outgoing edge of $U$
  - $a$ is incoming edge of $V$

- **Degree of a vertex (number of edges):**
  - The degree $\text{deg}(X)$ of $X$ is 5

- **Self-loops:**
  - $j$ is a self-loop
Graphs: Terminology (continued)

- **Path:**
  - sequence of alternating vertices and edges
  - begins and ends with vertices
  - each edge is preceded by its start-point and followed by its end-point
  - e.g. $U \overset{c}{\rightarrow} W \overset{e}{\rightarrow} X \overset{g}{\rightarrow} Y \overset{f}{\rightarrow} W \overset{d}{\rightarrow} V$

- **Cycle:**
  - path the first and last vertex are equal

A path is simple if all vertices are distinct (except first & last).
Implementing Graphs

Vertex stores:
  ▶ element

Edge stores:
  ▶ element
  ▶ start-point and end-point

**Edge List Structure**
Graph is stored as list of vertexes and list of edges.

**Adjacency List Structure**
Graph is stored as list of vertexes. Each vertex stores:
  ▶ element
  ▶ list of (outgoing and incoming) edges
Implementing Graphs: Performance

Assume that we have a graph with \( n \) vertices, and \( m \) edges.

<table>
<thead>
<tr>
<th>Function</th>
<th>Edge List</th>
<th>Adjacency List</th>
</tr>
</thead>
<tbody>
<tr>
<td>outgoingEdges((v))</td>
<td>( O(m) )</td>
<td>( O(\text{deg}(v)) )</td>
</tr>
<tr>
<td>areAdjacent((v, w))</td>
<td>( O(m) )</td>
<td>( O(\min(\text{deg}(v), \text{deg}(w))) )</td>
</tr>
<tr>
<td>insertVertex((o))</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>insertEdge((v, w, o))</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>removeVertex((v))</td>
<td>( O(m) )</td>
<td>( O(\text{deg}(v)) )</td>
</tr>
<tr>
<td>insertEdge((e))</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>
In a **weighted graph** each edge has an associated number called **weight**.

The weight can represent distances, costs, ...
Shortest Path Problem

Given: weighted graph and vertices $u, v$

- find the shortest path between $u$ and $v$
- shortest means ‘minimal total weight’

Example: shortest path from $U$ to $Z$ has weight 8
Dijkstra’s Algorithm

The distance of a vertex $v$ from a vertex $s$:

- minimal weight of a path from $v$ to $s$

Dijkstra’s algorithm computes distances (shortest paths):

- from a given start vertex $s$ to all other vertices
- assumptions: the edge weights are non-negative

Dijkstra’s algorithm:

- For every vertex $v$ we store the estimated distance $d(v)$. (initially $d(s) = 0$ and $d(v) = \infty$ for all other vertices $v$)
- We start with the set of vertices $S = \{s\}$ and in every step:
  - add to $S$ the vertex $v$ outside of $S$ with smallest $d(v)$
  - update distances of direct neighbours $n$ of $v$:
    
    $d(n) = \min(d(n), d(v) + \text{weight of the edge from } v \text{ to } n)$
Dijkstra’s Algorithm: Example

Example: shortest paths from $U$

First we add $U$ to $S$, and update the distances neighbours of $U$. 
Example: shortest paths from $U$

Now $W$ is the closest node outside $S$:
- we add $W$ to $S$ and update the distances of its neighbours
Dijkstra’s Algorithm: Example

Example: shortest paths from $U$

Now $V$ is the closest node outside $S$:
- we add $V$ to $S$ and update the distances of its neighbours
Example: shortest paths from $U$

Now $X$ is the closest node outside $S$:
- we add $X$ to $S$ and update the distances of its neighbours
Example: shortest paths from $U$

Now $Y$ is the closest node outside $S$:
- we add $Y$ to $S$ and update the distances of its neighbours
Example: shortest paths from $U$

Now $Z$ is the closest node outside $S$:
- we add $Z$ to $S$ and update the distances of its neighbours
Dijkstra’s Algorithm: Example

Example: shortest paths from \( U \)

We obtain a tree of all shortest paths from \( U \) to all other nodes. (e.g. the shortest path from \( U \) to \( Z \) has length 8)
Dijkstra’s Algorithm

Assume that we have a graph with $n$ vertices, and $m$ edges.
- A priority queue stores (heap) vertices outside of $S$:
  - key: distance
  - element: vertex
- Each vertex $v$ stores distance $d(v)$ and link to heap node.

Algorithm on the following slide...
Dijkstra’s Algorithm

\textbf{DijkstraDistances}(G, s):

- \(Q = \) new heap-based priority queue
  (set all distances to \(\infty\) except for \(s\) with distance 0)

\textbf{for each} \(v \in G.\text{vertices}()\) \textbf{do}

- \(v.\text{setDistance}(0)\) \textbf{if} \(v == s\) \textbf{then}
  \(v.\text{setDistance}(\infty)\) \textbf{else}
  \(h = Q.\text{insert}(v.\text{getDistance}(), v)\) \textbf{(returns heap node)}
  \(v.\text{setHeapNode}(h)\)

\textbf{while} \(-Q.\text{isEmpty}()\) \textbf{do}

- \(v = Q.\text{removeMin}()\)
  (update the distance of all neighbours)

\textbf{for each} \(e \in v.\text{outgoingEdges}()\) \textbf{do}

- \(w = e.\text{endPoint}()\)
  \(d = v.\text{getDistance}() + e.\text{getWeight}()\)
  \textbf{if} \(d < w.\text{getDistance}()\) \textbf{then}
    \(w.\text{setDistance}(d)\)
    \(Q.\text{replaceKey}(w.\text{getHeapNode}(), d)\)

\textbf{done}
Dijkstra’s Algorithm: Performance

Assume that we have a graph with \( n \) vertices, and \( m \) edges.

- The first loop is executed \( n \) times:
  - \( n \) times inserting in priority queue \( (O(\log n)) \)

- The second loop is executed \( n \) times:
  - \( n \) times removing from the priority queue \( (O(\log n)) \)

- The third loop is executed for every edge once \( (m \) times):
  - \( m \) times updating a key in the priority queue \( (O(\log n)) \)

In total we have: \( (n \cdot \log n) + (n \cdot \log n) + (m \cdot \log n) \).

- \( O((n + m) \cdot \log n) \)
  (if we use the adjacency list structure)

Dijkstras algorithm is \( O(m \cdot \log n) \) (graph is connected).
Dijkstra’s algorithm is based on the greedy method:
▶ it adds vertices by increasing size

Suppose it would not work:
▶ Let \( x \) be the closest vertex with wrongly assigned distance.
▶ Let \( y \) be the previous node on the shortest path from \( s \) to \( x \).
▶ Then \( y \) is closer, and has correct distance:
   ▶ thus \( y \) has been processed before \( x \), and
   ▶ then \( x \) must have been assigned the correct distance.

Hence there cannot be a wrong vertex. The algorithm works!
Why It Doesn’t Work For Negative Weights

Example: shortest path from $X$

We have processed all nodes, but $V$ has the wrong distance! The problem with negative weights is:

- The nodes on a shortest path do not always have increasing distance.
Example: shortest path from $X$
Why It Doesn’t Work For Negative Weights

Example: shortest path from $X$

We have processed all nodes, but $V$ has the wrong distance!

The problem with negative weights problem is:

- The nodes on a shortest path do not always have increasing distance.
Why It Doesn’t Work For Negative Weights

Example: shortest path from $X$

The problem with negative weights is:

▶ The nodes on a shortest path do not always have increasing distance.
Why It Doesn’t Work For Negative Weights

Example: shortest path from $X$

![Graph with nodes and edges labeled with weights]

We have processed all nodes, but $V$ has the wrong distance!

The problem with negative weights problem is:

- The nodes on a shortest path do not always have increasing distance.
Bellman-Ford Algorithm

Bellman-Ford algorithm:

- works with negative weights
- iteration $i$ finds shortest paths of length $i$

BellmanFord($G, s$):

```plaintext
for each $v \in G$.vertices() do
    if $v == s$ then $v$.setDistance(0) else $v$.setDistance($\infty$)
for $i = 1$ to $n - 1$ do
    for each $e \in G$.edges() do
        $v = e$.startPoint()
        $w = e$.endPoint()
        $d = v$.getDistance() + $e$.getWeight()
        if $d < w$.getDistance() then
            $w$.setDistance($d$
        done
```
The nodes are labeled with the values $d(v)$.
All-Pair Shortest Paths

Find the distance between every pair of vertices:

- $n$ calls to Dijkstra’s algorithm is $O(nm \log n)$ time
- $n$ calls to Bellman-Ford is $O(n^2 m)$ time

We can do it in $O(n^3)$ as follows (assumes nodes are 1, \ldots, $n$):

**AllPair($G$):**

```
for each vertex pairs $(i,j)$ do
  if $i \neq j$ and $(i,j)$ is an edge in $G$ then
    $D_0[i,j] =$ weight of edge $(i,j)$
  else
    if $i == j$ then $D_0[i,j] = 0$ else $D_0[i,j] = \infty$
  for $k = 1$ to $n$ do
    for $i = 1$ to $n$ do
      for $j = 1$ to $n$ do
        $D_k[i,j] = \min(D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j])$
      return $D_n$;
```
### Example

**Graph:**
- Nodes: 1, 2, 3, 4
- Edges: 1-2, 2-3, 3-4, 4-1

**Distance Matrix:**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>∞</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Initial Distance Matrices (k = 1, 2, 3, 4):**

#### k = 1

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>∞</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

#### k = 2

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>∞</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

#### k = 3

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>∞</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

#### k = 4

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Minimum Spanning Trees

Spanning tree $T$ of a weighted graph $G$:

- $T$ contains all nodes and a subset of the edges of $G$
- $T$ is a tree

Spanning tree is minimal if the total edge weight is minimal.

Application:
- Communication networks.
- Transportation networks.
Cycle Property

Cycle property:

- Let $T$ be a minimal spanning tree for a weighted graph $G$.
- Let $e$ be an edge that is not in $T$.
- Let $C$ be the cycle formed by $e$ and $T$.

Then for every edge $f$ of $C$ we have $\text{weight}(f) \leq \text{weight}(e)$.

Proof:

- Assume $\text{weight}(f) > \text{weight}(e)$
- Replacing $f$ by $e$ in $T$ would yield a better spanning tree

Example: $8 \geq 4, 1, 6$. 
Partition property:

- Consider a partition of the vertices in \( A \) and \( B \).
- Let \( e \) be a minimum edge between \( A \) and \( B \).

Then there is a minimum spanning tree containing \( e \).

Proof: let \( T \) be an MST

- if \( e \not\in T \) then it must create a cycle \( C \) with \( T \); let \( f \) be an edge of \( C \) between \( A \) and \( B \)
- by the cycle property \( \text{weight}(f) \leq \text{weight}(e) \)
- thus \( \text{weight}(f) = \text{weight}(e) \)
- replacing \( f \) with \( e \) yields MST
Prim-Jarnik’s Algorithm

Similar to Dijkstra’s algorithm:

- we start with a node $s$, $S = \{s\}$
- build the MST while stepwise extending a set $S$

But something is changed:

- nodes do not store the distance to to $s$, but instead
- nodes store the distance to the cloud $S$

Prim-Jarnik’s algorithm: pick a node $s$

- For every vertex $v$ we store the distance $d(v)$ to $S$.
  (initially $d(s) = 0$ and $d(v) = \infty$ for all other vertices $v$)
- We start with the set of vertices $S = \{s\}$ and in every step:
  - add to $S$ the vertex $v$ outside of $S$ with smallest $d(v)$
  - update distances of direct neighbours $n$ of $v$:
    $$d(n) = \min(d(n), \text{ weight of the edge from } v \text{ to } n)$$
    (whenever $d(n)$ is changed we do $n\.setParent(v)$)
We compute the minimal spanning tree starting from $U$:
We compute the minimal spanning tree starting from $U$:
We compute the minimal spanning tree starting from $U$:
Example

We compute the minimal spanning tree starting from $U$:
We compute the minimal spanning tree starting from $U$: 

![Graph Diagram]

Prim-Jarnik’s algorithm is $O(m \cdot \log n)$.
Example

We compute the minimal spanning tree starting from $U$: 

![Graph diagram with labels and edges showing the minimal spanning tree from U to Z.]
We compute the minimal spanning tree starting from \( U \):

Prim-Jarnik’s algorithm is \( O(m \cdot \log n) \).

(same analysis as Dijkstra’s algorithm)
A string is a sequence of characters. Let $S$ be a string of size $m$:

- A substring of $S$ is a string of the form $S[i \ldots j]$.
- A prefix of $S$ is a string of the form $S[0 \ldots i]$.
- A suffix of $S$ is a string of the form $S[i \ldots (m - 1)]$.

The alphabet $\Sigma$ is the set of possible characters.

**Example**

Let $S = “acaabca”$:

- “abc” is a substring of $S$
- “acaa” is a prefix of $S$
- “ca” is a suffix of $S$
The pattern matching problem:

- given: strings $T$ (text) and $P$ (pattern)
- task: find a substring of $T$ equal to $P$

Example: find $P = \text{“ake”}$ in $T = \text{“Hey there, wake up!”}$.

Applications:
- text editors
- search engines
- biological research
- ...
Brute-Force Algorithm

The brute-force algorithm:

- compares $P$ with text $T$ for each possible shift of $P$

Let $n$ be the size of $T$ and $m$ the size of $P$.

```python
bruteForceMatch(T, P):
    for pos = 0 to n - m do
        match = true
        for i = 0 to m - 1 do
            if $P[i] \neq T[pos + i]$ then
                match = false
                break the for-i-loop
        if match then return pos  \ (match at position pos)
    return -1 \ (no match)
```

Worst case complexity: $O(n \cdot m)$

- Example for worst-case: $T = aaa \ldots aah, P = aaah$
Boyer-Moore Heuristics

Boyer-Moore’s pattern matching algorithm:

- **Looking-glass heuristic:**
  - compare $P$ with substring of $T$ backwards

- **Character-jump heuristic:** if mismatch occurs at $T[i] = c$
  - if $P$ does not contain $c$, shift $P$ to align $P[0]$ with $T[i + 1]$
  - else, shift $P$ to align the last occurrence of $c$ in $P$ with $T[i]$

Example: we search the pattern $P =$ “rithm” in

```
    pattern matching algorithm
    rithm rithm rithm
    rithm rithm rithm
    rithm rithm rithm
    rithm rithm rithm
    1  3  5  7
```

Last-Occurrence Function

The last-occurrence function $L(c)$:
- $L(c)$ maps letters $c \in \Sigma$ to integers $L(c)$ where:
  - $L(c)$ is the largest index $i$ such that $P[i] = c$, or
  - $L(c)$ is $-1$ if no such index exists.

Example:
- $\Sigma = \{a, b, c, d\}$
- $P = “abacab”$

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L(c)$</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

The last-occurrence function can be computed in $O(m + s)$ time:
- where $m$ is the size of $P$ and $s$ the size of $\Sigma$
- $L(c)$ can be represented as array where the numerical codes of letters $c \in \Sigma$ are the index
Boyer-Moore Algorithm

BoyerMooreMatch\((T, P, \Sigma)\):
\[
L = \text{lastOccurrenceFunction}(P, \Sigma)
\]
\[
pos = 0
\]
\[
\text{while } pos \leq n - m \text{ do}
\]
\[
match = \text{true}
\]
\[
\text{for } i = m - 1 \text{ downto } 0 \text{ do}
\]
\[
\text{if } P[i] \neq T[pos + i] \text{ then}
\]
\[
match = \text{false}
\]
\[
(\text{align last occurrence, but move at least 1})
\]
\[
pos = pos + \max(1, \ i - L(T[pos + i]))
\]
\[
\text{break the for-}i\text{-loop}
\]
\[
\text{done}
\]
\[
\text{if } match \text{ then return } pos \quad (\text{match at position } pos)
\]
\[
\text{done}
\]
\[
\text{return } -1 \quad (\text{no match})
\]
Example:

Example:

- $\Sigma = \{a, b, c, d\}$
- $P = "abacab"$

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L(c)$</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

`a b a c a a b c a d a b a c a b a a a b b`
Example:

- \( \Sigma = \{a, b, c, d\} \)
- \( P = \text{“abacab”} \)

<table>
<thead>
<tr>
<th>c</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(c)</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>-1</td>
</tr>
</tbody>
</table>

\[
\begin{array}{ccccccccccc}
\text{a} & \text{b} & \text{a} & \text{c} & \text{a} & \text{a} & \text{b} & \text{c} & \text{a} & \text{d} \\
\text{a} & \text{b} & \text{a} & \text{c} & \text{a} & \text{b} & \text{a} & \text{a} & \text{b} & \text{b} \\
\text{a} & \text{b} & \text{a} & \text{c} & \text{a} & \text{b} \\
\end{array}
\]
Example:

Example:

- $\Sigma = \{a, b, c, d\}$
- $P = "abacab"$

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L(c)$</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>
Example:

- $\Sigma = \{a, b, c, d\}$
- $P = \text{"abacab"}$

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L(c)$</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>
Example:

- $\Sigma = \{a, b, c, d\}$
- $P = "abacab"$
Example:

- $\Sigma = \{a, b, c, d\}$
- $P = \text{“abacab”}$

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L(c)$</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

Example:

```
abacab
```

```
abacab
```

```
abacab
```

```
abacab
```
Example:

- $\Sigma = \{a, b, c, d\}$
- $P = \text{“abacab”}$
Example:

- $\Sigma = \{a, b, c, d\}$
- $P = "abacab"$

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L(c)$</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:
```
  a b a c c a b c d a b a c a b a a a a b b
  a b a c a b
  a b a c a b
  a b a c a b
```
Example:

Example:

- $\Sigma = \{a, b, c, d\}$
- $P = "abacab"$

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L(c)$</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>-1</td>
</tr>
</tbody>
</table>

![Diagram showing examples of strings and positions with arrows indicating transitions between characters.](image)
Example:

- $\Sigma = \{a, b, c, d\}$
- $P = \text{"abacab"}$

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L(c)$</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>
Example:

- \( \Sigma = \{a, b, c, d\} \)
- \( P = \text{“abacab”} \)

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L(c) )</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>-1</td>
</tr>
</tbody>
</table>
Example:

- $\Sigma = \{a, b, c, d\}$
- $P = "abacab"$

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L(c)$</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>
Example:

- $\Sigma = \{a, b, c, d\}$
- $P = "abacab"$

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L(c)$</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

```
abacab
```

```
abacab
```

```
abacab
```

```
abacab
```

```
abacab
```

```
abacab
```

```
abacab
```

```
abacab
```
Example:

- $\Sigma = \{a, b, c, d\}$
- $P = "abacab"$

|   | a | b | a | c | a | a | b | c | a | d | a | b | a | c | a | b | a | a | b | b |
| L(c) | 4 | 5 | 3 | -1 |

![Diagram showing the process of generating a prefix code for the string P = "abacab" using the alphabet Σ = \{a, b, c, d\}.]
Example:

- $\Sigma = \{a, b, c, d\}$
- $P = "abacab"$

<table>
<thead>
<tr>
<th>$c$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>3</td>
<td>-1</td>
<td></td>
</tr>
</tbody>
</table>

$L(c)$
Example:

- $\Sigma = \{a, b, c, d\}$
- $P = \text{“abacab”}$

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>c</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L(c)$</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram of abacab](image.png)
Example:

\[ \Sigma = \{a, b, c, d\} \]

\[ P = "abacab" \]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(c)</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>-1</td>
</tr>
</tbody>
</table>

Diagram of the example:
Example:

$a, b, c, d$

$P = \text{"abacab"}$
The worst case time complexity of Boyer-Moore is $O(n \cdot m + s)$

- Example of the worst-case: $T = \text{“aaa...a”}$ and $P = \text{“baaa“}$

The worst-case:
- may occur in images or DNA sequences
- is unlikely in English text

Boyer-Moore’s algorithm is
- significantly faster than brute-force for English text
Knuth-Morris-Pratt (KMP) Algorithm

Knuth-Morris-Pratt’s algorithm:

- compares the pattern left-to-right
- shifts pattern more intelligently than brute-force algorithm

When a mismatch occurs: what is the most we can shift the pattern to avoid redundant comparisons?

- Answer: largest prefix of \( P[0 \ldots i] \) that is suffix of \( P[1 \ldots i] \) (where \( i \) is the last compared position)

![Diagram of KMP algorithm]

No need to compare again. Resume comparing here.
KMP Failure (Shift) Function

The Knuth-Morris-Pratt’s failure function $F(i)$:

- the algorithm preprocesses the pattern to compute $F(i)$
- $F(i)$ is the largest $j$ such that $P[0 \ldots j]$ is suffix of $P[1 \ldots i]$

Example: $P =$ ”abaaba“

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P[i]$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>$F(i)$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

---

---
Knuth-Morris-Pratt Algorithm

KMPMatch(\(T, P\)):

\[ F = \text{failureFunction}(P) \]

\( pos = 0 \)

\( i = 0 \)

\[ \text{while } pos \leq n - m \text{ do} \]

\[ \text{if } P[i] \neq T[pos + i] \text{ then} \]

\[ \text{if } i > 0 \text{ then} \]

\[ pos = pos + i - F(i - 1) \]

\[ \text{else} \]

\[ pos = pos + 1 \]

\[ i = F(i - 1) \]

\[ \text{else} \]

\[ i = i + 1 \]

\[ \text{if } i == m \text{ then return } pos \quad \text{(match at position } pos) \]

\[ \text{done} \]

\[ \text{return } -1 \quad \text{(no match)} \]
Example

\[
\begin{array}{cccccc}
\text{a} & \text{b} & \text{a} & \text{a} & \text{b} & \text{c} \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{a} & \text{b} & \text{a} & \text{a} & \text{b} & \text{a} \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{a} & \text{b} & \text{a} & \text{a} & \text{b} & \text{a} \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{a} & \text{b} & \text{a} & \text{a} & \text{b} & \text{a} \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{a} & \text{b} & \text{a} & \text{a} & \text{b} & \text{a} \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{a} & \text{b} & \text{a} & \text{a} & \text{b} & \text{a} \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{a} & \text{b} & \text{a} & \text{a} & \text{b} & \text{a} \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{a} & \text{b} & \text{a} & \text{a} & \text{b} & \text{a} \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{a} & \text{b} & \text{a} & \text{a} & \text{b} & \text{a} \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{a} & \text{b} & \text{a} & \text{a} & \text{b} & \text{a} \\
\end{array}
\]

\[
\begin{array}{cccc}
i & 0 & 1 & 2 & 3 & 4 & 5 \\
F(i) & 0 & 0 & 1 & 1 & 2 & 3 \\
\end{array}
\]
Knuth-Morris-Pratt: Analysis

Knuth-Morris-Pratt’s algorithm runs in $O(n + m)$ time:

- Failure function can be computed in $O(m)$.
- In each iteration of the while-loop:
  - either $i$ increases by 1, or
  - $pos$ increases by at least the amount $i$ decreases.
- Thus the while-loop is executed at most $2n$ times.
Binary Character Encoding

Given: alphabet $A_1, A_2, \ldots, A_n$.

- find binary codes $c(A_i)$ of the letters $A_i$

Encode letters as binary numbers of bit length $\lceil \log_2 n \rceil$:

- e.g. alphabet $A, B, C$
- $c(A) = 00, c(B) = 01, c(C) = 11$
- $ACBA = 00110100$

More efficient encodings:

- $c(A) = 0, c(B) = 10, c(C) = 11$
- $ACBA = 011100$

The coding should be unambiguous. Bad example:

- Let $c(A) = 10$, for $c(B) = 01$, and for $c(C) = 0$.
- Then both BC and CA yield the 010.

The code must be **prefix-free**: no $c(A_i)$ is prefix of a $c(A_j)$!
Huffman code

Given:

- Alphabet $A_1, A_2, \ldots, A_n$.
- Probabilities $0 \leq p(A_i) \leq 1$ for every letter $A_i$.

Problem:

- Find binary codes $c(A_i)$ of the letters $A_i$.
- Expected length for the codes, that is,

$$
\sum_{1 \leq i \leq n} p(A_i) \cdot |c(A_i)|
$$

should be minimal.
Huffman Algorithm:

- Create for every letter $A_i$ a tree consisting only of $A_i$.
- Search the two trees with the lowest sum of probabilities:
  - merge these two trees with a new node at the root
- Repeat the last step until only one tree is left.

Example:

\[
p(A) = .15, \ p(B) = .2, \ p(C) = .15, \ p(D) = .4, \ p(E) = .1
\]

<table>
<thead>
<tr>
<th>Letter</th>
<th>Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>101</td>
</tr>
<tr>
<td>C</td>
<td>01</td>
<td>110</td>
</tr>
<tr>
<td>D</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>11</td>
<td>111</td>
</tr>
</tbody>
</table>

average code length:

\[
3 \cdot 0.6 + 1 \cdot 0.4 = 2.2
\]