1 Complexity, Analysis of Algorithms

1. (a) Alternative 1: asymptotic analysis.

\[ \frac{(n + 1)^2}{n^3} = \frac{n^2}{n^3} + \frac{2n}{n^3} + \frac{1}{n^3} \to 0 \text{ for } n \to \infty \]

thus \((n + 1)^2 \in O(n^3)\) since \(0 < \infty\)

Alternative 2: analysis based on the definition.

Can we find \(n_0\) and \(c > 0\) such that \((n + 1)^2 \leq c \cdot n^3\) for all \(n \geq n_0\)?

\[
\begin{array}{ccc}
  n & (n + 1)^2 & n^3 \\
  1 & 4 & 1 \\
  2 & 9 & 8 \\
  3 & 16 & 27 \\
\end{array}
\]

Choose \(c = 1\) and \(n_0 = 3\). Then a cubic polynomial grows faster than a quadratic one.

(b) Alternative 1:

\[ \frac{(n - 1)^3}{n^2} \to \infty \text{ for } n \to \infty \]

Thus \((n - 1)^3 \notin O(n^2)\) since \(\infty \not< \infty\).

Alternative 2:

As a cubic polynomial grows faster than a quadratic we cannot find any \(c > 0\) and \(n_0\) such that

\[(n - 1)^3 \leq c \cdot n^2 \text{ for all } n \geq n_0\]

(c) Alternative 1:

\[ \frac{2^{n+1}}{2^n} = 2 \to 2 \text{ for } n \to \infty \]

true: \((2 < \infty)\)
Alternative 2:
Choose \( c = 2 \) and \( n_0 = 1 \):
\( 2^{n+1} \leq 2 \cdot 2^n \) for every \( n \geq 1 \)

(d) Alternative 1:
\[
\frac{3^{n-1}}{2^n} = \frac{1}{3} \left( \frac{3}{2} \right)^n \rightarrow \infty \text{ for } n \rightarrow \infty
\]
false: \( (\infty \not< \infty) \)

Alternative 2:
Assume the statement is true. Then there exist \( c \) and \( n_0 \) such that for every \( n > n_0 \):
\[
3^{n-1} \leq c \cdot 2^n
\]
Hence
\[
3^n \leq 3c \cdot 2^n
\]
and
\[
\left( \frac{3}{2} \right)^n \leq 3c
\]
for every \( n > n_0 \), which cannot be true since \( c \) is a constant. Thus the statement is false.

(e) Recall the definition: \( f(n) \in O(g(n)) \) iff there exist constants \( c > 0 \) and \( n_0 \geq 0 \) such that \( f(n) \leq cg(n) \) for every \( n \geq n_0 \). Take \( c = 2 \) and an arbitrary \( n_0 \). Then \( 2 \cdot |\sin(n)| \leq c \cdot 1 \) (for every \( n \)). The asymptotic method cannot be used in that case since \( \sin \) is a periodic function.

(f) Alternative 1:
\[
\frac{(n + 1)^2}{n^3} \rightarrow 0 \text{ for } n \rightarrow \infty
\]
false: \( (0 \neq 0) \)

Alternative 2:
Assume that the statement is true. Then there exist constants \( c > 0 \) and \( n_0 \geq 0 \) such that for every \( n > n_0 \):
\[
(n + 1)^2 \geq c \cdot n^3
\]
Hence for every \( n > n_0 \):
\[
\frac{(n + 1)^2}{n^3} \geq c
\]
which cannot be true since \( c \) is a constant. Thus the statement is false.

(g) Alternative 1:
\[
\frac{(n - 1)^3}{n^2} \rightarrow \infty \text{ for } n \rightarrow \infty
\]
true: \( (\infty > 0) \)

Alternative 2:
Take \( c = 1 \) and \( n_0 = 4 \):
\( (n - 1)^3 \geq n^2 \) for every \( n \geq 3 \)
(h) Alternative 1:
\[
\frac{2^{n+1}}{2^n} = 2 \to 2 \text{ for } n \to \infty
\]
true: \((2 > 0)\)

Alternative 2:
Take \(c = 1\) and \(n_0 = 0\):
\(2^{n+1} \geq 2^n\) for every \(n \geq 0\)

(i) Alternative 1:
\[
\frac{3^{n-1}}{2^n} = \frac{1}{3} \left(\frac{3}{2}\right)^n \to \infty \text{ for } n \to \infty
\]
true: \((\infty > 0)\)

Alternative 2:
Take \(c = 1\) and \(n_0 = 3\):
\(3^{n-1} \geq 2^n\) for every \(n \geq 3\)

(j) Alternative 1:
\[
\frac{(n+1)^2}{n^3} \to 0 \text{ for } n \to \infty
\]
false: \((0 \neq 0 < \infty)\)

Alternative 2:
Notice that \(f(n) \in \Theta(g(n))\) iff \(f(n) \in O(g(n))\) and \(f(n) \in \Omega(g(n))\). We proved (see (f)) that \((n+1)^2 \notin \Omega(n^3)\) thus the statement is false.

(k) Alternative 1:
\[
\frac{(n-1)^3}{n^2} \to \infty \text{ for } n \to \infty
\]
false: \((0 < \infty \neq \infty)\)

Alternative 2:
false: follows from (b)

(l) Alternative 1:
\[
\frac{2^{n+1}}{2^n} = 2 \to 2 \text{ for } n \to \infty
\]
true: \((0 < 2 < \infty)\)

Alternative 2:
true: follows by (c) and (h)

(m) Alternative 1:
\[
\frac{3^{n-1}}{2^n} = \frac{1}{3} \left(\frac{3}{2}\right)^n \to \infty \text{ for } n \to \infty
\]
false: \((0 < \infty \neq \infty)\)
**Alternative 2:**

false: follows by (d)

2. Lemma: \( f \in O(g) \Rightarrow O(f) \subseteq O(g) \)

We conclude:

(a) \( k^{n+1} = k \cdot k^n \Rightarrow O(k^{n+1}) = O(k^n) \)

(b) \( n^k \in O(n^{k+1}) \), but not vice versa (i.e. \( n^{k+1} \not\in O(n^k) \))

(c) \( n^{k+1} \in O(k^n) \), but not vice versa

(d) \( \log(k^{\log n}) = \log n \cdot \log k = \log(n^{\log_k k}) \Leftrightarrow \) (since \( \log \) is a monotone function)

\[
k^{\log_k n} = n^{\log_k k}
\]

(e) as \( \log k < k \) we have:
\( n^{\log_k k} \in O(n^k) \), but not vice versa

(f) \( \log k \geq 2 \) since \( k \geq 4 \)

\[
\frac{n^{3/2}}{n^{\log_k k}} = n^{3/2-\log_k k} \to 0 \text{ for } n \to \infty
\]

\( n^{3/2} \in O(n^{\log_k k}) \), but not vice versa

(g)

\[
\frac{n \log n}{n^{3/2}} = \frac{\log n}{n^{1/2}} \to 0 \text{ for } n \to \infty
\]

\( n \log n \in O(n^{3/2}) \), but not vice versa

By the relations proved above and by the lemma we obtain:
\[
O(n \log n) \subset O(n^{3/2}) \subset O(n^{\log_k k}) = O(k^{\log n}) \subset O(n^k) \subset O(n^{k+1}) \subset O(k^n) = O(k^{n+1})
\]
3. (a) Show first that \( f(n) + g(n) \in O(\max(f(n), g(n))) \):

For any \( n, f, g \) we have

\[
f(n) + g(n) \leq 2 \cdot \max(f(n), g(n))
\]

take \( c = 2 \) and \( n_0 = 1 \) to see that \( f(n) + g(n) \in O(\max(f(n), g(n))) \).

Prove then \( \max(f(n), g(n)) \in O(f(n) + g(n)) \).

As both \( f(n) \) and \( g(n) \) are positive we have:

\[
f(n) \leq f(n) + g(n) \text{ och } g(n) \leq f(n) + g(n) \text{ thus}
\]

\[
\max(f(n), g(n)) \geq f(n) + g(n)
\]

Take e.g. \( c = n_0 = 1 \) to conclude that

\( \max(f(n), g(n)) \in O(f(n) + g(n)) \).

Now by lemma used in Problem 2 we get \( O(\max(f(n), g(n))) = O(f(n) + g(n)) \).

(b) As for every \( h \) we have \( h \in O(h) \) we obtain the result by putting \( h(n) = f(n)g(n) \).

4. (a)

\[
\frac{\log_2 3n}{\log_2 2n} = \frac{\log_2 3n}{\log_2 2n} \cdot \log_2 3 = \frac{\log_2 3 + \log_2 n}{\log_2 2 + \log_2 n} \cdot \log_2 3 \to 1 \cdot \log_2 3 \text{ for } n \to \infty
\]

Thus in this case the relation holds: \( 0 < \log_2 3 < \infty \).

(b) \( n \) is a natural number \( \Rightarrow \sin 2\pi n = 0 \)

\( O(n \sin 2\pi n + 2) = O(2) \subset O(n + 2) \)

5. Take \( f(n) = n^2 \) and \( g(n) = n \). Clearly \( \log f(n) = 2 \log n \in O(\log g(n)) = O(n) \), but \( n^2 \notin O(n) \). Thus the statement is false.

6. As \( f \in O(g) \), then there exist constants \( c_f > 0 \) and \( n_{0_f} \geq 0 \) such that for every \( n \geq n_{0_f} \) we have \( f(n) \leq c_f g(n) \). Similarly, we have constants \( c_g \) and \( n_{0_g} \) such that for every \( n \geq n_{0_g} \) we have \( g(n) \leq c_g h(n) \). Take \( n_0 = \max(n_{0_f}, n_{0_g}) \) and \( n \geq n_0 \).

Then \( f(n) \leq c_f g(n) \leq c_f c_g h(n) \). Hence \( f \in O(h) \) and the respective constants are

\( c = c_f c_g \) and \( n_0 \) as defined above.

7. Assume \( f \in \Theta(g) \).

Hence \( f \in O(g) \) and \( f \in \Omega(g) \).

Thus \( g \in \Omega(f) \) and \( g \in O(f) \).

It follows that \( g \in \Theta(f) \).

8. • The measurement was done on data of too small size \( < n_0 \).

  • The data could have been chosen in such a way that they were close to the best case data for A and close to the worst case data for B.

  • The big-O notation gives only the upper bound for time complexity. If A is in \( O(n^2) \) it may also be in e.g. \( O(n) \).

9. (a) Yes, follows by (b) below.
(b) Yes. We know that that the worst case complexity of the algorithm is at least \( \Theta(n) \). So it may as well be worse, such as \( \Theta(n^2) \). We have no information on the execution time in the best case, but for some algorithms the best case complexity may coincide with the worst case complexity (take for example a list traversal algorithm: for any list of length \( n \) the time needed is \( \Theta(n) \) and there is no distinction between worst-case and best-case data.

(c) Yes. For some data the algorithm may have higher complexity than for the best case data.

(d) No. For the best case data the execution time is characterized by a function
\[
f(n) \in \Theta(n) \text{ and } f(n) \notin \Omega(n^2)
\]

10. (a) Number the for-loops from 1 to 3.

Simple analysis of each loop: 1: \( \Theta(n) \), 2: \( O(n) \), 3: \( O(n) \), thus altogether:
\[
\Theta(n) \cdot O(n) \cdot O(n) \cdot O(1) = O(n^3)
\]

More precise analysis: Change first \( O(1) \) to a constant \( c \).

\[
\begin{align*}
\text{loop 3:} & \sum_{k=1}^n c = cj \\
\text{loop 2:} & \sum_{j=i+1}^n cj = c(n - (i + 1) + 1)\frac{i + i + n}{2} = \frac{c}{2}(n - i)(1 + i + n) \\
\text{loop 1:} & \sum_{i=1}^{n-1} \frac{c}{2}(n - i)(1 + i + n) = \\
& \text{[by a complex computation]} = \\
& \frac{c}{6}n(n - 1)(n + 1) \in \Theta(n^3)
\end{align*}
\]

(b) Simple analysis: Both internal loops take time \( \Theta(n) \). The external loop is executed \( \Theta(n) \) times, the body of the if-instruction is executed every second time out of these. This gives:
\[
\Theta(n) \cdot \frac{1}{2} \Theta(n) = \Theta(n^2)
\]

(c) \( T(n) = \sum_{i=1}^n \sum_{j=3}^{n-1} 5 \cdot c \log n = 5c \sum_{i=1}^n (n - 4) \log n = 5cn(n - 4) \log n = \\
= 5cn^2 \log n - 20cn \log n \in \Theta(n^2 \log n) \)

(d) The outer loop will be repeated \( n \) times. The inner loop will be repeated: once for \( i = 1 \), twice for \( i = 2 \), ..., \( 1 \cdot 2 \cdot 3 \ldots k \) thus \(( k!) \) times for \( i = k \). This gives time complexity \( \Theta(n!) \).
11. Time complexity analysis of $foo$:

\[
\sum_{i=1}^{4} \Theta(1) = 4\Theta(1) = \Theta(1)
\]

- $O(0)$ includes only the functions which $\to 0$ for $n \to \infty$
- $O(7) = O(1) \supseteq \Theta(1)$
- $O(n) \supseteq \Theta(1)$
- $\Omega(7) = \Omega(1) \supseteq \Theta(1)$

Thus: $T(n) \in O(7)$, $T(n) \in O(n)$, $T(n) \in \Omega(7)$

12. This is a variant of binary search algorithm. Let $k$ by the smallest natural number such that $n \leq 2^k$. Then in the worst case the inner loop will be executed $k + 1$ times, hence the worst case time complexity is $\Theta(\log n)$

13. $T(0) = 1$

$T(n + 1) = T(n) + 1$

Hence $T(n) = n + 1 \in \Theta(n)$.

14. The first for-loop initialises the array so that all elements are 0. This takes $O(n)$ time.

In the next step we analyse the contents of loop1. The first for-loop prints the contents of array $A$. This takes $O(n)$ time. The tricky part is loop2, however this is the interesting part which determines the overall complexity of the algorithm. So it is important to understand exactly what this code does. It first increases the first position of $A$ with 1. Hence after the first iteration of loop1, $A$ contains a 1 in position 1 and 0 in the rest of the positions. In the next iteration position 1 is increased again and the value becomes 2. In this situation the value is reset to 0 and the next position is increased by 1. The next position is again checked whether the value equals to 2. If so, the value is reset and the next position after that is examined. In the worst case loop2 must examine all positions in the array, so the time complexity of loop2 is $O(n)$. Notice that this loop uses $A$ as a binary counter with the first position being the least significant: at each entrance the counter is increased by one. The content of the counter is not changed in loop1, which simply prints it and reenters loop2 with $i = 1$. The big question to answer now is when $i = n$, i.e. the condition when loop1 exits. This happens when, after being increased, all positions of $A$ except the last one have value 1. Since this algorithm counts, in binary numbers, from 0 to the number with $n$ 1’s, loop1 must iterate this number of times, i.e. $O(2^n)$.

Overall time complexity: $T(n) \in O(n) + O(2^n) \cdot (O(n) + O(n)) = O(n2^n)$

Analysing the space complexity is simple. The array occupies memory of size $\Theta(n)$, hence this is the total space complexity.
2. Stacks and Queues, Hashing, Trees

1. We use the following notation for operations:
   
   - **send** the first input character is removed from the input and sent directly to the output.
   
   - **push** the first input character is removed from the input and pushed on the stack.
   
   - **pop** the stack is popped and the obtained character is sent to the output.
   
   - **enqueue** the first input character is removed from the input and placed in the queue.
   
   - **dequeue** a character is removed from the queue and sent to the output.

(a) The permutation X U W V Y can be produced by the following operations:

   send, push, send, push, push, pop, pop, pop

   It cannot be produced by using queue. After sending X directly to the output we have to read Y. To obtain the required sequence it cannot be sent to the output. We can only enqueue it. Then U has to be sent to the output, since it has to precede Y in the required sentence. The next character is V. We cannot send it to the output but if we enqueue it we still cannot obtain the required sequence since on the output it will be preceded by already enqueued Y.

(b) The permutation Y U W X V cannot be obtained by using stack. The longest possible sequence of operations not violating the requirement is: push, send, send. The next input character is V. We cannot send it to output. If we push it on the stack, then in the resulting sequence it will precede X which is presently on the stack.

This permutation can be obtained by using queue:

   enqueue, send, send, enqueue, send, dequeue, dequeue

2. (a) Denote the stacks S1 and S2. The items pushed on S1 are stored in the cells T[0], T[1], ..., T[k1] where k1 is the index of the top when S1 contains k1 + 1 items. Similarly the items pushed on S2 are stored in T[n], T[n−1], ..., T[n−k2] where n−k2 is the index of the top when S2 contains k2 + 1 items. Overflow error is reported by an attempted push operation if k1 + k2 = n + 1.

(b)

\[
\begin{align*}
\text{a** } & \text{ Front } = 0, \text{ Length } = 1 \\
\text{ab* } & \text{ Front } = 0, \text{ Length } = 2 \\
\text{abc } & \text{ Front } = 0, \text{ Length } = 3 \\
\text{abc } & \text{ Front } = 1, \text{ Length } = 2 \\
\text{abc } & \text{ Front } = 2, \text{ Length } = 1 \\
\text{dbc } & \text{ Front } = 2, \text{ Length } = 2.
\end{align*}
\]
3. In the following we assume that \( \text{input} \) reads and returns one character from the input

\[
\textbf{function} \ \text{palindrome}(n): \ \textbf{boolean} \\
\text{makeEmptyStack}(S); \\
\text{for} \ i = 1 \ \text{to} \ \lfloor n/2 \rfloor \ \text{do} \\
\quad \text{Push}(S, \text{input}) \\
\quad \text{if} \ 2 \cdot \lfloor n/2 \rfloor \neq n \ \text{then} \ \text{input} \\
\text{for} \ i = 1 \ \text{to} \ \lfloor n/2 \rfloor \ \text{do} \\
\quad \text{if} \ \text{Pop}(S) \neq \text{input} \ \text{then return false} \\
\text{return true}
\]

The worst case running time is \( O(n) \) (justification omitted).

4. (a) \bullet \textit{MakeEmptyQueue}: create two empty stacks: \textit{E} and \textit{D}.

\bullet \textit{To enqueue} data item \( d \) push it on \( E \).

\bullet \textit{To dequeue}:

\quad If \( D \) is empty and \( E \) is empty report error.

\quad Otherwise:

\quad i. If \( D \) is empty, pop one-by-one all elements of \( E \) and push them on \( D \). After this all enqueued data items are placed on \( D \) from top to bottom in order of enqueueing: the most recently enqueued one is in the bottom and the \textit{Front} element is placed on the top of \( D \). We pop it to complete the dequeue operation.

\quad ii. Otherwise pop from \( D \).

\bullet \textit{Front} operation is as \textit{dequeue} but the data item is not popped from \( D \).

\bullet \textit{IsEmptyQueue} is implemented as two checks: \textit{IsEmptyStack}(E) and \textit{IsEmptyStack}(D); it returns \textit{true} iff both checks return \textit{true}.

(b) Pseudocode not included

(c) We assume that each stack operation runs in \( \Theta(1) \).

\quad i. \textit{Enqueue} operation is always a push on \( E \), so it runs in constant time \( \Theta(1) \).

\quad ii. \bullet \text{A single } \textit{enqueue} \text{ operation is implemented as push on } E, \text{ thus works } \text{always} \text{ in } \Theta(1); \text{ there is no specific worst case.}

\quad \bullet \text{The worst case of } \textit{dequeue} \text{ is if it is requested after pushing } n \text{ data items on } E. \text{ In that case we need:}

\quad \hspace{1cm} (1) \ \text{emptiness check of both stacks done in } \Theta(1)

\quad \hspace{1cm} (2) \ n \ \text{pop operations and } n \ \text{emptiness checks on } E \ \text{done in } \Theta(n)

\quad \hspace{1cm} (3) \ n \ \text{push operations on } D \ \text{done in } \Theta(n)

\quad \hspace{1cm} (4) \ \text{one pop operation on } D \ \text{done in } \Theta(1)

\text{Thus the worst case dequeue takes } \Theta(n) \text{ time.}
(d) Each data item of the sequence is only once pushed on E, popped from E, pushed on D and popped on D. Thus the amortized cost to enqueue and to dequeue all n data items is $\Theta(n)$, and the amortized cost per item is $\Theta(1)$.

5. The solution is a simple modification of the ring buffer (see p. 76 of the textbook). Thus the elements placed in deque D are stored in a table $T[0..N-1]$. Two variables: F and L are used to store, respectively, the index of the front and the length of the actually stored sequence. Empty deque is created by setting F and L to 0 and allocating T. If D is nonempty its elements $d_0, \ldots, d_{L-1}$, are stored in the positions $T(F), T(F+1) \mod N, \ldots, T(F+L-1) \mod N$. The operations are executed as follows:

- **addEnd(E, D)**: If $L = N$ report error. Otherwise increase L by one and place E in $T((F + L - 1) \mod N)$.
- **addFront(E, D)**: If $L = N$ report error. Otherwise increase L by one and place E in $T(N-1)$ if $F = 0$ and in $T(F-1)$ if $F \neq 0$. Update F respectively.
- **deleteEnd(D)**: If $L = 0$ report error. Otherwise return $T((F+L-1) \mod N)$ and decrease L by one.
- **deleteFront(D)**: If $L = 0$ report error. Otherwise return $T(F)$, decrease L by one and increase F by one modulo N.

6. (a) Coalesced chaining: the pointers are represented as table indices.

<table>
<thead>
<tr>
<th></th>
<th>0 x</th>
<th>8a 0 x</th>
<th>8a 0 2</th>
<th>8a 0 2</th>
<th>8a 0 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>15c</td>
<td>0</td>
<td>15c</td>
<td>0</td>
<td>15c</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>14b</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>8a 0 2</th>
<th>8a 0 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>15c 1 0</td>
<td>15c 1 0</td>
</tr>
<tr>
<td>2</td>
<td>14b 0 5</td>
<td>14b 0 5</td>
</tr>
<tr>
<td>3</td>
<td>4e 0 x</td>
<td>4e 0 x</td>
</tr>
<tr>
<td>4</td>
<td>32d 0 3</td>
<td>32d 0 3</td>
</tr>
<tr>
<td>5</td>
<td>0 x</td>
<td>7f 0 x</td>
</tr>
<tr>
<td>6</td>
<td>0 x</td>
<td>0 x</td>
</tr>
</tbody>
</table>

(b) Solution not included.

(c) Double hashing.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>15c</th>
<th>0</th>
<th>14b</th>
<th>0</th>
<th>14b</th>
<th>0</th>
<th>14b</th>
<th>0</th>
<th>14b</th>
<th>0</th>
<th>14b</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>15c</td>
<td>0</td>
<td>15c</td>
<td>0</td>
<td>15c</td>
<td>1</td>
<td>15c</td>
<td>1</td>
<td>15c</td>
<td>1</td>
<td>15c</td>
<td>1</td>
<td>15c</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>8a 0</td>
<td>8a 0</td>
<td>8a 0</td>
<td>8a 0</td>
<td>8a 0</td>
<td>8a 0</td>
<td>8a 0</td>
<td>8a 0</td>
<td>8a 0</td>
<td>8a 0</td>
<td>8a 0</td>
<td>8a 0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>32d</td>
<td>0</td>
<td>32d</td>
<td>0</td>
<td>32d</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4e 0</td>
<td>4e 0</td>
<td>4e 0</td>
<td>4e 0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7f 0</td>
<td>7f 0</td>
</tr>
</tbody>
</table>

7. With the deleted bit marking technique the space of the deleted elements is not
re-used for insertion. The probing sequences become very long even if they include mostly the deleted elements, this increases the time of the search.

8. Two letters give less conflicts over the space of all car identifications and are thus better than the remaining variants, but require larger hash table.

9. The level order.

10. (a) preorder
    (b) \textbf{procedure} TraverseTree(n: treenode);
        \begin{verbatim}
        var Q: ADT Queue
        MakeEmptyQueue(Q)
        Enqueue(n)
        while not IsEmptyQueue(Q) do
            n \leftarrow Dequeue(Q)
            Print label of n
            \textbf{foreach} child c of n \textbf{in reverse order} \textbf{do}
                Enqueue(c)
        \end{verbatim}

    This is “reversed” level order traversal.

11. (a) full trees: a, c, f, g

    (b) complete trees: a, b, c, g

    (c) perfect trees: a, c

    The trees d and h do not belong to any of these categories.

12. The indexing of the array starts with 1, according to the variant in the course book p.296 and p.338, or with 0, according to the variant discussed at the lecture.

    \begin{center}
    \begin{tabular}{|c|c|c|c|c|c|}
    \hline
    0 & 1 & 2 & 3 & 4 & 5 \\
    \hline
    \end{tabular}
    \begin{tabular}{|c|c|c|c|c|c|c|}
    a & b & i & c & f & j & k \\
    \hline
    d & e & g & h \\
    \hline
    \end{tabular}
    \end{center}
3 Search Trees, Heap, Union/Find

1. (a) Refer to the definition of the BST in p. 418 of the course book.
   
   (b) In the following solution Deletion uses in-order predecessor:

   ```
   12
   / \  
  6   21
   / \  / \  
  5 10 18 25  
  /  
  7  
  \  
  9
   ```

   (c) No, the balance of the node with key 10 is 2.

2. We consider BST’s which do not allow multiple occurrences of a key.

   (a) Yes. Assume the contrary, i.e. assume that there exist two different BST trees $T_1$ and $T_2$ with the same keys which have identical trace in the preorder traversal. As $T_1$ and $T_2$ are different BST trees and have the same keys there exist nodes $n_1$ and $n_2$ such that $n_1$ is an ancestor of $n_2$ in $T_1$ and $n_2$ is an ancestor of $n_1$ in $T_2$. Thus, $n_1$ will precede $n_2$ in the preorder traversal of $T_1$ and $n_2$ will precede $n_1$ in the preorder traversal of $T_2$. This contradicts the assumption. Given the trace of the preorder traversal $k_1, \ldots, k_n$, the tree is reconstructed by iterating the insertion operation, starting from the empty BST: e.g. 5,4,12,7,8,15 results in

   ```
   5
   / \  
  4 12  
   / \  
  7 15  
  \  
  8
   ```

   Notice that there are sequences of keys which are not traces of the pre-order traversal of a BST, e.g. 5,12,4

   (b) Yes. A justification is similar to that given above. The tree can be reconstructed by insertion of the keys from right to left. For example 4,8,7,15,12,5 inserted in the reverse order 5,12,15,7,8,4 will reconstruct the tree of example a.

   (c) No. Both trees

   ```
   5 4
   / \  
  4 6 5  
  \  
  6
   ```
have the same trace of inorder traversal: 4,5,6.

(d) Yes. The maximal growing subsequences of a given sequence correspond to the levels of the tree. The elements of incomplete levels should be distributed according to the definition of BST.

For example 10, 5,15, 3, 12, 20 gives the tree

```
  10
 /   \
5     15
 /     /
3     12   20
```

Notice that there are sequences of keys which cannot be obtained as a result of the level-order traversal of a BST.

3. For every tree we compute the balance of each node. If for some node the balance is not in $[-1,1]$ then the tree is not AVL.

The trees with AVL topologies are thus: A, B, F

4. Try to find a counterexample. We can find it for $h = 2$:

```
\begin{tabular}{|c|c|c|c|c|}
\hline
h & max & n & min & n \\
\hline
0 & 1 & & 2 & \\
1 & 3 & & 4 & \\
2 & 7 & & 7 & \\
\hline
\end{tabular}
```
5. (a) We count and indicate the balance of each node: the trees T1 and T2 are
AVL trees since the balance of each node is 1,0 or -1. The remaining ones are
not AVL-trees.

(b) We only show result of Insert(41)

(c) i. First do Splay(4). As 4 is not in the tree the algorithm brings 3 to the
root by means of double rotation. Then 4 is inserted as the new root.

ii. **Advantages:** Splay tree needs no extra information in the nodes. It
is very easy to implement. It works particularly well if the same
key is accessed several times in row (since the accessed key is always
brought to the root).

**Disadvantage:** The worst case time of a single operation may be linear
on the number of nodes. The operations have *amortized* logaritmic
time complexity (i.e. the execution time is on average logarithmic for sequences of operations, but it is not logarithmic for every operation, like in some other dictionaries (which ones?)).

6. Solution not provided.

7. Both arrays can represent heap: see course book p.338, assume that indexing of the array starts with 1.

8. Yes, if the index of the root is 0, then the children of a node with index $i$ have indices $3i + 1, 3i + 2, 3i + 3$.

9. The original heap is to be transformed to a new one with reversed priorities (i.e. the largest number denotes the highest priority). Transform the heap by using a heapification technique similar to that used in Heap Sort algorithm. This can be done in time $\Theta(n)$, the space needed is $\Theta(1)$.

10. $Find(4)$ introduces no changes.
4 Sorting

1. We assume that the sequence to be sorted is stored in the memory as a table $T$ with indices 0 to 4.
   
   (a) $[7, 4, 12, 2, 5]$
   $[4, 7, 12, 2, 5]$
   $[2, 4, 7, 12, 5]$
   $[2, 4, 5, 7, 12]$
   
   (b) We use a heap with a reverse comparator, where the largest key is at the top, see 8.3.5. The table $T$ is indexed from 1, the root of the reverse heap is in $T'[1]$. The Heap Sort algorithm has two phases:
   
   - Heap initialization, where $T$ is transformed to heap:
     $[7, 4, 12, 2, 5]$
     $[12, 4, 7, 2, 5]$
     $[12, 5, 7, 2, 4]$
   
   - Sorting by extraction of the largest element(*) and re-heapification:
     $[4, 5, 7, 2, 12] (\ast)$
     $[7, 5, 4, 2, 12]$
     $[2, 5, 4, 7, 12] (\ast)$
     $[5, 2, 4, 7, 12]$
     $[4, 2, 5, 7, 12] (\ast)$
     $[2, 4, 5, 7, 12] (\ast)$
   
   (c) see 11.2.2
     $[7, 4, 12, 2, 5]$ pivot 5
     $[2, 4, 12, 7, 5]$ divided into
     $[2, 4][12, 2]$
     Recursive calls give:
     $[2, 4][2, 12]$

2. Only SelectionSort.

3. Create two initially empty lists F (females) and M (males). Traversing the array from left to right attach the scanned record at the end of the respective list F or M, depending on its content. This is done in $\Theta(n)$. As array was sorted wrt the names, so are the lists. Their elements can now be copied back to the array in time $\Theta(n)$.

4. (a) $[aba, dab, bbe, ccc, abd, cad]$
    $[dab, cad, aba, bbe, abd, ccc]$
    $[aba, abd, bbe, cad, ccc, dab]$

    (b) Bucket Sort would need more buckets, which may be mostly unused, but still take space.

    (c) Bucket Sort must be stable for Radix Sort to work. Hence Radix Sort is also stable.
5. The internal if-statement executes in time $\Theta(1)$. The internal for-loop is repeated $O(n)$ times and the external one $\Theta(n)$ times. The result is thus $O(1)O(n)O(n) = O(n^2)$.

A more precise analysis:

$$\sum_{i=0}^{n-2} \sum_{j=n-1}^{i+1} \Theta(1) = \sum_{i=0}^{n-2} (n-i-1)\Theta(1) = \frac{(n-1)(n-1+1)}{2}\Theta(1) = \frac{n^2-n}{2}\Theta(1) = \Theta(n^2)$$

The comparisons done by the algorithm are strict ($<$). The equal elements cannot change relative position, thus the algorithm is stable.

6. (a) The input list is sorted in the opposite direction (i.e. the elements appear in the decreasing order).

(b) For the partition of Quick Sort to be efficient it is necessary to be able to access both the next element and the previous element in data in constant time. The latter is not possible in a singly linked list.

(c) The analysis of the lower bounds for comparison-based sorting uses the decision tree (see p.395 in the course book). For the analysis of the lower bound in the worst case we consider the length of the longest path in the decision tree and we conclude that it cannot be made shorter than $\log n!$, which happens when the decision tree is well balanced.

The best case corresponds to the shortest path in the decision tree. Therefore we would like to have the decision trees as imbalanced as possible to get the shortest paths as short as possible. However, for every input sequence we have to decide which of possible permutations it is. For this every element of the sequence must be compared at least once. This gives the best case lower bound $\Omega(n)$, which is achieved by Insertion Sort.

7. (a) The advantage of starting with least significant fragments is that for every consecutive fragment we call Bucket Sort for sorting all table. It is not possible when starting from most significant fragments, as schematically illustrated by the following example:

BAC, ABA, BAA $\rightarrow$ ABA, BAC, BAA

Now sorting on the second letter cannot be done on whole array: we have to sort separately all elements starting with A and all elements starting with B.

(b) To be correct Radix Sort must use a stable Bucket Sort. To achieve this each bucket should be organized as a queue.

8. (a) Quick Sort: $\Theta(n^2)$

Radix Sort: $m\Theta(n) = \Theta(mn)$

When $n$ is much greater than $m$ the Radix Sort is better.

(b) Quick Sort: $\Theta(n \log n)$

Radix Sort: $m\Theta(n) = \Theta(mn)$

Radix Sort must always go through the maximal number of fragments, so that the result is not influenced by the decrease of the average length.
In the analysis of Quick Sort one should take into account that the time needed for comparison of strings depends on their length. This would make the analysis much more difficult, especially for the average case. To be more precise about the meaning of “much greater” one should do some experiments to get some idea about the constants involved. $\Theta$-expressions alone say nothing about that.

9. A: Insertion Sort naive: $O(n^2)$; but the internal loop is never executed more than $k$ times: $O(nk)$
   
   Selection Sort: $O(n^2)$
   
   B: Insertion Sort: $O(k^2) \frac{n}{k} = O(nk)$
   
   Selection Sort: $O(k^2) \frac{n}{k} = O(nk)$
   
   Thus Selection Sort goes slower in case A.

10. see solution to problem (8).

11. Explanations and examples not included.
   - Best case: a sorted array;
     Worst case: an array sorted in reverse order.
   - Best case: the pivot at each step is the median of the set.
     Worst case: the pivot at each step is a maximal element (or a minimal element) (see p.507 in Goodrich,Tamassia 4th edition)
     If the last element is always selected as the pivot the worst case is a sorted (or inverse sorted) sequence.
   - Best case: the first selected pivot for quickSelect$(S, k)$ is the $k$-th smallest element of $S$
   - Worst case: e.g. $k = 1$ and the pivot selected at each step is a maximal element of the argument set at this stage.

12. The explanations are not included
   - Yes
   - It is $O(n^2)$ thus also $O(n^3)$
   - No, it is $O(n^2)$
5 Graphs

1. In this case both.
2. (a) c a b f e d
   (b) c a f e b d
   (c) c a b d f e

3. 

The most important operations are InsertEdge and DeleteEdge together with the traversal operations First and Next. We summarize the complexity results in the table below. Both IsIndex and First execute in constant time, independently of the representation chosen. By a neighbour we mean in this case a vertex reachable by an outgoing edge.

<table>
<thead>
<tr>
<th>Operation</th>
<th>adj. matrix</th>
<th>adj. list</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert Edge</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Delete Edge</td>
<td>O(1)</td>
<td>O(1), O(k)</td>
</tr>
<tr>
<td>First</td>
<td>O(1), O(v)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Next</td>
<td>O(1), O(v)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Exist Edge</td>
<td>O(1)</td>
<td>O(1), O(k)</td>
</tr>
</tbody>
</table>

The instruction

$$\textbf{for \ each \ neighbour \ } u \ \textbf{of} \ v \ \textbf{do}$$
$$\textbf{foo}$$

can be implemented as:

$$i \leftarrow \text{First}(v)$$
$$\textbf{while IsIndex}(i) \ \textbf{do}$$
$$\textbf{foo}$$
$$i \leftarrow \text{Next}(i, v)$$

<table>
<thead>
<tr>
<th>Operation</th>
<th>No. of calls</th>
<th>adj. matrix</th>
<th>adj. list</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>1</td>
<td>O(1), O(v)</td>
<td>O(1)</td>
</tr>
<tr>
<td>IsIndex</td>
<td>O(1), O(k)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Vertex</td>
<td>O(0), O(k)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Next</td>
<td>O(1), O(k)</td>
<td>O(1), O(v)</td>
<td>O(1)</td>
</tr>
<tr>
<td>summa:</td>
<td>O(1), O(k)</td>
<td>O(1), O(k)</td>
<td>O(1), O(k)</td>
</tr>
</tbody>
</table>
For the adjacency matrix representation we can do a more precise analysis if we look together at all calls of First and Next. This requires only traversal of one row in the matrix, which includes $v$ vertices. We get thus complexity $\Theta(v)$, which is more precise than $\Omega(1), O(k^v)$.

4. (a) Modify depth-first search algorithm (p. 595).

Notice that the DFS algorithm called for $G$ and $v$:

- visits only the vertices in the connected component of $G$ including $v$;
- marks the edges as discovery edges and back edges; a graph is cyclic if there exists a back edge.

The modification of DFS algorithm includes thus:

- exploration of the whole graph $G$ (in contrast to DFS exploring only one connected component). This can be achieved by starting DFS for each unexplored vertex of $G$.
- reporting a cycle and stopping instead of marking an edge as a back edge.

(b) We assume that

- accessing a vertex and accessing a neighbour of a vertex takes time $O(1)$: this is true when graph is represented by adjacency list,
- Changing the encountered information of a vertex and checking it takes time $O(1)$

At initialization each vertex is visited once, $\Theta(n)$. DFS is called at most once on each vertex (since discovering of a cycle stops the exploration), and every edge is examined at most twice, when checking encountered neighbours for both end vertices of the edge. Hence the complexity is $O(n + e)$. Notice that an acyclic graph has at most $n-1$ edges which gives $O(n)$ complexity in that case.

(c) A tree is a connected acyclic graph. Thus, if the first call of DFS does not find a cycle, it suffices to check if it explored all vertices of the graph.

5. (a) This is a directed acyclic graph:

(b) We use Topological Sort. The result is shown below: the annotation 1..9 shows a possible ordering of the courses to be taken by Johan.
(c) The DAG describes a partial ordering of prerequisites: \( c_1 > c_2 \) if there exists a path from \( c_1 \) to \( c_2 \). The courses which are not ordered can be taken in parallel. But the courses on the same path must not be taken in the same term. Thus the number of nodes on the longest path determines the minimum number of terms needed to schedule all courses. The scheduling can be done by the breadth-first search.

In our example the maximal number of nodes on a path is four. Thus the schedule may look as follows:

Term1: DM, P1; Term 2: DS, TP; Term 3: FL, P2, AA; Term 4: CC, OT.

6. The idea is to label the vertices by different natural numbers showing their topological order. To achieve this extend the DFS algorithm with a counter initialized to the number of vertices in the graph. At the postvisit of each vertex we assign to the vertex the actual number of the counter, and the content of the counter is decreased. If the graph is not connected the DFS has to be successively called on each of the components. For the example see the solution of the next problem.

7. The following graph shows the restrictions:

This is a directed acyclic graph and the problem can be solved by Topological Sort. It is based on the modification of depth-first search discussed above. We assume here that the neighbours of a vertex are explored in DFS in the order reflecting their geometrical placement on the figure: the neighbours placed higher are selected first.

With the above assumption on ordering of neighbours the numbers are assigned in the following order:

- Moving in: 12
• Painting interior: 11
• Floor: 10
• Roof cover: 9
• Roof: 8
• Chimney : 7
• Insulation: 6
• Doors: 5
• Interior walls: 4
• Windows: 3
• Exterior walls: 2
• Ground: 1

The schedule is determined by undertaking the task in increasing order of their computed numbers.