Data Structures and Algorithms–Tutorial Guide

Fall 2024

Jelle Hellings

Department of Computing and Software
Faculty of Engineering, McMaster University
1280 Main Street West, Hamilton, ON L8S 4L7, Canada

Structure of this document

This document provides a *study and tutorial guide* for 'Data Structures and Algorithms' (COMPSCI 2C03), Section C01.

This guide provides a rough per-week outline of the timeline of this course. The timeline in this guide is *indicative* and are subject to change. For each week, this guide describes

Material The relevant course material (from the Textbook) that will be covered during the week. It is recommended to study the material *before* the lectures and tutorials to maximize your opportunities to ask the right questions.

Additional Material In support of the material from the textbook, if any (e.g., lecture notes).

Slides The relevant slide set covering the material of the week.

Tutorial Exercises The exercises from the textbook that the tutorial will focus on: these exercises are on the difficulty level of the assignments and, hence, provide excellent preparation for the assignments.

Problems Additional exercises that train mainly *algorithmic problem solving*. Only a select few of these exercises are covered in the tutorial. Detailed solutions are available via the solution guide.

We assume familiarity with the preliminaries (Lecture Notes, Chapter 2) throughout this course.

1 Week 1

1.1 Material

Textbook, Section 1.4.

1.2 Additional Material

Lecture Notes, Chapter 1. Chapter 3, Section 3.1–3.3.

1.3 Slides

Part 1.

1.4 Tutorial Exercises

Problem 1.1.

P1.1.1 Consider the following functions of n:

$$\begin{array}{ccccccc}
n^2 & \sum_{i=0}^{n} 5 \cdot i & n^3 \cdot \sqrt{\frac{1}{n^3}} & n^2 + 2^n & \left(\prod_{i=1}^{9} i\right) & \left(\sum_{i=0}^{\log_2(n)} 2^i\right) + 1 & 7^{\ln(n)} \\
-\ln\left(\frac{1}{n}\right) & \ln(2^n) & 10 & n\log_2(n^7) & \sqrt{n^4} & n^n & 5n
\end{array}$$

Group the above functions that have identical growth rate and order these groups on increasing growth. Hence,

(a) if you place functions $f_1(n)$ and $f_2(n)$ in the same group, then we must have $f_1(n) = \Theta(f_2(n))$;

(b) if you place function $f_1(n)$ in a group ordered before the group in which you place function $f_2(n)$, then we must have $f_1(n) = \mathcal{O}(f_2(n))$ but *not* $f_1(n) = \Omega(f_2(n))$.

Explain your answers.

Hint: You do not have to explain any of the well-known identities of Chapter 2 of the course notes or any of the comparisons of the order-of-growth of two functions that is already explained in Example 3.26 of the course notes (if you use results from Example 3.16, then do make sure that it is clear which result you are using and why you can use that result).

Solution.

$$\begin{aligned}
n^2 &= \Theta(n^2); \\
\sum_{i=0}^{n} 5 \cdot i &= 5\left(\sum_{i=0}^{n} i\right) = 5\frac{n(n+1)}{2} = \Theta(n^2) && \text{(Guass);} \\
n^3 \cdot \sqrt{\frac{1}{n^3}} &= n^3 \cdot \frac{1}{n^{\frac{3}{2}}} = n^{3-\frac{3}{2}} = \Theta(n^{\frac{3}{2}}); \\
n^2 + 2^n &= \Theta(2^n); \\
\left(\prod_{i=1}^{9} i\right) &= 9! = 362880 = \Theta(1); \\
\left(\sum_{i=0}^{\log_2(n)} 2^i\right) + 1 &= \left(2^{\log_2(n)+1} - 1\right) + 1 = 2 \cdot 2^{\log_2(n)} = 2n = \Theta(n) && \text{(Geometric series);} \\
7^{\ln(n)} &= \begin{cases} (e^{\ln(7)})^{\ln(n)} = \left(e^{\ln(n)}\right)^{\ln(7)} = n^{\ln(7)} = \Theta(n^{1.9459\ldots}), \text{ or} \\ 7^{\frac{\log_7(n)}{\log_7(e)}} = n^{\frac{1}{\log_7(e)}} = \Theta(n^{1.9459\ldots}); \end{cases} \\
-\ln\left(\frac{1}{n}\right) &= -\ln(n^{-1}) = -(-1)\ln(n) = \ln(n) = \Theta(\ln(n)); \\
\ln(2^n) &= n\ln(2) = \Theta(n); \\
10 &= \Theta(1); \\
n\log_2(n^7) &= n7\log_2(n) = \Theta(n\log_2(n)); \\
\sqrt{n^4} &= \left(n^4\right)^{\frac{1}{2}} = n^{\frac{4}{2}} = \Theta(n^2); \\
n^n &= \Theta(n^n); \\
5n &= \Theta(n).
\end{aligned}$$

The groups are ordered as follows:

$$\begin{aligned}
\Theta(1)&: \left(\prod_{i=1}^{9} i\right), 10;\\
\Theta(\ln(n))&: -\ln\left(\tfrac{1}{n}\right);\\
\Theta(n)&: \left(\sum_{i=0}^{\log_2(n)} 2^i\right), \ln(2^n), 5n;\\
\Theta(n\log_2(n))&: n\log_2(n^7);\\
\Theta(n^{\frac{3}{2}})&: n^3\cdot\sqrt{\tfrac{1}{n^3}};\\
\Theta(n^{1.9459\ldots})&: 7^{\ln(n)};\\
\Theta(n^2)&: n^2, \sum_{i=0}^{n} 5\cdot i, \sqrt{n^4};\\
\Theta(2^n)&: n^2+2^n;\\
\Theta(n^n)&: n^n.
\end{aligned}$$

P1.1.2 Consider the recurrence

$$T(n) = \begin{cases} 7 & \text{if } n \leq 1;\\ 3T\,(n-2) & \text{if } n > 1.\end{cases}$$

Use induction to prove that $T(n) = f(n)$ with $f(n) = 7\cdot 3^{\lfloor\frac{n}{2}\rfloor}$.

Solution. The base cases are $T(0)$ and $T(1)$. By the definition of T, we have

$$T(0) = 7, \qquad T(1) = 7$$

and, by the definition of f, we have

$$f(0) = 7\cdot 3^{\lfloor\frac{0}{2}\rfloor} = 7\cdot 3^0 = 7, \qquad f(1) = 7\cdot 3^{\lfloor\frac{1}{2}\rfloor} = 7\cdot 3^0 = 7.$$

Hence, $T(0) = f(0)$ and $T(1) = f(1)$.

As the induction hypothesis, we assume $T(n) = f(n)$ for all $0 \leq n < i$.

For the induction step, we prove $T(i) = f(i)$, $i > 1$. We have

$$T(i) = 3T(i-2) \qquad \text{(definition of } T(i)\text{)}$$

Note $i - 2 < i$. Hence, we can apply the induction hypothesis on $T(i-2)$:

$$T(i) = 3\cdot f(i-2) = 3\cdot 7\cdot 3^{\lfloor\frac{i-2}{2}\rfloor} \qquad \text{(definition of } f(i-2)\text{)}$$

We simplify the above as follows:

$$T(i) = 7\cdot 3^{\lfloor\frac{i-2}{2}+1\rfloor} = 7\cdot 3^{\lfloor\frac{i}{2}\rfloor}.$$

Hence, by the definition of $f(i)$, we conclude

$$T(i) = 7\cdot 3^{\lfloor\frac{i}{2}\rfloor} = f(i).$$

Problem 1.2. Consider the following COUNT algorithm.

```
Algorithm COUNT(L, v) :
Pre: L is an array, v a value.
$i, c := 0, 0.$
while $i \neq |L|$ do
  if $L[i] = v$ then
    $c := c + 1.$
  end if
```

```
    $i := i + 1$.
end while
return $c$.
```

Post: return the number of copies of v in L.

P1.2.1 What is the runtime complexity of Count? What is the memory complexity of Count?

Solution. Runtime complexity: $\Theta(|L|)$. Memory complexity: $\Theta(1)$.

2 Week 2

2.1 Material

Textbook, Section 1.4.

2.2 Additional Material

Lecture Notes, Chapter 3, Section 3.4–3.5.

2.3 Slides

Part 1.

2.4 Tutorial Exercises

Exercises 1.4.10, 1.4.16, 1.4.19, 1.4.20, 1.4.24, 1.4.26, 1.4.32.

Problem 2.1. Consider again the COUNT algorithm of Week 1.

P2.1.1 Provide an invariant for the while loop at Line 2 of the COUNT algorithm.

Solution. Invariant I: $0 \leq i \leq |L|$ and $c =$ "the number of copies of v in $L[0, i)$".

HINT: More formal versions are possible that imply the same, but we kept the invariant in line with what we expect as the post condition.

P2.1.2 Provide a bound function for the while loop at Line 2 of the COUNT algorithm.

Solution. Bound function $b : |L| - i$.

P2.1.3 Prove that the COUNT algorithm is correct. Use the invariant and bound function of your answers to P2.1.1 and P2.1.2.

Solution. First, we prove that the loop at Line 2 satisfies Invariant I.

In the base case (after Line 1), we have $i = 0$ and $c = 0$. Hence, Invariant I with $i = 0$ and $c = 0$ yields

$$0 \leq 0 \leq |L| \text{ and } 0 = \text{"the number of copies of v in } L[0, 0)\text{"},$$

which voidly holds as $L[0, 0)$ is an empty list.

As the induction hypothesis, we assume that Invariant I holds for the first i iterations of the while-loop at Line 2.

For the induction step, we prove that Invariant I holds after the $i + 1$-th iteration of the while-loop at Line 2.

Let I_{old} and I_{new} be versions of the variant in which i, c are replaced by the values $i_{\text{old}}, c_{\text{old}}$ (the values at the start of the iteration) and the values $i_{\text{new}}, c_{\text{new}}$ (the values at the end of the iteration), respectively.

Approach 1 We perform a case distinction on whether the condition $L[i] = v$ at Line 3 holds.

If $L[i] = v$, then Line 4 and 5 are executed.

We need to prove that $I_{\text{old}}, i_{\text{old}} \neq |L|, L[i_{\text{old}}] = v, i_{\text{new}} = i_{\text{old}} + 1, c_{\text{new}} = c_{\text{old}} + 1$ implies I_{new}.

A: By $I_{\text{old}}, i_{\text{old}} \neq |L|, i_{\text{new}} = i_{\text{old}} + 1$, we conclude $0 \leq i_{new} \leq |L|$.

B: By $I_{\text{old}}, L[i_{\text{old}}] = v$, we conclude $c_{\text{old}} + 1 =$ "the number of copies of v in $L[0, i_{\text{old}} + 1)$". Hence, by $i_{\text{new}} = i_{\text{old}} + 1, c_{\text{new}} = c_{\text{old}} + 1$, we conclude $c_{\text{new}} =$ "the number of copies of v in $L[0, i_{\text{new}})$".

By A and B, we conclude I_{new}.

Else, if $L[i] \neq v$, then only Line 5 is executed.

We need to prove that $I_{\text{old}}, i_{\text{old}} \neq |L|, L[i_{\text{old}}] \neq v, i_{\text{new}} = i_{\text{old}} + 1, c_{\text{new}} = c_{\text{old}} + 1$ implies I_{new}.

A: By $I_{\text{old}}, i_{\text{old}} \neq |L|, i_{\text{new}} = i_{\text{old}} + 1$, we conclude $0 \leq i_{new} \leq |L|$.

B: By $I_{\text{old}}, L[i_{\text{old}}] \neq v$, we conclude $c_{\text{old}} =$ "the number of copies of v in $L[0, i_{\text{old}} + 1)$". Hence, by $i_{\text{new}} = i_{\text{old}} + 1, c_{\text{new}} = c_{\text{old}}$, we conclude $c_{\text{new}} =$ "the number of copies of v in $L[0, i_{\text{new}})$".

By A and B, we conclude I_{new}.

Approach 2 A similar approach as above can be used to prove that after Line 5, the predicate

$$I_{\text{old}}, i_{\text{old}} \neq |L|, c_{\text{new}} = \text{"the number of copies of } v \text{ in } L[0, i_{\text{old}} + 1)\text{"}$$

holds. Using this predicate and the statement of Line 6, one can then use an approach similar as above to prove that after Line 6, I_{new} holds.

Having proven that the Invariant I holds, we next need to prove that the while-loop terminates. We do so by proving that b is indeed a bound function:

- b starts with the value $|L| - 0$, a natural number;
- b decreases every round of the while-loop due to Line 6; and
- if b reaches 0, then, due to the condition of the while-loop, the loop stops.

After Line 7, we know that the Invariant I holds and that $\neg(i \neq |L|)$. Hence, $i = |L|$ and we conclude $c =$ "the number of copies of v in $L[0, |L|)$" and, hence, $c =$ "the number of copies of v in L". As Line 8 returns c, we conclude that the algorithm is correct.

Problem 2.2. Consider the following FastPower algorithm:

```
Algorithm FastPower($x$, $y$) :
Pre: $y \in \mathbb{N}$, $X = x$, and $Y = y$.
1: $r, t, n := 1, x, y$.
2: while $n \neq 0$ do
3:     if $n$ is even then
4:         $t, n := t \cdot t, n/2$.
5:     else
6:         $r, n := r \cdot t, n - 1$.
7:     end if
8: end while
9: return $r$.
Post: returns $X^Y$.
```

P2.2.1 Why does the pre-condition introduce terms X and Y (that are not used in the program itself)?

Solution. Without using X and Y, the following program would be correct:

```
Algorithm BadPower($x$, $y$) :
Pre: $y \in \mathbb{N}$.
$r, x, y := 1, 1, 0$.
return $r$.
Post: returns $x^y$.
```

In this program, the post-condition is technically true as the program changes the values of x and y.

P2.2.2 Is the algorithm correct? If so, prove that the FastPower algorithm is correct. Otherwise, show how to fix the algorithm.

Solution. In the FastPower program, an obvious bottom-up step is available: the return statement at Line 9 returns the value r which, according to the post-condition, must be equivalent to X^Y. Hence, after the while-loop, we need to be able to prove that $r = X^Y$ holds using the invariant and, as the guard no longer holds, $n = 0$. As a first guess for the invariant, we can try to reformulate $r = X^Y$ (which must hold after the loop) in terms of the variables r, t, and n, as these are the only variables changed during the loop. The limited information availble without inspecting the rest of the program does not provide a good guess for such a reformulation.

Next, we perform a top-down step: before the while-loop, we only perform a single assignment statement $r, t, n := 1, x, y$ and we already know that the pre-condition holds before this assignment statement. Hence, after the assignment statement, we know $r = 1$, $t = x$, and $n = y$ (due to the assignment) and $y \in \mathbb{N}$, $X = x$, and $Y = y$. We note that $r \cdot t^n = X^Y$ holds before the while-loop. As $n = 0$ after the while-loop and $r = X^Y$ must hold after the while-loop, $r \cdot t^n = X^Y$ must also hold after the while loop. Hence, a reasonable guess for the invariant is $r \cdot t^n = X^Y$:

```
Algorithm FastPower($x$, $y$) :
Pre: $y \in \mathbb{N}$, $X = x$, and $Y = y$.
 1: $r, t, n := 1, x, y$.
    Known /* $P_{\text{while}}$ */: $r = 1$, $t = x$, $n = y$, $y \in \mathbb{N}$, $X = x$, and $Y = y$.
    Prove /* invariant */: $r \cdot t^n = X^Y$.
 2: while $n \neq 0$ do /* invariant: $r \cdot t^n = X^Y$, bound function $n$. */
      Known /* invariant ∧ guard */: $r \cdot t^n = X^Y$ and $n \neq 0$.
 3:   if $n$ is even then
 4:     $t, n := t \cdot t, n/2$.
 5:   else
 6:     $r, n := r \cdot t, n - 1$.
 7:   end if
      Prove /* invariant */: $r \cdot t^n = X^Y$.
 8: end while
    Known /* invariant ∧ ¬guard */: $r \cdot t^n = X^Y$ and $\neg(n \neq 0)$.
    Prove /* $Q_{\text{while}}$ */: $r = X^Y$.
 9: return $r$.
Post: returns $X^Y$.
```

Next, we prove the proof-obligations of the while-loop. first, we prove that /* P_{while} */ implies /* invariant */. In the invariant, we can fill in $r = 1$, $t = x$, $n = y$, $x = X$, and $y = Y$, which are all given by /* P_{while} */. Doing so yields $1 \cdot X^Y = X^Y$, which is obviosuly true.

Next, we prove that /* invariant ∧ ¬guard */ implies /* Q_{while} */. In the invariant, we can fill in $n = 0$ as provided by /* ¬guard */ to obtain $r \cdot t^0 = r = X^Y$. Hence, /* Q_{while} */ holds.

Finally, we prove that /* invariant */ holds *after* execution of the loop body, given that /* invariant ∧ guard */ holds *before* execution of the loop body. The if-statement at Line 3 performs a case distinction based on the value n. Next, we formalize the standard structure of a proof involving an if-statement.

First, we consider the if-case. We shall prove that /* Q_{if} */ holds after the assignment on Line 4. For the variables t and n affected by the assignment, we shall write t_{old} and n_{old} to denote their values before executing the assignment and t_{new} and n_{new} to denote their values after execution. Before the assignment, we have $r \cdot t^n = X^Y$, $n \neq 0$, and n is odd. Hence, after the assignment we have $r \cdot {t_{\text{old}}}^{n_{\text{old}}} = X^Y$, $n_{\text{old}} \neq 0$, and n_{old} is even. Furthermore, the assignment itself introduces the facts $t_{\text{new}} = t_{\text{old}} \cdot t_{\text{old}}$ and $n_{\text{new}} = n_{\text{old}}/2$. We have to prove that /* Q_{if} */ holds for the new values of t and n. Let

$$A : r \cdot {t_{\text{old}}}^{n_{\text{old}}} = X^Y,\ n_{\text{old}} \neq 0,\ n_{\text{old}} \text{ is even},\ t_{\text{new}} = t_{\text{old}} \cdot t_{\text{old}}, \text{ and } n_{\text{new}} = n_{\text{old}}/2,$$

be all facts that hold directly after the assignment. We must prove that A implies $r \cdot {t_{\text{new}}}^{n_{\text{new}}} = X^Y$. In $r \cdot {t_{\text{new}}}^{n_{\text{new}}} = X^Y$, we fill in the facts $t_{\text{new}} = t_{\text{old}} \cdot t_{\text{old}}$ and $n_{\text{new}} = n_{\text{old}}/2$ from A to obtain $r \cdot (t_{\text{old}} \cdot t_{\text{old}})^{n_{\text{old}}/2} = X^Y$. Hence, we must prove that A implies $r \cdot (t_{\text{old}} \cdot t_{\text{old}})^{n_{\text{old}}/2} = X^Y$. We apply ${t_{\text{old}}}^{n_{\text{old}}} = {t_{\text{old}}}^{2n_{\text{old}}/2} = (t_{\text{old}} \cdot t_{\text{old}})^{n_{\text{old}}/2}$ to the fact $r \cdot {t_{\text{old}}}^{n_{\text{old}}} = X^Y$ in A to conclude that A implies $r \cdot (t_{\text{old}} \cdot t_{\text{old}})^{n_{\text{old}}/2} = X^Y$, completing our proof.

The else-case is similar. We shall prove that /* Q_{if} */ holds after the assignment on Line 6. Before the assignment, we have $r \cdot t^n = X^Y$, $n \neq 0$, and n is not even. As n is not even, n must be odd. Hence, after the assignment we have $r_{\text{old}} \cdot t^{n_{\text{old}}} = X^Y$, $n_{\text{old}} \neq 0$, and n_{old} is odd. Furthermore, the assignment itself introduces the facts $r_{\text{new}} = r_{\text{old}} \cdot t$ and $n_{\text{new}} = n_{\text{old}} - 1$. Let

$$A : r_{\text{old}} \cdot t^{n_{\text{old}}} = X^Y,\ n_{\text{old}} \neq 0,\ n_{\text{old}} \text{ is odd, } r_{\text{new}} = r_{\text{old}} \cdot t, \text{ and } n_{\text{new}} = n_{\text{old}} - 1,$$

be all facts that hold directly after the assignment. We must prove that A implies $r_{\text{new}} \cdot t^{n_{\text{new}}} = X^Y$. In $r_{\text{new}} \cdot t^{n_{\text{new}}} = X^Y$, we fill in the facts $r_{\text{new}} = r_{\text{old}} \cdot t$ and $n_{\text{new}} = n_{\text{old}} - 1$ from A to obtain $r_{\text{old}} \cdot t \cdot t^{n_{\text{old}}-1} = X^Y$. Hence, we must prove that A implies $r_{\text{old}} \cdot t \cdot t^{n_{\text{old}}-1} = X^Y$. We apply $t^{n_{\text{old}}} = t \cdot t^{n_{\text{old}}-1}$ to the fact $r \cdot {t_{\text{old}}}^{n_{\text{old}}} = X^Y$ in A to conclude that A implies $r_{\text{old}} \cdot t \cdot t^{n_{\text{old}}-1} = X^Y$, completing our proof.

As `/* invariant */` holds after the if-statement of Line 3, the `/* invariant */` is maintained by the loop, which completes our proof-obligations with respect to the invariant of the while-loop of Line 2.

What remains is to prove that the while-loop terminates. Hence, we need to prove that n is a bound function. This is straightforward to prove. Initially, $n = Y$, which is a natural number due to the pre-condition. Next, if we assume that n is a natural number before the assignments in the if-statement at Line 3, then halving n (if n is a non-zero and even) or subtracting one (if n is non-zero and odd) will always yield a result in a new value of n that is a natural number strictly smaller than the original value n.

3 Week 3

3.1 Material

Textbook, Section 1.3.

3.2 Additional Material

Lecture Notes, Chapter 4.

3.3 Slides

Part 2 and 3.

3.4 Tutorial Exercises

Exercises 1.3.3, 1.3.4, 1.3.6, 1.3.13.

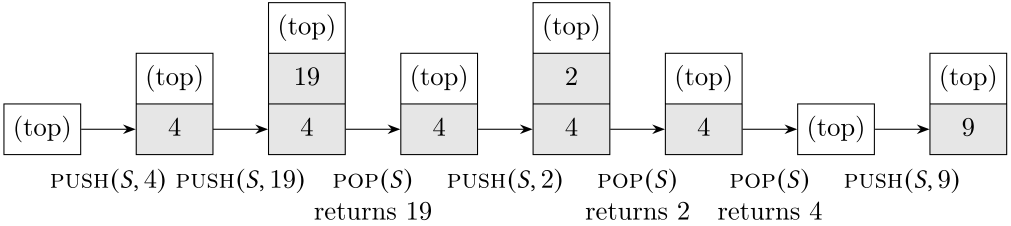
Problem 3.1.

P3.1.1 Consider an initially-empty stack S and the sequence of operations

$$\textsc{push}(S, 4),\ \textsc{push}(S, 19),\ \textsc{pop}(S),\ \textsc{push}(S, 2),\ \textsc{pop}(S),\ \textsc{pop}(S),\ \textsc{push}(S, 9).$$

Illustrate the result of each operation (clearly indicate the content of the stack after the operation and, in case of a $\textsc{pop}$, the value returned by the operation).

Solution.

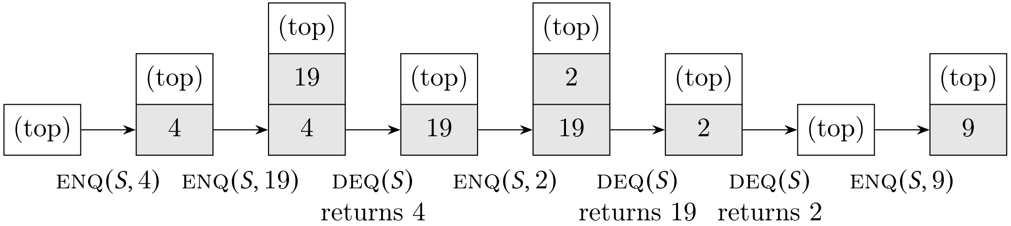


P3.1.2 Consider an initially-empty queue Q and the sequence of operations

$$\textsc{enqueue}(Q, 4),\ \textsc{enqueue}(Q, 19),\ \textsc{dequeue}(Q),\ \textsc{enqueue}(Q, 2),$$
$$\textsc{dequeue}(Q),\ \textsc{dequeue}(Q),\ \textsc{enqueue}(Q, 9).$$

Illustrate the result of each operation (clearly indicate the content of the stack after the operation and, in case of a $\textsc{dequeue}$, the value returned by the operation).

Solution. We use shorthands $\textsc{enq}$ for $\textsc{enqueue}$ and $\textsc{deq}$ for $\textsc{dequeue}$.



P3.1.3 Assume we have a stack implementation `MyDynArrayStack` using dynamic arrays: the implementation supports N $\textsc{push}$ operations with an amortized runtime complexity of $\Theta(1)$ per operation, and the implementation supports $\textsc{pop}$, $\textsc{Empty}$, and $\textsc{Size}$ operations with a runtime complexity of $\Theta(1)$.

Provide a *queue* implementation that uses `MyDynArrayStack` and supports any valid sequence of N Enqueue and Dequeue operations with an amortized runtime complexity of $\Theta(1)$ per operation. Explain why your implementation has the stated amortized runtime complexity for Enqueue and Dequeue.

Solution.

- We will use two stacks: stack T (top) and stack B (bottom).
- Stack T holds *recently-added values* with the most-recent value at the top of stack T.
- Stack B holds *old values* with the least-recent value at the top of stack B.
- To enqueue a value, we push it to the top of stack T.
- To dequeue a value, we pop it from stack B.
- If no values are in B, then we have to move values from T to B: if we pop all values from T, then we get the most-recent value first and the least-recent value last. Hence, if we pop all values from T and push them to B, then the least-recent value in T will end up at the top of B and the most recent value in T will end up at the bottom of B. Hence, this operation maintains the properties of T and B.
- Each value is pushed to a stack at-most twice (once to T, then once to B) and popped from a stack at-most twice (once from T, then once to B). These four operations are spread out over all Enqueue and Dequeue operations performed on the queue. Lastly, each of these push and pop operations has either a runtime cost or an amortized runtime cost of $\Theta(1)$. Hence, the implementation of our queue has the stated amortized runtime complexity for Enqueue and Dequeue.

Problem 3.2. Consider again the Count algorithm of Week 1.

P3.2.1 Provide an algorithm FastCount(L, v) that operates on *ordered lists* L and computes the same result as Count(L, v) with a running time of $\mathcal{O}(\log_2(|L|))$.

Solution.

```
Algorithm FastCount(L, v) :
Pre: L is an ordered array, v a value.
return  UpperBound($L, v, 0, |L|$) − LowerBound($L, v, 0, |L|$).
```

In the above, UpperBound is a variant of LowerBound that returns the first offset in L of a value $w > v$. Let next(v) be the smallest value larger than v, e.g., if all values under consideration are integers, then next(v) $= 1$. We can simply implement UpperBound(L, v) by LowerBound(L, next(v)).

The algorithm performs two binary searches over the entire list that each cost $\Theta(\log_2(|L|))$. Hence, the running time is $\Theta(\log_2(|L|))$.

Problem 3.3. The main difference between an array and a linked list is that arrays support *random access* efficiently: one can lookup the i-th value in an array in constant time. Many algorithms such as BinarySearch rely on random access.

P3.3.1 Provide an algorithm Get(L, i) that returns the i-th value in linked list L. What is the complexity of this algorithm?

Solution.

```
Algorithm Get(L, i) :
$node, pos := L.first, 0$.
while $pos < i$ do
    $node, pos := node.next, pos + 1$.
end while
```

```
return $node$.
```

The complexity of this operation is $\mathcal{O}(i)$.

P3.3.2 Consider the algorithm BinarySearch from the slides. What is the *average* complexity of this algorithm to search for a value $v \in L$ when list L is represented by a linked lists (using the above Get algorithm to implement $L[mid]$). Feel free to assume that the length of the list is an exact power of two if that simplifies your argument.

Solution. Assume $N = |L|$. Consider the values v_i and v_{N-i} at positions i and $N - i$. The pattern via which the binary searches for v_i and v_{N-i} traverse the list L mirror each other completely: if the x-th recursive call in the binary search for value v_i inspects a value at position p (Line 4), then the x-th recursive call in the binary search for value v_{N-i} inspects a value at position $N - p$. Hence, the average complexity of the x-th recursive call in the binary searches for values v_i and v_{N-i} is $\frac{p+N-p}{2} = \frac{N}{2}$. Hence, the average complexity to find either v_i or v_{N-i} is $\Theta(\frac{N}{2} \log_2(N)) = \Theta(N \log(N))$. We can perform the above analysis for all i, $0 \le i < N$, to conclude that the average complexity of this variant of BinarySearch is $\Theta(N \log_2(N))$.

4 Week 4

4.1 Material

Textbook, Chapter 2.1 and 2.2.

4.2 Additional Material

Lecture notes, Chapter 5.

4.3 Slides

Part 4 and 5.

4.4 Tutorial Exercises

Exercises 2.1.2, 2.1.3, 2.1.13, 2.1.15, 2.2.4, 2.2.7, 2.2.8.

Problem 4.1. Consider the following program

```
Algorithm SORT($L[0 \dots N)$) :
Pre: $L$ is an array.
while $L$ is not sorted do
    $L :=$ a random permutation of $L$.
end while
Post: $L$ is sorted.
```

Assume we can test that L is sorted in $\Theta(|L|)$ and that we can compute a random permutation of L in $\Theta(|L|)$.

P4.1.1 Does the Sort program sort correctly? If yes, then provide an invariant for the while-loop and provide a bound function that can be used to prove the correctness of the program. If no, then argue why the program is not correct.

Solution. If the program finishes the while-loop, then we have $\neg(L \text{ is not sorted})$. Hence, if the program finishes, then list L is sorted.

Unfortunately, the program is not guaranteed to finish: for most lists L, most permutations of L are *not* sorted and the program can indefinitely choose such non-sorted permutations.

P4.1.2 Assume the program Sort is correct. Is the program stable? Explain why.

Solution. No.

By taking any random permutation, we can end up changing the ordering between equal values. For example, if $L = [2_{\text{first copy}}, 1, 2_{\text{second copy}}]$ is the input, then $[1, 2_{\text{second copy}}, 2_{\text{first copy}}]$ would be a permutation that is sorted but is not stable.

P4.1.3 What is the worst case runtime complexity of this program? What is the best case runtime complexity of this program? Is this program optimal? Explain your arguments.

Solution. In the worst case, the program never finishes (see P1.1).

In the best case, L is sorted at input. Hence, in this case, we only check whether L is sorted in $\Theta(|L|)$.

P4.1.4 What is the expected case runtime complexity of this program? Explain your answer.

Hint: A *Bernoulli trial* is a random experiment with two possible outcomes: success and failure. If a Bernoulli trial T has a probability p of success, then the expected number of trials T needed to get one success is $\frac{1}{p}$. For example, if you want to throw a six-sided dice until you see the face-value 4, then the probability of success is $\frac{1}{6}$ and the expected number of throws is 6.

Solution. Given a list L, there are $|L|!$ permutations. If the list L has distinct values only, then only one of these lists is sorted. Hence, the probability of picking a sorted list is $p = \frac{1}{|L|!}$.

Based on the hint, we expect to try $\frac{1}{p} = |L|!$ permutations on average before finding a sorted permutation. For each of these tries, we check whether it is sorted and we have to compute the permutation: both cost $\Theta(|L|)$. Hence, the expected complexity is $\Theta(|L||L|!)$.

Problem 4.2. Consider the sorting algorithm WEIRDSORT:

```
Algorithm WEIRDSORT($L[start, end)$) :
 1: if $end - start = 2$ then
 2:   if $L[start] \geq L[start + 1]$ then
 3:     Exchange $L[start]$ and $L[start + 1]$.
 4:   end if
 5: else if $end - start > 2$ then
 6:   $k := (end - start)$ div 3.
 7:   WEIRDSORT($L$, $start$, $end - k$).
 8:   WEIRDSORT($L$, $start + k$, $end$).
 9:   WEIRDSORT($L$, $start$, $end - k$).
10: end if
```

Let $n = |L|$ be the length of the list being sorted by WEIRDSORT.

P4.2.1 Is WEIRDSORT a *stable* sort algorithm? If yes, explain why. If no, show why not and indicate whether the algorithm can be made stable.

Solution. The algorithm is not stable. Consider the list with two pairs $L = [(2, a), (2, b)]$ which we sort by comparing the first value of each pair (the numbers). We have $2 \geq 2$. Hence, the algorithm exchanges the two values on Line 3 resulting in the sorted list $L = [(2, b), (2, a)]$. To *fix* the algorithm, we change Line 2 to "**if** $L[start] > L[start + 1]$ **then**" (which prevents swapping of equal values).

P4.2.2 Prove via induction that WEIRDSORT will sort list L.

Solution. We will prove that WEIRDSORT, when applied on a list L with $0 \leq start \leq end \leq |L|$, will sort the portion of the list $L[start \dots end)$.

Base case The algorithm has *three* base cases.

- $end - start = 0$. A list with zero elements is always sorted.
- $end - start = 1$. A list with one element is always sorted.
- $end - start = 2$. The if-test and exchange at Lines 2 and 3 will assure that the largest of the two values in the list will come last. Hence, in this case WEIRDSORT sorts the list.

Induction Hypothesis WEIRDSORT sorts lists with less-than n values: WERIRDSORT sorts when applied on lists L with $0 \leq start \leq end \leq |L|$ such that $end - start < n$.

Induction step Consider WEIRDSORT with $2 < end - start = n$ values.

We will write $\mathcal{L}$ to refer to the list L *before* any sorting steps.

First, we note that $n \geq 3$ and $k = \lfloor \frac{n}{3} \rfloor$. Hence, $0 < k < n$. As such $start < start + k < end - k < end$ holds and $(end - k) - (start + k) = n - 2k \geq k$.

start	*start* + k	*end* - k	*end*
k values	$\geq k$ values	k values	

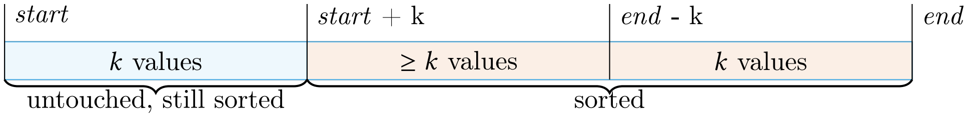
As $start < end - k < end$, we can use the induction hypothesis to conclude that the first recursive call at Line 7 sorts. We have:

start	*start* + k	*end* - k	*end*
k values	$\geq k$ values	k values	
sorted		untouched	

We will write $\mathcal{L}_1$ to refer to the permutation of list L obtained by the first sorting step.

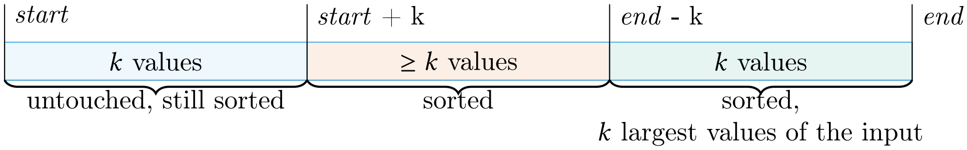
As $\mathcal{L}_1[\mathit{start} \ldots \mathit{end}-k)$ is now sorted, the k largest values in $\mathcal{L}[\mathit{start} \ldots \mathit{end}-k)$ are now guaranteed to be in $\mathcal{L}_1[\mathit{end}-2k \ldots \mathit{end}-k)$. Note that $\mathit{end}-2k \geq \mathit{start}+k$.

As $\mathit{start} < \mathit{start}+k < \mathit{end}$, we can use the induction hypothesis to conclude that the second recursive call at Line 8 sorts. We have:

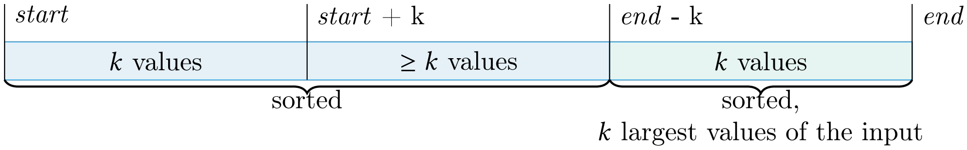


We will write $\mathcal{L}_2$ to refer to the permutation of list L obtained by the second sorting step.

As $\mathcal{L}_2[\mathit{start}{+}k \ldots \mathit{end})$ is now sorted, the k largest values in $\mathcal{L}_1[\mathit{start}{+}k \ldots \mathit{end})$ are now guaranteed to be in $\mathcal{L}_2[\mathit{end}-k \ldots \mathit{end})$. Each of these k largest values either came from $\mathcal{L}_1[\mathit{end}-k \ldots \mathit{end})$–which was unchanged and must hold the same values as $\mathcal{L}[\mathit{end}-k \ldots \mathit{end})$–or from $\mathcal{L}_1[\mathit{start}+k \ldots \mathit{end}-k])$. The previous recursive call guaranteed that $\mathcal{L}_1[\mathit{start}+k \ldots \mathit{end}-k])$ (which holds all of $\mathcal{L}_1[\mathit{end}-2k \ldots \mathit{end}-k))$ holds the k largest values of $\mathcal{L}[\mathit{start} \ldots \mathit{end}-k)$. Hence, $\mathcal{L}_2[\mathit{end}-k \ldots \mathit{end})$ now holds the k largest values of the original input $\mathcal{L}[\mathit{start} \ldots \mathit{end})$. Hence, we have



As $\mathit{start} < \mathit{end}-k < \mathit{end}$, we can use the induction hypothesis to conclude that the third recursive call at Line 9 sorts. We have:



We will write $\mathcal{L}_3$ to refer to the permutation of list L obtained by the third sorting step.

By induction, $\mathcal{L}_3[\mathit{start} \ldots \mathit{end}-k)$ is sorted. In addition, $\mathcal{L}_3[\mathit{end}-k \ldots \mathit{end})$ is untouched and, hence, is still sorted and holds the k largest values from the original input. Hence, $\mathcal{L}_3[\mathit{end}-k-1] \leq \mathcal{L}_3[\mathit{end}-k]$. We conclude that the entire list must be sorted.

P4.2.3 Give a recurrence $T(n)$ for the runtime complexity of WEIRDSORT.

Solution. The runtime complexity of WEIRDSORT is given by the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n \leq 2; \\ 3T(\lceil \frac{2n}{3} \rceil) + 1 & \text{if } n > 2. \end{cases}$$

We have three recursive calls that each operate on $n - k = \lceil \frac{2n}{3} \rceil$ values. Besides the recursive calls, WEIRDSORT does a constant amount of work.

P4.2.4 Solve the recurrence $T(n)$ using the *recursion tree method* by proving that $T(n) = \Theta(f(n))$ for some function $f(n)$. Feel free to make a suitable assumption about n (e.g., n is a power-of-two).

Solution. To simplify the construction of our recursion tree, we assume that n is always divisible by 3 (hence, each recursive call is exactly of size $n - k = \frac{2n}{3}$. Hence, we can simplify $T(n)$ to $T(n) = 3T(\frac{2n}{3}) + 1$.

We note that the first call to WEIRDSORT will operate on a list of n values. This call will cause *three* recursive calls to WEIRDSORT that each operate on a list of $\frac{2n}{3}$ values. These *three* recursive calls will themselves each cause three recursive calls to WeirdSort that each operate on a list of $\frac{4n}{9}$ values, for a total of *nine* recursive calls to WeirdSort that each operate on a list of $\frac{4n}{9}$ values. We can generalize this analysis by considering a call of WeirdSort on a list of size $n' = n\left(\frac{2}{3}\right)^i, 1 \le n'$. We have

(a) There will be 3^i calls of WeirdSort on lists of size $n' = n\left(\frac{2}{3}\right)^i$.

(b) The call to WeirdSort with a list of size $1 < n'$ will itself perform three recursive calls to WeirdSort with lists of size $\frac{2n'}{3} = n\left(\frac{2}{3}\right)^{i+1}$.

(c) The call to WeirdSort with a list of size $2 < n'$ will result in a constant amount of local work.

(d) The call to WeirdSort with a list of size $0 \le n' \le 2$ will do a constant amount of work.

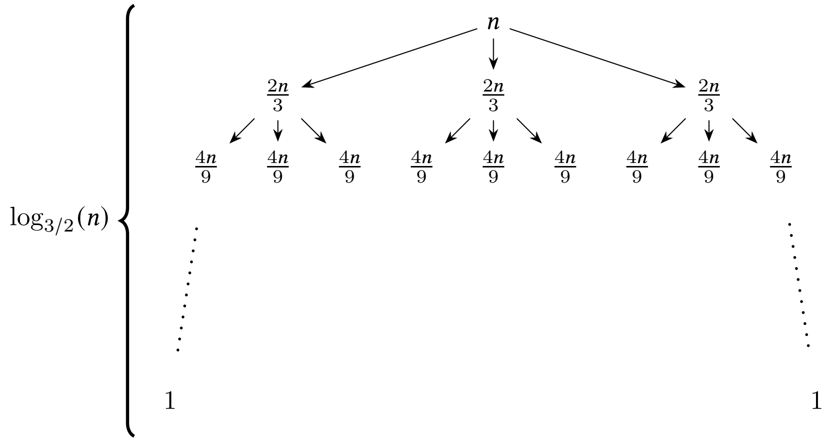
As each recursive call to WeirdSort reduces the size of the list being sorted by one-thirds, WeirdSort will perform recursive calls that go $\left\lfloor \log_{\frac{3}{2}}(n) \right\rfloor$ levels deep. We have drawn the recursion tree of WeirdSort based on the above analysis in Figure **??**. Based on this recursion tree, we conclude that

$$\begin{aligned} T(n) &= \sum_{i=0}^{\left\lfloor \log_{\frac{3}{2}}(n) \right\rfloor} 3^i \\ &\approx \sum_{i=0}^{\log_{\frac{3}{2}}(n)} 3^i \\ &= \frac{3^{\log_{\frac{3}{2}}(n)+1} - 1}{3-1} && (\textit{Geometric series}). \end{aligned}$$

We note that $\log_{\frac{3}{2}}(n) = \frac{\log_3(n)}{\log_3(3/2)}$. Next, we notice that $\frac{1}{\log_3(3/2)} = \log_{3/2}(3)$. Hence,

$$\begin{aligned} T(n) &\approx 3/2\left(\left(3^{\log_3(n)}\right)^{\log_{3/2}(3)} - 1\right) \\ &= 3/2\left(n^{\log_{3/2}(3)} - 1\right) \\ &= \Theta(n^{\log_{3/2}(3)}) = \Theta(n^{2.7095\ldots}). \end{aligned}$$

The following figure draws the resulting recursion tree.



Nodes	Local Work	Total
$1 = 2^0$	1	1
$3 = 3^1$	1	3^1
$9 = 3^2$	1	3^2
⋮	⋮	⋮
3^i	1	3^i
⋮	⋮	⋮
		+
		$\sum_{i=0}^{\left\lfloor \log_{\frac{3}{2}}(n) \right\rfloor} 3^i$

P4.2.5 Solve the recurrence $T(n)$ using the *Master Theorem* by proving that $T(n) = \Theta(f(n))$ for some function $f(n)$.

Solution. We can use the Master Theorem. We have $a = 3$, $b = 3/2$, and $f(n) = 1$. Note that $\log_b(a) = \log_{3/2}(3) \approx 2.7095\ldots$. We have $f(n) = 1 = n^0 = n^{\log_b(1)} = n^{\log_b(a-\epsilon)}$ with $\epsilon = a - 1 = 2 > 0$. Hence, Case 1 applies and we conclude $T(n) = \Theta(n^{\log_{3/2}(3)}) = \Theta(n^{2.7095\ldots})$.

5 Week 5

5.1 Material

Textbook, Chapter 2.3, 2.4, and 2.5.

5.2 Slides

Part 6.

5.3 Tutorial Exercises

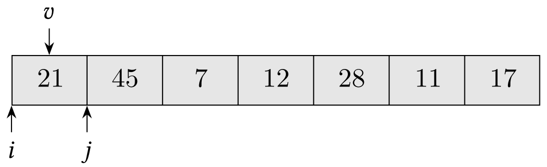
Exercises 2.3.5, 2.3.8, 2.3.9, 2.4.2, 2.4.4, 2.4.11, 2.4.12, 2.4.14, 2.4.20, 2.4.21, 2.5.22, 2.5.24.

Problem 5.1. Consider the following Partition algorithm used by QuickSort (this version of Partition is based on the algorithm from the slides with the for-loop replaced by a while-loop).

```
Algorithm Partition($L$, $start$, $end$) :
$v, i, j := L[start], start, start + 1$.
while $j \neq end$ do
    if $L[j] \leq v$ then
        $i := i + 1$.
        Exchange $L[i]$ and $L[j]$.
    end if
    $j := j + 1$.
end while
Exchange $L[i]$ and $L[start]$
return $i$.
```

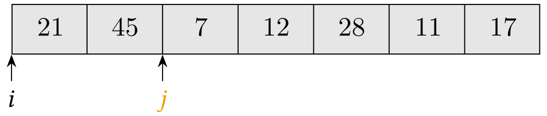
P5.1.1 Illustrate the operations performed by Partition on the array $A = [21, 45, 7, 12, 28, 11, 17]$. Show the content of A after each execution of the loop body.

Solution. Before the loop starts:



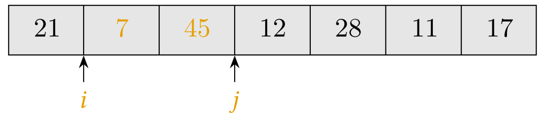
21	45	7	12	28	11	17

After one step of the loop:



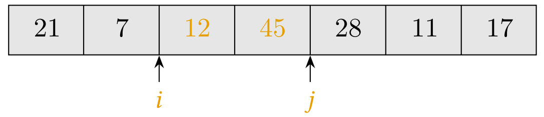
21	45	7	12	28	11	17

After two steps of the loop:



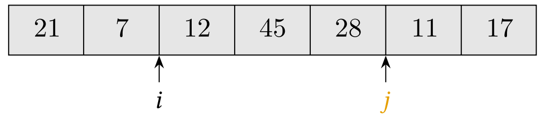
21	7	45	12	28	11	17

After three steps of the loop:



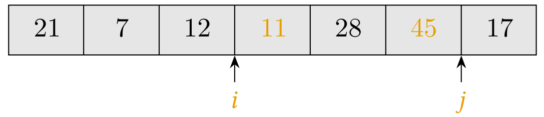
21	7	12	45	28	11	17

After four steps of the loop:



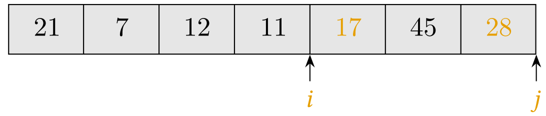
21	7	12	45	28	11	17

After five steps of the loop:



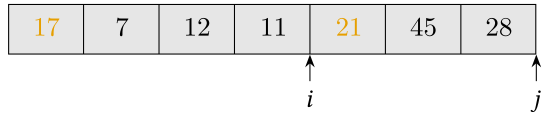
21	7	12	11	28	45	17

After six steps of the loop:



21	7	12	11	17	45	28

After Line 9:



17	7	12	11	21	45	28

P5.1.2 Provide pre-conditions and post-conditions for PARTITION and provide an invariant and bound function for the while-loop at Line 2. Prove the correctness of PARTITION.

Solution. We annotated the program with a pre-condition, invariant, bound function, and post-condition. We use $\mathcal{L}$ to denote the original value for list L.

Algorithm **PARTITION**(L, $start$, end) **:**

Pre: $\mathcal{L} = L[0\dots N)$ is a list with N values, $0 \leq start < end \leq N$.

```
1: $v, i, j := L[start], start, start + 1$.
2: while $j \neq end$ do
     inv: $start \leq i < j \leq end$, $v = L[start]$ is the pivot value (the value $\mathcal{L}[start]$). The values $L[start+1\dots i+1)$ are all values in the original list $\mathcal{L}[start+1\dots j)$ that are smaller-or-equal than the pivot value (except for the pivot itself). The values $L[i+1\dots j)$ are all values in the original list $\mathcal{L}[start+1\dots j)$ that are larger than the pivot value. The values $L[j\dots end)$ are identical to the values $\mathcal{L}[j\dots end)$ in the original list:
```

$start$	$start + 1$ i	i + 1	j	end
v	$\leq v$	$> v$	untouched	

```
     vf: $end - j$.
3:   if $L[j] \leq v$ then
4:     $i := i + 1$.
```

```
 5:        Exchange $L[i]$ and $L[j]$.
 6:     end if
 7:     $j := j + 1$.
 8: end while
 9: Exchange $L[i]$ and $L[\mathit{start}]$
10: return $i$.
```

Post: $L[i]$ is the pivot value (the value $v = \mathcal{L}[\mathit{start}]$). The values $L[\mathit{start} \dots i)$ are all values in the original list $\mathcal{L}[\mathit{start} + 1 \dots \mathit{end})$ that are smaller-or-equal than the pivot value (except for the pivot itself). The values $L[i+1 \dots \mathit{end})$ are all values in the original list $\mathcal{L}[\mathit{start}+1 \dots \mathit{end})$ that are larger than the pivot value.

To prove the correctness of Partition, we first prove that the invariant holds.

Before the loop, $L = \mathcal{L}$. Hence, $v = L[\mathit{start}] = \mathcal{L}[\mathit{start}]$ holds. In addition, $L[\mathit{start} + 1 \dots i + 1) = L[\mathit{start} + 1 \dots \mathit{start} + 1] = \emptyset$ and $L[i + 1 \dots j] = L[\mathit{start} + 1 \dots \mathit{start} + 1] = \emptyset$. Hence, the invariant holds.

To prove that the invariant is maintained by the loop, we perform a case distinction based on the **if**-test of Line 3. We have two cases:

- Case $L[j] \leq v$. In this case, we execute Lines 4, 5, and 7.

  Before executing Line 4, we have $j \neq \mathit{end}$, $L[j] \leq v$, and the invariant. Let L_{old}, i_{old}, and j_{old} be the values of L, i, and j before executing these three lines.

  After executing Line 7, $i = i_{\text{old}} + 1$ (Line 4), $L_{\text{old}}[i]$ and $L_{\text{old}}[j_{\text{old}}]$ are exchanged (Line 5), and $j = j_{\text{old}} + 1$ (Line 7). Only positions i and j_{old} in L are exchanged, all other values in L remain unchanged.

  As $j_{\text{old}} \neq \mathit{end}$ (while-loop), $\mathit{start} \leq i_{\text{old}} < j_{\text{old}} \leq \mathit{end}$ (invariant), $i = i_{\text{old}} + 1$, and $j = j_{\text{old}} + 1$, we must have $\mathit{start} \leq i < j \leq \mathit{end}$.

  By $L[j_{\text{old}}] \leq v$ (if-statement) and the invariant (values $L[i + 1 \dots j_{\text{old}})$ are all values in the original list $\mathcal{L}[\mathit{start} + 1 \dots j_{\text{old}})$ that are smaller-or-equal than the pivot value (except for the pivot itself)), we can conclude that values $L[\mathit{start} \dots i + 1)$ are all values in the original list $\mathcal{L}[\mathit{start} + 1 \dots j)$ that are smaller-or-equal than the pivot value (except for the pivot itself). With a similar argument, we can also conclude that values $L[i + 1 \dots j)$ are all values in the original list $\mathcal{L}[\mathit{start} + 1 \dots j)$ that are larger than the pivot value. Hence, the invariant holds after executing Line 7.

- Case $L[j] > v$. In this case, we execute only Line 7.

  Before executing Line 7, we have $j \neq \mathit{end}$, $L[j] > v$, and the invariant.

  Let j_{old} be the value of j before Line 7. All other variables remain unchanged. As $j_{\text{old}} \neq \mathit{end}$ (while-loop), $i < j_{\text{old}} \leq \mathit{end}$ (invariant), and $j = j_{\text{old}} + 1$, we must have $i < j \leq \mathit{end}$.

  By $L[j_{\text{old}}] > v$ (if-statement) and the invariant (values $L[\mathit{start} + 1 \dots i + 1)$ are all values in the original list $\mathcal{L}[\mathit{start} + 1 \dots j_{\text{old}} + 1) = \mathcal{L}[\mathit{start} + 1 \dots j)$ that are smaller-or-equal than the pivot value (except for the pivot itself)), we can also conclude that values $L[i + 1 \dots j)$ are all values in the original list $\mathcal{L}[\mathit{start} + 1 \dots j)$ that are larger than the pivot value. Hence, the invariant holds after executing Line 7.

In both cases, the invariant holds again after Line 7. Hence, the loop maintains the invariant.

Finally, we have to prove that the post-condition holds. Before Line 9, the invariant holds and we have $j = \mathit{end}$ (as the **while**-loop at Line 2 ended). Hence, $L[\mathit{start}]$ is the pivot value (the value $v = \mathcal{L}[\mathit{start}]$), the values $L[\mathit{start} + 1 \dots i + 1)$ are all values in the original list $\mathcal{L}[\mathit{start} + 1 \dots \mathit{end})$ that are smaller-or-equal than the pivot value (except for the pivot itself), and the values $L[i + 1 \dots \mathit{end})$ are all values in the original list $\mathcal{L}[\mathit{start} + 1 \dots \mathit{end})$ that are larger than the pivot value.

The exchange of Line 9 swaps the pivot at $L[\mathit{start}]$ with the value $L[i]$. Due to the above, we know that $L[i]$ was a value smaller-or-equal than the pivot value. Hence, after Line 9, we have $L[i]$ is the pivot value (the value $v = \mathcal{L}[\mathit{start}]$), the values $L[\mathit{start} \dots i)$ are all values in the original list $\mathcal{L}[\mathit{start} + 1 \dots \mathit{end})$ that are smaller-or-equal than the pivot value (except for the pivot itself), and the values $L[i + 1 \dots \mathit{end})$

are all values in the original list $\mathcal{L}[\mathit{start}+1\ldots\mathit{end})$ that are larger than the pivot value. Hence, the post-condition holds.

To complete the proof, we have to prove that the while-loop terminates. It is straightforward to verify that the provided function is a bound function, proving termination.

P5.1.3 Argue how Partition can be adjusted to run on singly linked lists L, while keeping a running time of $\mathcal{O}(|L|)$.

Solution. In the original version of Partition, we exchange values pointed to by positions *start* (input), i (starts at *start*), and j (starts at $\mathit{start}+1$), and we change the positions i and j only by going to their next positions. We can turn *start*, i, and j into pointers to singly linked list nodes. To exchange values of two singly linked list nodes n_1 and n_2, we simply exchange $n_1.\mathit{item}$ and $n_2.\mathit{item}$. To go to the next position of a node n, we simply follow the next-pointer $n.\mathit{next}$.

To make this all work, the singly linked list version of partition must be called with singly linked list nodes *start* and *end* that indicate the range of the list to be partitioned. The resulting algorithm will return the singly linked list node i that holds the partition value.

Problem 5.2. Consider pairs (x_i, y_i) such that x_i is the time at which person $i = 0, 1, 2, \ldots$ enters the museum and y_i is the time at which person i leaves the museum. You may assume that consecutive people enter the museum in order of increasing time $(x_0 \leq x_1 \leq \ldots)$.

P5.2.1 Provide an algorithm MaxVisitors that takes as input $L = [(x_0, y_0), \ldots, (x_{N-1}, y_{N-1})]$ and computes in $\mathcal{O}(N \log N)$ the maximum number of visitors in the museum at any time.

Solution.

```
Algorithm MaxVisitors(L) :
Pre: $L = [(x_0, y_0), \ldots, (x_{N-1}, y_{N-1})]$ is a list of (enter time, leave time)-pairs with $(x_i \leq y_i)$, $0 \leq i < N$,
    (one enters before leaving) that is ordered on enter-times.
 1: $m := 0$.
 2: $wla$ := an empty min-heap.
 3: for all $(x, y) \in L$ do
 4:     while the minimum value $v$ in $wla$ is less-than $y$ do
 5:         DelMin($wla$).
 6:     end while
 7:     Add($wla$, $y$).
 8:     $m := \max(m, \text{Size}(wla))$.
 9: end for
10: return $m$.
```

P5.2.2 Argue why your algorithm MaxVisitors is correct and has a runtime complexity of $\mathcal{O}(N \log N)$.

Solution. The min-heap *wla* holds all leave-times of persons still in the museum after the previous person entered the museum. When a new person p arrives at time x, we remove everyone from the heap that has already left before x (the **while**-loop at Line 4). Next, we add y to the min-heap indicating that p will leave at y. Hence, at Line 8, we set m to the maximum of any previous occupancy of the museum or the occupancy of the museum after p arrived.

In the end, we add and remove every person once from the min-heap: we perform N DelMin and N Add operations on a min-heap with at-most N values. Hence, the total cost is worst-case $\mathcal{O}(N \log_2(N))$.

P5.2.3 Assume that the museum has a maximum capacity of M. Provide a datastructure with an operation PersonEnters(x_i, y_i) that computes in at-most $\mathcal{O}(\log M)$ the number of visitors in the museum when person i enters the museum (for any number of persons).

Solution. We use a min-heap *wla* and the operation PERSONENTERS(x_i, y_i) performs one pass of the **for**-loop of MAXVISITORS. After doing so, we simply return SIZE(*wla*) to report the current occupancy of the museum.

P5.2.4 Argue why your algorithm PERSONENTERS is correct and has a runtime complexity of $\mathcal{O}(\log M)$.

Solution. As the maximum occupancy is M, the min-heap will hold at-most M leave times for persons that are still in the museum after the previous person entered the museum.

After adding the i-th person, we have performed i ADD operations and at-most $i-1$ DELMIN operations on a min-heap with at-most M values. Hence, the total cost is worst-case $\mathcal{O}(i \log_2 M)$, which we amortize over i persons. Hence, the amortized cost per person is $\mathcal{O}(\log_2(M))$.

6 Week 6

6.1 Material

Textbook, Section 3.1 and Section 3.2.

6.2 Slides

Part 7 and 8.

6.3 Tutorial Exercises

Exercises 3.1.13, 3.1.14, 3.2.2, 3.2.5, 3.2.11, 3.2.22, 3.2.23, 3.2.29, 3.2.31, 3.2.37.

Problem 6.1. Consider the sequence of values $S = [3, 42, 39, 86, 49, 89, 99, 20, 88, 51, 64]$.

P6.1.1 Draw the min heap (as a tree) obtained by adding the values in S in sequence. Show each step.

Solution.
```
Add 3:
3

Add 42:
3 (42 *)

Add 39:
3 (42 39)

Add 86:
3 (42 (86 *) 39)

Add 49:
3 (42 (86 49) 39)

Add 89:
3 (42 (86 49) 39 (89 *))

Add 99:
3 (42 (86 49) 39 (89 99))

Add 20:
3 (20 (42 (86 *) 49) 39 (89 99))

Add 88:
3 (20 (42 (86 88) 49) 39 (89 99))

Add 51:
3 (20 (42 (86 88) 49 (51 *)) 39 (89 99))

Add 64:
3 (20 (42 (86 88) 49 (51 64)) 39 (89 99))
```

P6.1.2 Draw the max heap (as a tree) obtained by adding the values in S in sequence. Show each step.

Solution.
```
Add 3:
3
```

```
Add 42:
42 (3 *)

Add 39:
42 (3 39)

Add 86:
86 (42 (3 *) 39)

Add 49:
86 (49 (3 42) 39)

Add 89:
89 (49 (3 42) 86 (39 *))

Add 99:
99 (49 (3 42) 89 (39 86))

Add 20:
99 (49 (20 (3 *) 42) 89 (39 86))

Add 88:
99 (88 (49 (3 20) 42) 89 (39 86))

Add 51:
99 (88 (49 (3 20) 51 (42 *)) 89 (39 86))

Add 64:
99 (88 (49 (3 20) 64 (42 51)) 89 (39 86))
```

P6.1.3 Draw the binary search tree obtained by adding the values in *S* in sequence. Show each step.

Solution. Add 3:

```
3

Add 42:
3 (* 42)

Add 39:
3 (* 42 (39 *))

Add 86:
3 (* 42 (39 86))

Add 49:
3 (* 42 (39 86 (49 *)))

Add 89:
3 (* 42 (39 86 (49 89)))

Add 99:
3 (* 42 (39 86 (49 89 (* 99))))

Add 20:
3 (* 42 (39 (20 *) 86 (49 89 (* 99))))
```

```
Add 88:
3 (* 42 (39 (20 *) 86 (49 89 (88 99))))

Add 51:
3 (* 42 (39 (20 *) 86 (49 (* 51) 89 (88 99))))

Add 64:
3 (* 42 (39 (20 *) 86 (49 (* 51 (* 64)) 89 (88 99))))
```

Problem 6.2. Consider non-empty binary search trees T_1 and T_2 such that all values in T_1 are smaller than the values in T_2. The SetUnion operation takes binary search trees T_1 and T_2 and returns a binary search trees holding all values originally in T_1 and T_2 (destroying T_1 and T_2 in the process).

P6.2.1 Assume the binary search trees storing T_1 and T_2 have the same height h. Show how to implement the SetUnion operation in $\mathcal{O}(h)$ such that the resulting tree has a height of at-most $h + 1$.

Solution. Remove the *minimum* value v from T_2, resulting in tree T_2'.

We can do so in at-most $\mathcal{O}(h)$. By doing so, the tree T_2 has a height of either h or $h - 1$.

P6.2.2 Assume that T_1 and T_2 are red-black trees with the same black height h. Show how to implement a SetUnion operation that returns a red-black tree in $\mathcal{O}(h)$.

Solution. We construct the tree with root node n with value v such that the root of T_1 is the left child of n and the root of T_2' is the right child of n.

We can do so in at-most $\mathcal{O}(1)$. By doing so, the resulting tree has height $\max(h_1, h_2)$ in which $h_1 = h$ is the height of T_1 and $h - 1 \leq h_2 \leq h$ is the height of T_2'. Hence, the resulting tree has height $h + 1$.

P6.2.3 Assume that T_1 and T_2 are red-black trees with black heights $h_1 > h_2$. Show how to implement a SetUnion operation that returns a red-black tree in $\mathcal{O}(h_1)$.

Solution. We generalize the strategy used above:

- Remove the *minimum* value v from T_2, resulting in tree T_2'.

  We can do so in at-most $\mathcal{O}(h_2)$. By doing so, the tree T_2 has a black height h_2' of either h_2 or $h_2 - 1$.
- Find the unmarked node m on the path from the root in T_1 to the maximum value in T_1 (hence, along the right-most path in T_1) such that m has $h_1 - h_2'$ unmarked ancestors.

  We can do so in at-most $\mathcal{O}(h_2)$.
- Change the tree T_1 by replacing node m by a new node n with value v such that the node m is the left child of n and the root r_2' of T_2' is the right child of n. Color both node m and r_2'.

  We can do so in at-most $\mathcal{O}(1)$. We note that we have not changed the black height along the path from the root of the resultant tree to any leaf.
- Both children of the new node n are color marked. Hence, this is not a left-leaning red-black tree. This coloring anomaly also occurs when we add new values to the tree, and we use the same method to deal with this anomaly (push the color up and rotate when necessary to put the color on a left-leaning branch).

  We can do so in at-most $\mathcal{O}(h_2)$ (the length of the path from n to the root of its tree).

7 Week 7

7.1 Material

Textbook, Section 3.3 and Section 3.4.

7.2 Slides

Part 9.

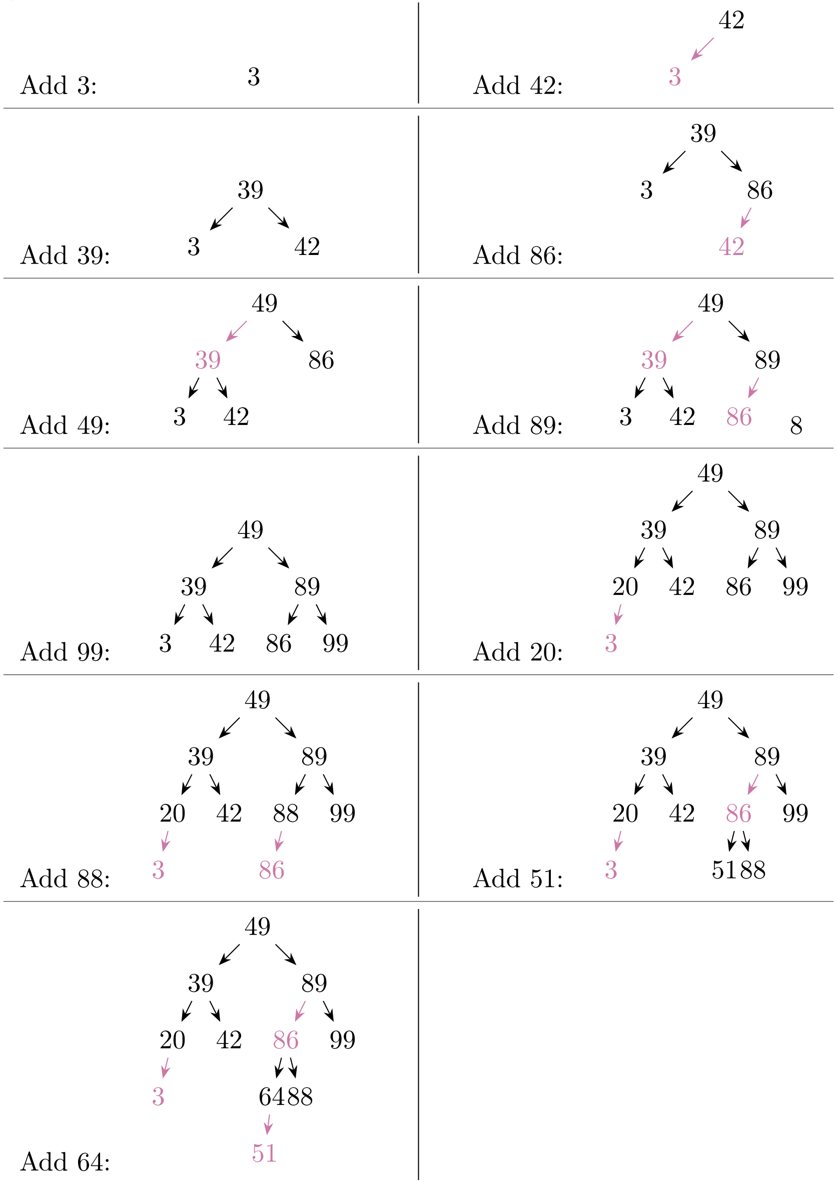
7.3 Tutorial Exercises

Exercises 3.3.4, 3.3.9, 3.3.13, 3.3.15, 3.3.37, 3.4.5, 3.4.12, 3.4.32, 3.4.33.

Problem 7.1. Consider the sequence of values $S = [3, 42, 39, 86, 49, 89, 99, 20, 88, 51, 64]$.

P7.1.1 Draw the *left-leaning* red-black tree obtained by adding the values in S in sequence. Show each step.

Solution.



P7.1.2 Consider the hash function $h(x) = (x + 7) \bmod 13$ a hash-table of 13 table entries that uses hashing with separate chaining. Draw the hash-table obtained by adding the values in S in sequence. Show each step.

Solution. First, we compute $h(x)$ for each value $x \in S$. See the following table:

x	$x + 7$	$(x + 7) \bmod 13$
3	10	10
42	49	10
39	46	7
86	93	2
49	56	4
89	96	5
99	106	2
20	27	1
88	95	4
51	58	6
64	71	6

Note: in the below, I will add new values to the back of the list in each table slot. If the student *consistently* adds all values to the front of the list, then that is equally correct!

Add 3:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Value											[3]		

Add 42:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Value											[3,42]		

Add 39:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Value								[39]			[3,42]		

Add 86:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Value			[86]					[39]			[3,42]		

Add 49:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Value			[86]		[49]			[39]			[3,42]		

Add 89:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Value			[86]		[49]	[89]		[39]			[3,42]		

Add 99:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Value			[86,99]		[49]	[89]		[39]			[3,42]		

Add 20:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Value		[20]	[86,99]		[49]	[89]		[39]			[3,42]		

Add 88:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Value		[20]	[86,99]		[49,88]	[89]		[39]			[3,42]		

Add 51:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Value		[20]	[86,99]		[49,88]	[89]	[51]	[39]			[3,42]		

Add 64:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Value		[20]	[86,99]		[49,88]	[89]	[51,64]	[39]			[3,42]		

P7.1.3 Consider the hash function $h(x) = (x + 7) \bmod 13$ a hash-table of 13 table entries that uses hashing with linear probing. Draw the hash-table obtained by adding the values in S in sequence. Show each step.

Solution. Same hash function as before. Hence, we reuse the table from before.

Add 3:	Index	0	1	2	3	4	5	6	7	8	9	10	11	12
	Value											3		

Add 42:	Index	0	1	2	3	4	5	6	7	8	9	10	11	12
	Value											3	42	

Add 39:	Index	0	1	2	3	4	5	6	7	8	9	10	11	12
	Value								39			3	42	

Add 86:	Index	0	1	2	3	4	5	6	7	8	9	10	11	12
	Value			86					39			3	42	

Add 49:	Index	0	1	2	3	4	5	6	7	8	9	10	11	12
	Value			86		49			39			3	42	

Add 89:	Index	0	1	2	3	4	5	6	7	8	9	10	11	12
	Value			86		49	89		39			3	42	

Add 99:	Index	0	1	2	3	4	5	6	7	8	9	10	11	12
	Value			86	99	49	89		39			3	42	

Add 20:	Index	0	1	2	3	4	5	6	7	8	9	10	11	12
	Value		20	86	99	49	89		39			3	42	

Add 88:	Index	0	1	2	3	4	5	6	7	8	9	10	11	12
	Value		20	86	99	49	89	88	39			3	42	

Add 51:	Index	0	1	2	3	4	5	6	7	8	9	10	11	12
	Value		20	86	99	49	89	88	39	51		3	42	

Add 64:	Index	0	1	2	3	4	5	6	7	8	9	10	11	12
	Value		20	86	99	49	89	88	39	51	64	3	42	

Do *not* spend time drawing beautiful trees or tables: a clear textual representation is good enough.

Problem 7.2. Consider a set of strings S. We want to figure out whether S has duplicates efficiently. We do not want to do so by sorting S and then checking for duplicates: comparing strings can be a lot of work (e.g., they might differ in only a single character).

Assume that you have a hash function h that can compute a suitable hash code for any string $s \in S$ in $\mathcal{O}(|s|)$. Show how one can use hashing to find whether S has duplicates without performing many comparisons between strings. Your algorithm should have an expected runtime of $\mathcal{O}(|S|)$ in which $|S| = \sum_{s \in S} |s|$ represents the total length of all strings in S.

Solution. Let $s_0, \dots, s_{n-1}$ be the strings in S. To detect duplicates, we will add each string s_i, $0 \leq i < n$, to a hash table. When we find a collision between s_i and some other string $s' \in S$, we are very likely to have found a duplicate. We still need to verify that s_i and s' are equivalent, however, which takes at-most $|s_i|$ steps per collision.

To assure we are unlikely to have collisions for non-duplicate values, we construct a hash table that is sufficiently large: we choose $M = 2 \cdot n$ entries in our table and we use chaining to store collisions. When adding string s_i, we expect to find $\frac{i}{2n} \leq \frac{n}{2n} = \frac{1}{2}$ other strings s' with $h(s_i) = h(s')$. Hence, we expect at-most $\frac{|s_i|}{2}$ steps to compare s_i with any other strings s'. In addition, we need to compute the hash value $h(s_i)$, before we can add s_i to the hash table. Computing $h(s_i)$ will take $|s_i|$ steps. Hence, the total expected cost is upper-bounded by $\sum_{s_i \in S} \frac{3|s_i|}{2} = \mathcal{O}(|S|)$.

8 Week 8

8.1 Material

Textbook, Section 4.1 and 4.2.

8.2 Slides

Part 10.

8.3 Tutorial Exercises

Exercises 4.1.1, 4.1.10, 4.1.12, 4.1.14, 4.1.18, 4.2.1, 4.2.10, 4.2.12, 4.2.13, 4.2.18, 4.2.19, 4.2.22, 4.2.23, 4.2.25, 4.2.27.

Problem 8.1. Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and assume we have a weight function $\mathit{weight} : \mathcal{E} \rightarrow \{1, \dots, W\}$ with W some maximum integer value W.

P8.1.1 Assume $W = 1$ (all edges have weight 1). Given node $n \in \mathcal{N}$, provide an algorithm that can compute the shortest path from node n to any other node m in $\mathcal{O}(|\mathcal{N}| + |\mathcal{E}|)$. In this case, the shortest path is the path with the fewest edges.

Solution. We can perform breadth-first search to compute the shortest paths from node s to all nodes m. To record the paths, we use an array *paths* in which $\mathit{paths}[v]$ points to the preceding node on the path from n to v.

```
Algorithm BFS-ShortestPaths($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$) :
paths := $\{n \mapsto (n \neq s) \mid n \in \mathcal{N}\}$.
path := $\{n \mapsto ? \mid n \in \mathcal{N}\}$.
$Q$ := a queue holding only $s$.
while $\neg$Empty($Q$) do
    $n$ := Dequeue($Q$).
    for all $(n, m) \in \mathcal{E}$ do
        if $\neg$marked[$m$] then
            marked[$m$] := true.
            paths[$m$] := $n$.
            Enqueue($S, m$).
        end if
    end for
end while
return paths.
```

We use an adjacency list represention in which the above breadth-first search BFS-ShortestPaths has a complexity of $\Theta(|\mathcal{N}| + |\mathcal{E}|)$.

P8.1.2 Given node $n \in \mathcal{N}$, provide an algorithm that can compute the shortest path (in terms of the sum of the weights of edges on the path) from node n to any other node m in $\mathcal{O}(|\mathcal{N}| + W|\mathcal{E}|)$.

Solution. Consider an edge (u, v) with weight $w, 1 < w \leq W$. We can introduce nodes $z_1, \dots, z_{w-1}$ and replace the edge (u, v) by edges (u, z_1), (z_1, z_2), ..., (z_{w-1}, v) all of weight 1. Now there is a *path* from u to v whose total cost is w *instead* of an edge of weight w.

We can apply this transformation on $\mathcal{G}$, resulting in $\mathcal{G}^T$. By doing so, we will introduce at-most $(W-1)|\mathcal{E}|$ new nodes and replace all edges in the original graph by $(W-1)|\mathcal{E}|$. Hence, using an adjacency list representation, the above BFS-ShortestPaths algorithm applied on $\mathcal{G}^T$ has a complexity of $\Theta(|\mathcal{N}| + W|\mathcal{E}| + W|\mathcal{E}|) = \Theta(|\mathcal{N}| + W|\mathcal{E}|)$.

Note that this breadth-first search on $\mathcal{G}^T$ computes paths in $\mathcal{G}^T$ and not in $\mathcal{G}$. Hence, we also need to show how to reduce the paths in $\mathcal{G}^T$ to paths in $\mathcal{G}$. We can do so with minor changes to the above BFS-ShortestPaths algorithm:

```
Algorithm BFS-ShortestPaths-W($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $s \in \mathcal{N}$) :
$\mathcal{N}', \mathcal{E}' := \mathcal{N}, \emptyset$.
for all $(m, n) \in \mathcal{E}$ do
    Add fresh $w - 1 = \mathit{weight}((m, n)) - 1$ nodes $z_1, \dots, z_{w-1}$ to $\mathcal{N}'$.
    Add edges $(m, z_1), (z_1, z_2), \dots, (z_{w-1}, n)$ to $\mathcal{E}'$.
end for
$\mathit{marked} := \{n \mapsto (n \neq s) \mid n \in \mathcal{N}'\}$.
$\mathit{paths} := \{n \mapsto \mathit{?} \mid n \in \mathcal{N}'\}$.
$Q$ := a queue holding only $s$.
while $\neg$Empty($Q$) do
    $n$ := Dequeue($Q$).
    for all $(n, m) \in \mathcal{E}'$ do
        if $\neg \mathit{marked}[m]$ then
            $\mathit{marked}[m]$ := true.
            if $n \in \mathcal{N}$ then /* id($n$) $\geq |\mathcal{N}|$ */
                $\mathit{paths}[m] := n$.
            else
                $\mathit{paths}[m] := \mathit{paths}[n]$.
            end if
            Enqueue($S, m$).
        end if
    end for
end while
return $\mathit{paths}$.
```

We use an adjacency list represention. The complexity of the above is $\Theta(|\mathcal{N}| + W|\mathcal{E}|)$.

P8.1.3 Given node $n \in \mathcal{N}$, provide an algorithm that can compute the shortest path (in terms of the sum of the weights of edges on the path) from node n to any other node m in $\mathcal{O}(W|\mathcal{N}| + |\mathcal{E}|)$.

Solution. We will improve on the above solution. The above solution is rather verbose: we replace each distinct edge of weight w into a long sequence of w edges. We can adopt a much more efficient translation.

Consider two edges (u_1, v) and (u_2, v) with weights w_1 and w_2. To simplify presentation, we assume $w_1 \geq w_2$. If we follow the above approach, we end up with two distinct paths

$$(u_1, z_{1,1}), (z_{1,1}, z_{1,2}), \dots, (z_{1,w_1-1}, v) \text{ and } (u_2, z_{2,1}), (z_{2,1}, z_{2,2}), \dots, (z_{2,w_2-1}, v).$$

These path are directed paths that *only* lead to v. Hence, there is no need to have two distinct paths: we can reuse part of the path $(u_1, z_{1,1}), (z_{1,1}, z_{1,2}), \dots, (z_{1,w_1-1}, v)$ in the construction of the path from u_2 to v by adding the edge (u_2, z_{1,w_1-w_2+1}). Take, for example $w_1 = 6$ and $w_2 = 3$. We end up:

$$(u_1, z_{1,1}), (z_{1,1}, z_{1,2}), (z_{1,2}, z_{1,3}), (z_{1,3}, z_{1,4}), (z_{1,4}, z_{1,5}), (z_{1,5}, v) \text{ and } (u_2, z_{1,4}), (z_{1,4}, z_{1,5}), (z_{1,5}, v).$$

The last two edges of this second path are also in the first path! Hence, we went from $w_1 + w_2$ edges to only $w_1 + 1$ edges. We can apply this same idea to *all* edges leading to v. To do so, we need a path $(z_1, z_2), (z_2, z_3), \dots, (z_{W-1}, v)$ of $w - 1$ edges leading to v and a single edge $(u, z_{W-\mathit{weight}((u,v))+1})$ *per* edge (u, v) that leads onto this path.

We can apply this transformation on $\mathcal{G}$, resulting in $\mathcal{G}^S$. By doing so, we will introduce at-most $(W-1)|\mathcal{N}|$ new nodes and replace all edges in the original graph by $(W-1)|\mathcal{N}| + |\mathcal{E}|$ edges. Next, we can perform an algorithm similar to BFS-ShortestPaths-W to solve the requested problem. We use an adjacency list represention, in which case this algorithm will have a complexity of $\Theta((W-1)|\mathcal{N}| + |\mathcal{E}|) = \Theta(W|\mathcal{N}| + |\mathcal{E}|)$.

For each question, explain why your algorithm is correct, why your algorithm achieves the stated complexity, and which graph representation you use.

Problem 8.2. Consider an $m \times n$ game board in which each cell has a numeric value, e.g.,

	A	B	C	D	E	F
G	1	2	2	3	4	2
H	3	4	4	4	4	1
I	1	4	1	3	1	4
J	2	3	1	4	1	2
K	3	3	2	2	4	2

A player starts the game with a token in the top-left cell (the cell GA in this example) and the player finishes the game by moving to the bottom-right cell (the cell KF in this example). In each round of the game, the player can move in *four* directions (up, down, left, and right). The distance of each move is determined by the value of the cell. When going over the border of the game board, one ends up on the other side. For example, if the player is in the cell JB, which has value 3, then the player can move 3 steps up (reaching GB), 3 steps right (reaching JE), 3 steps down (reaching HB), and 3 steps left (reaching JE).

The score of a player is determined in the total number of rounds the player needs to reach the bottom-right cell. For example, a player can move from GA to KA, from KA to KD, and from KD to KF, for a score of three rounds.

P8.2.1 Model the above problem as a graph problem: what are the nodes and edges in your graph, do the edges have weights, and what problem are you trying to answer on your graph?

Solution. The *nodes* are the cells in our game board, for example, each of the cells JB, GB, JE, HB, and JE is a node.

The *directed edges* connect cells to the cells they can reach. Each cell has at-most outgoing four edges: one for each direction. For example, cell JB has outgoing edges (JB, GB), (JB, JE), (JB, HB), and (JB, JE). Note that going up or down or going left or right might lead to the same node. Hence, some nodes might have less-than four edges.

Edges do *not have weight*: taking one edge corresponds to one round of the game.

The problem asks to find the shortest path (in terms of the number of edges) from the node representing the cell in the top-left corner to the node representing the cell in the bottom-right corner.

P8.2.2 Provide an efficient algorithm that, given a $m \times n$ game board, will find an optimal solution (a minimum number of steps to reach the bottom-right cell) if such a solution exists. If the game board has no solution, then the algorithm should report that the game board is invalid. The runtime of your algorithm should be worst-case $\mathcal{O}(mn)$.

Solution. We use the BFS-based single-source shortest-path algorithm for unweighted graphs starting at the node representing the top-left corner. This algorithm will provide details on the path to the node representing the cell in the bottom-right corner if such a path exist.

Note that we have $n \times m$ nodes. As each node has at-most *four* outgoing edges, we have at-most $4 \times n \times m$ edges.

P8.2.3 Explain why your algorithm is correct and has a complexity that is worst-case $\mathcal{O}(mn)$.

Solution. Correct by construction. The standard BFS-based single-source shortest-path algorithm on a adjacency list representation of graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ has a complexity of $\mathcal{O}(|\mathcal{N}|+|\mathcal{E}|)$. As $|\mathcal{N}| \leq |\mathcal{E}| \leq 4 \times n \times m$ in our representation, we have $\mathcal{O}(|\mathcal{N}| + |\mathcal{E}|) = \mathcal{O}(nm)$.

P8.2.4 Which of the two *graph representation* we saw in the course material did you use to store the game board? What would the complexity of your algorithm be if you used the other graph representation?

Solution. We can easily read out a matrix-based game board representation as an adjacency list by computing the at-most 4 outgoing edges of each node when needed. For example, the cell reached from JB (in column 1) when going left by 3 steps is in column $-2 \bmod 6 = 4$ (as we have 6 columns). Hence, the cell reached is cell JE.

If we used the matrix representation, then the size of this matrix becomes $(nm)^2$ and the complexity of the standard BFS-based single-source shortest-path algorithm on this representation becomes $\mathcal{O}((nm)^4)$.

9 Week 9

9.1 Material

Textbook, Section 4.2 and 4.3.

9.2 Additional Material

CLRS, Chapter 20 and 21.

9.3 Slides

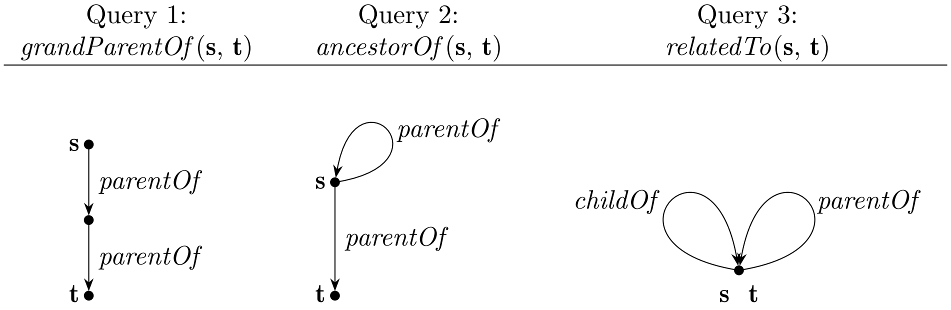
Part 10 and 11.

9.4 Tutorial Exercises

Exercises 4.2.32, 4.2.33, 4.2.35, 4.3.1, 4.3.3, 4.3.5, 4.3.7, 4.3.12, 4.3.13, 4.3.14, 4.3.15, 4.3.19, 4.3.20.

Problem 9.1. Edge-labeled graphs are graphs in which edges have labels that represent the type of relationship that is expressed by that edge. For example, in a social network graph, the edges could be labeled *parentOf*, *friendOf*, and *worksWith*.

One way to express *graph queries* (that express how new information can be derived from edge-labeled graphs) is via a *query graph* that expresses how new relationships between source node **s** and target node **t** can be derived from existing information. Consider, for example, the following three example query graphs:


Query 1: *grandParentOf*(**s**, **t**)
Query 2: *ancestorOf*(**s**, **t**)
Query 3: *relatedTo*(**s**, **t**)

The first query relates nodes that represent grandparents and their grandchildren, the second query relates nodes that represent ancestors and their descendants, and the third query relates everyone with a direct family relationship.

Let Q be a graph query and G be an edge-labeled graph representing a data set. The *graph query evaluation* problem is the problem of computing the derived relationship in G expressed by Q.

Typically, queries are *small* and data graphs are enormous. Hence, here we will assume that the size of a query graph is *constant*.

P9.1.1 Model the *graph query evaluation* problem as a graph problem: what are the nodes and edges in your graph, do the edges have weights, and what problem are you trying to answer on your graph?

Solution. Consider a node n in graph $G = (\mathcal{N}, \mathcal{E})$ and query $Q = (\mathcal{N}_Q, \mathcal{E}_Q)$. To determine whether the pair (n, m) is in the output of query Q on graph G, we have to find two paths:

(a) a path $\pi = ne_1u_1 \dots u_{k-1}e_km$ using edges $e_1, \dots, e_k \in \mathcal{E}$ and visiting nodes $u_1, \dots, u_{k-1} \in \mathcal{N}$ in graph G; and

(b) a path $\tau = \mathbf{s}f_1v_1 \dots v_{k-1}f_k\mathbf{t}$ using edges $f_1, \dots, f_k \in \mathcal{E}_Q$ and visiting nodes $v_1, \dots, v_{k-1} \in \mathcal{N}_Q$ in the query graph Q

such that matching edges on these two paths have *identical labeling*. Hence, $l_1 = \text{label}(u_1) = \text{label}(f_1)$, $\dots$, $l_k = \text{label}(u_k) = \text{label}(f_k)$.

In the following, we will write (p, l, q) to denote an edge from node p to node q labeled with label l.

We can construct a new joint-graph $J = (\mathcal{N}_J, \mathcal{E}_J)$ in which paths represent exactly these kinds of pairs-of-paths. We do so using the following product construction:

$$\mathcal{N}_J = \{(e, f) \mid e \in \mathcal{N} \wedge f \in \mathcal{N}_Q\};\ \text{and}\mathcal{E}_J \quad = \{((u, f), l, (u', f')) \mod (u, l, u') \in \mathcal{E} \wedge (f, l, f') \in \mathcal{E}_Q\}.$$

By construction, we have paths π in G, τ in Q if and only if we have path

$$(n, \mathbf{s})((n, s), l_1, (u_1, f_1))(u_1, f_1) \dots (u_{k-1}, f_{k-1})((u_{k-1}, f_{k-1}), l_k, (m, \mathbf{t}))(m, \mathbf{t})$$

in graph J. Hence, to determine whether the pair (n, m) is in the output of query Q on graph G, we simply check whether $(m, \mathbf{t})$ is reachable from $(n, \mathbf{s})$.

P9.1.2 Provide an efficient algorithm that, given a a graph G, a source node n in graph G, and query Q, will find all nodes m such that the pair (n, m) is in the derived relationship in G expressed by Q. Assuming Q has a constant time, the runtime of your algorithm should be worst-case $\mathcal{O}(|G|)$ in which $|G|$ is the total number of nodes and edges in G.

P9.1.3 Explain how you represented your graph G, why your algorithm is correct, and why your algorithm has a complexity that is worst-case $\mathcal{O}(|G|)$.

Problem 9.2. Consider a remote community of N houses $h_1, \dots h_N$. We want to provide each house with internet access at minimal cost.

We can make a local network between these houses by connecting them via long-range Wi-Fi using directional antennas. The cost to connect two houses h_i, h_j is given by $C(h_i, h_j)$ and will depend on their distance and the terrain in between (e.g., long distances and hills require strong radios and repeaters).

In addition, we can connect one or more houses directly to the internet via a new fiber connection. For house h_i, the cost of this fiber connection is $F(h_i)$. If a house has a direct internet connection, then it can share this connection with all other houses that are reachable via a path of long-range Wi-Fi connections.

The community wants to find how to connect all members of this community with internet at a minimal cost.

P9.2.1 Model the above problem as a graph problem: what are the nodes and edges in your graph, do the edges have weights, and what problem are you trying to answer on your graph?

Solution. First, we construct the above product graph J.

Then we run depth-first search starting at the node $(n, \mathbf{s})$ to find all nodes reachable from $(n, \mathbf{s})$ in J.

Finally, we return $\{m \mid (m, \mathbf{t}) \text{ was reachable from } (n, \mathbf{s})\}$.

P9.2.2 Provide an efficient algorithm to find a way to connect all members of this community with internet at minimal cost. Explain why your algorithm is correct, what the complexity of your algorithm is, and which graph representation you use.

Solution. We have $|\mathcal{N}_J| = |\mathcal{N}| \times |\mathcal{N}_Q| = \mathcal{O}(|\mathcal{N}|)$ if we assume that $|\mathcal{N}_Q|$ is a constant and we have $|\mathcal{E}_J| \leq |\mathcal{E}| \times |\mathcal{E}_Q| = \mathcal{O}(|\mathcal{E}|)$ if we assume that $|\mathcal{E}_Q|$ is a constant.

Assume both input graphs G and Q are in adjacency-list representation. Each node (u, f) in J can get a unique identifier $0 \leq \text{id}((u, f)) < |\mathcal{N}| \times |\mathcal{N}_Q|$ by assigning $\text{id}((u, f)) = \text{id}(u) \times |\mathcal{N}_Q| + \text{id}(f)$. Hence, we can easily construct an adjacency list representation of J and find corresponding nodes and edges in $\mathcal{N}$ and $\mathcal{N}_G$.

To find each outgoing edge of (u, f), we can simply inspect each combination of an outgoing edge of u and outgoing edge of f. Assuming the size of Q is constant, this takes constant time per edge.

The algorithm is correct by construction.

The depth-first search on graph J will cost worst-case $\mathcal{O}(|\mathcal{N}_J| + |\mathcal{E}_J|)$. Assuming the size of Q is constant, we have $\mathcal{O}(|\mathcal{N}_J| + |\mathcal{E}_J|) = \mathcal{O}(|\mathcal{N}| + |\mathcal{E}|)$.

10 Week 10

10.1 Material

Textbook, Section 4.4.

10.2 Additional Material

CLRS, Chapter 22 and 23.

10.3 Slides

Part 12.

10.4 Practice Exercises

General exercises. Textbook, Exercises 4.4.22, 4.4.24, 4.4.25, 4.4.26, 4.4.33, and 4.4.36. *Exercises on graph properties and properties of graph algorithms.* Textbook, Exercises 4.4.1 and 4.4.15.

10.5 Tutorial Exercises

Problem 10.1. A company has several distribution centers. To simplify logistics, the company wants to figure out which distribution center is the "most central": the maximum time it takes to transport freight from this distribution center to any other distribution center is *minimal.* Assume we know, for every pair of distribution centers X and Y, the time it takes to transport freight from X to Y.

P10.1.1 Model the above problem as a graph problem: what are the nodes and edges in your graph, do the edges have weights, and what problem are you trying to answer on your graph?

Solution. We model the input as a *weighted directed graph* $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ in which the $\mathcal{N}$ represent the distribution centers and the edges $\mathcal{E}$ represent the connections between them. Each edge (n_X, n_Y) (representing the travel from X to Y) gets weight $\mathit{weight}((n_X, n_Y))$ representing the time it takes to transport freight form X to Y.

The problem asks us to find the node (distribution center) for which the maximum length of any shortest path to any other distribution center is *minimal.*

P10.1.2 Provide an efficient algorithm to find the most central distribution center. Explain why your algorithm is correct, what the complexity of your algorithm is, and which graph representation you use.

Solution.

```
Algorithm CENTRALCENTER($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, weight) :
maxLength, node := $\infty$, none.
for $n \in \mathcal{N}$ do
    /* Invariant: maxLength is the maximum length of the shortest path from node node to any
    other node such that no node inspected by this for-loop before the current node $n$ has a lower
    maximum length. */
    Use SSSP-DIJKSTRA to compute the shortest path from $n$ to every other node in $\mathcal{G}$.
    Let $d$ be the length of the longest path just computed by SSSP-DIJKSTRA.
    if $d <$ maxLength then
        maxLength, node := $d, n$.
    end if
end for
return node.
```

The main working of the algorithm is expressed by the invariant provided. The algorithm performs SSSP-Dijkstra $|\mathcal{N}|$ times. Hence, the total complexity is $\Theta(|\mathcal{N}|(|\mathcal{N}| \log(|\mathcal{N}|) + |\mathcal{E}|))$. To represent our graph data, we use the standard adjacency list representation.

Problem 10.2. Consider we want to build McMaster Maps, the best online route planning tool in existence. The developers of McMaster Maps have decided to represent road information in terms of a massive graph in which nodes are crossing and the edges are the roads between crossings.

The developers of McMaster Maps have determined that McMaster University is the *only destination that matters*. Hence, McMaster Maps will be optimized toward computing the directions to McMaster University. To do so, McMaster Maps maintains *single-sink shortest path index* that maintains the shortest path from any node to McMaster University. This index is represented by the typical *path* and *cost* arrays as computed by either Dijkstra or Bellman-Ford.

Once in a while, an update to the road network happens: the weight of a single edge changes (this can represent adding and removing edges: addition changes the weight of the edge from ∞ to a numeric value and removing changes the weight of the edge from a numeric value to ∞).

P10.2.1 Given the road network as a graph $\mathcal{G}$, the *shortest path index*, and the edge that was changed, write an algorithm that determines whether the shortest path index is still valid. You may assume the graph is already updated. Argue why your algorithm is correct. What is the complexity of your algorithm?

Solution. Let $path[v] = w$ and $cost[v] = c$ such that w is the next node on the path from v to McMaster and c is the cost of the path from v to McMaster. We write $weight((v, w))$ to indicate the cost of edge (v, w).

Assume the edge (m, n) was changed. We have the following cases:

- The cost of edge (m, n) increases and $path[m] \neq n$. Any path going through m did not visit n and the path via n became more expensive. Hence, no shortest paths were affected. We can check whether this is the case in $\mathcal{O}(1)$.
- The cost of edge (m, n) decreases and $cost[m] \leq cost[n] + weight((m, n))$. The new path from m via n to McMaster is more expensive than the existing path from m to McMaster. Hence, any path going through m will not visit n and no shortest paths were affected. We can check whether this is the case in $\mathcal{O}(1)$.
- The cost of edge (m, n) increases and $path[m] = n$. Any path going through m did visit n and the path via n has now a changed cost associated with it. Hence shortest paths are affected and the index is no longer valid. We can check whether this is the case in $\mathcal{O}(1)$.
- The cost of edge (m, n) decreases and $cost[m] > cost[n] + weight((m, n))$. Any path going through m can now be redirected via node n and this new path will be shorted. Hence shortest paths are affected and the index is no longer valid. We can check whether this is the case in $\mathcal{O}(1)$.

P10.2.2 Assume the shortest path index is no longer valid: provide a modification of Dijkstra that restores the index to a valid state without recomputing all shortest paths. You may assume the graph is already updated. Argue why your algorithm is correct.

Solution. For now, we assume we know that the set of nodes $C \subset \mathcal{N}$ is potentially affected by the change (note C should never include the nodes n and McMaster itself). Hence, we want to recompute the shortest paths for these nodes. For each of these nodes $c \in C$, we have two options:

- either the new shortest path from c to McMaster includes edge (m, n); or
- the new shortest path from c to McMaster does *not* include edge (m, n): in this case, there must be a pair of nodes $v \in C$ and $w \notin C$ such that the shortest path from c to McMaster includes the edge (v, w).

Based on the above, we can initialize *cost*, *path*, and priority queue Q such that we can rerun Dijkstra to recompute all paths from nodes $v \in C$ to McMaster. Note that all paths from nodes $w \notin C$ to McMaster remain unchanged. Hence, we can initialize with the *potentially-shortest paths* for each node $v \in C$ by setting up:

$$cost[v] = \min_{((v,w)\in\mathcal{E})\wedge(w\notin C)} cost[w] + weight((v, w));$$

$$path[v] = ? \text{ or } w \text{ if we can choose } w \notin C \text{ with } (v, w) \in \mathcal{E} \text{ and } cost[v] = cost[w] + weight((v, w));$$

$$Q = \{v \text{ with cost } cost[v] \mid v \in C\}.$$

Next, we simply restart the main-**while** loop of Dijkstra's algorithm (which we further restrict to only consider nodes in C in any future steps). Note that this is an all-source single-target shortest path algorithm, hence, we need access to the *incoming edges* for each node instead of the outgoing edges.

To assure our algorithm is correct, we have to make sure our choice of initial nodes C is correct. We distinguish two cases:

(a) The cost of edge (m, n) increases and $path[m] = n$. Exactly those nodes whose shortest path included the edge (m, n) are affected by the change (no new nodes will consider a path via edge (m, n), as previously such paths were deemed to not be shortest paths and these paths now become even more costly). Hence, in this case, C is the set of nodes whose shortest path includes node m according to *path*.

To be able to compute C, we want to turn *path* into an adjacency list representation such that node v has edge (v, w) if $path(w) = v$. With this representation, we can simply perform DFS starting at node m to find all nodes in C.

(b) The cost of edge (m, n) decreases and $cost[m] > cost[n] + weight((m, n))$. We might be able to redirect any existing paths with a cost higher than $cost[n] + weight((m, n))$ via the edge (m, n). Hence, in this case, C is the set of nodes v with $cost[v] > cost[n] + weight((m, n))$. (We can further restrict C to those nodes that can reach m; but as this is a road network that restriction is unlikely to remove any nodes).

P10.2.3 Explain which graph representation you used for your algorithm and what the complexity of your modified-Dijkstra algorithm is using this graph representation.

Hint: Express the complexity in terms of the number of nodes affected by the change. For example, use a notation in which C is the number of nodes affected by the edge change, $incoming(C)$ the incoming edges of C, and $outgoing(C)$ the outgoing edges of C.

Solution. We use the standard representation for *path* and *cost*. For the graph, we use an adjacency list graph representation in which nodes store the *incoming* edges.

We can construct the set C in $|\mathcal{N}|$ in both cases (note that the graph represented by *paths* has at-most $|\mathcal{N}|$ edges). Our Dijkstra algorithm will visit all C nodes once and will consider each of their incoming and outgoing edges once. Hence, the total complexity is $\mathcal{O}(|\mathcal{N}| + |C| \log_2(|C|) + X)$ in which $X = |incoming(C)| + |outgoing(C)|$.

P10.2.4 What is the worst-case complexity of your solution if you use the other graph representation? Explain your answer.

Solution. To find all incoming and outgoing edges of nodes C, we have to inspect C rows and columns of an adjacency matrix of size $|\mathcal{N}| \times |\mathcal{N}|$. Hence, in this case, $X = |C| \times |\mathcal{N}|$ and the rest of the complexity remains unchanged.

11 Week 11

11.1 Material

Textbook, Chapter 4.

11.2 Slides

Part 10, 11, and 12.

11.3 Tutorial Exercises

Exercises 4.4.1, 4.4.9, 4.4.14, 4.4.16, 4.4.17, 4.4.22, 4.4.29, 4.4.35 and left-over exercises from previous weeks.

Problem 11.1. Consider a communication network represented by a graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ in which the nodes represents network devices (e.g., computers, wireless access points, switches, and routers) and the edges represent a communication channel (e.g., a network cable or a wireless connection). In this graph, every edge $(m, n) \in \mathcal{E}$ has a weight $0 \leq \mathit{weight}(m, n) \leq 1$ that indicates the probability of *failure-free delivery* when a message is sent from m to n (the reliability of the communication channel).

You may assume that the probability that a message sent by m to n is delivered without failures is independent of the rest of the network. Provide an efficient algorithm that computes the most reliable communication-path between two nodes (such that the probability of failure-free delivery is maximal and the probability of failures is minimal).

Solution. Consider the path τ via which message delivery is most reliable. We will write $P(\pi)$ to denote the probability of failure-free delivery via path π. Hence, the probability $P(\tau)$ of failure-free delivery via path τ path is the highest among all paths of the form $\pi = mv_0 \dots v_i n$. The probability $P(\pi)$ is given by

$$P(\pi) = \mathit{weight}((m, v_0)) \times \mathit{weight}((v_0, v_1)) \times \dots \times \mathit{weight}((v_i, n)).$$

As $P(\pi)$ denotes a probability, we have $0 \leq P(\pi) \leq 1$.

Finding the path $m\pi n$ for which $P(\pi)$ is the highest is equivalent to finding the path for which

$$\log(P(\pi)) = \log(\mathit{weight}((m, v_0))) + \log(\mathit{weight}((v_0, v_1))) + \dots + \log(\mathit{weight}((v_i, n)))$$

is highest. As all weights w represent probabilities, we have $0 \leq w \leq 1$. Hence, $-\infty \leq \log(w) \leq 0$. As such, finding the path $m\pi n$ for which $\log(P(\pi))$ is the highest (least-negative) is equivalent to finding the path $m\pi n$ for which

$$-\log(P(\pi)) = -\log(\mathit{weight}((m, v_0))) + -\log(\mathit{weight}((v_0, v_1))) + \dots + -\log(\mathit{weight}((v_i, n)))$$

is the lowest (least-positive).

Hence, we have turned the stated problem into a *shortest path problem* for the graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ with weights $\mathit{weight}'((v, w)) = -\log(\mathit{weight}((v, w)))$. We note that every edge weight in weight' is positive. Hence, we can use SSSP-Dijkstra to compute the shortest path from m to n in $\Theta(|\mathcal{N}| \log(|\mathcal{N}|) + |\mathcal{E}|)$. The edge weight transformations can be incorporated into SSSP-Dijkstra at no additional cost. To represent our graph data, we use the standard adjacency list representation.

Problem 11.2. Consider a company managing many servers placed all over the world. The company wants to add network connections between a minimal amount of servers to ensure that there is a path of communication between all pairs of servers.

While researching this problem, the company was advised that some connections can be built more reliable than others: according to the consulted contractors, the probability that a connection (m, n) between servers m and n will work at any given time is $p(m, n)$ (we have $p(m, n) = p(n, m)$).

The company wants to *minimize* the number of connects, while *maximizing* the probability that all servers are connected to each other at any given time. We will help the company out in their challenge to figure out which connections they need to built.

P11.2.1 Model the above problem as a graph problem: what are the nodes and edges in your graph, do the edges have weights, and what problem are you trying to answer on your graph?

Solution. The servers are nodes.

If m, n are servers (nodes), then the undirected edge (m, n) is weighted $p(m, n)$.

The problem to be answered is finding a *spanning tree* such that the *product of all edge weights* is maximized.

P11.2.2 Provide an algorithm NetworkPlan to find the network connections to build. Explain why your algorithm is correct.

Solution. We want to turn the problem into a *minimum spanning tree* problem.

First, we note that $\log(a \cdot b) = a + b$. Hence, maximizing the product $\prod_{e \in \mathcal{T}} p(e)$ of all edge weights in a spanning tree $\mathcal{T}$ is equivalent to maximizing the sum $\sum_{e \in \mathcal{T}} \log(p(e))$. Maximizing the sum $\sum_{e \in \mathcal{T}} \log(p(e))$ is equivalent to minimizing the sum $-\sum_{e \in \mathcal{T}} \log(p(e)) = \sum_{e \in \mathcal{T}} -\log(p(e))$.

Hence, we want to find a *minimum spanning tree* in the graph in which we replace the undirected edges (m, n) with weight $p(m, n)$ with undirected edges (m, n) with weight $-\log(p(m, n))$. Note that we can eliminate any edges (m, n) with $p(m, n) = 0$.

Now we can simply solve the minimum spanning tree problem to find the edges we are interested in. We use Prim's Algorithm (using a Fibonacci heap) to compute the minimum spanning tree.

P11.2.3 Explain which graph representation you used for your algorithm and what the complexity of your algorithm is using this graph representation.

Solution. We choose the matrix representation as we have an extremely dense graph: we have $|\mathcal{E}| = |\mathcal{N}|(|\mathcal{N}|-1) = \Theta(|\mathcal{N}|^2)$. In this case, Prim's algorithm has a complexity of $\mathcal{O}(|\mathcal{N}| \log_2(|\mathcal{N}|) + |\mathcal{E}|) = \Theta(|\mathcal{N}|^2)$ *if* we use a Fibonacci heap. In this case, Prim's Algorithm has a lower complexity than Kruskal, making it the best choice.

Note If Kruskal is used or Prim without a Fibonacci heap, then the complexity must be $\Theta(|\mathcal{E}| \log_2(|\mathcal{E}|))$ or $\Theta(|\mathcal{E}| \log_2(|\mathcal{N}|))$ or $\Theta(|\mathcal{N}|^2 \log_2(|\mathcal{N}|))$ or something along those lines.

P11.2.4 What is the worst-case complexity of your solution if you use the other graph representation? Explain your answer.

Solution. Identical, as we have $|\mathcal{E}| = \Theta(|\mathcal{N}|^2)$.

12 Week 12

12.1 Material

Textbook, Section 5.1, 5.2, 5.3, and 5.4.

12.2 Slides

Part 13.

12.3 Tutorial Exercises

To be determined.

Problem 12.1. Consider a list L of $N = |L|$ integers with values between 0 and k. We want to perform *range-queries* on these integers: given a lower-bound a and upper-bound b, we want to determine how many integers are in the range $[a, b]$. (Hence, we want to determine $|\{v \in L \mid a \leq v \leq b\}|$).

We want to answer these *range-queries* in constant time ($\mathcal{O}(1)$). To enable this, you are allowed to preprocess the input L *once* to turn it into a more-efficient representation *before* we run range queries. This preprocessing step may cost $\mathcal{O}(N + k)$ in the worst case.

P12.1.1 Describe the data representation you choose and provide a range-query algorithm that uses this more-efficient data representation to answer range-queries in constant time. Argue why your range-query algorithm is correct and has a complexity of $\mathcal{O}(1)$.

Solution. We make an "index" of size k: a list $I[0 \dots k+1)$ such that $I[x]$ stores the number of elements smaller-or-equal to x in L (hence, $I[x] = |\{v \in L \mid v \leq x\}|$). Given this data representation, we can answer the range query for $[a, b]$ by returning $I[b] - I[a-1]$.

The complexity of this range query is $\Theta(1)$, as it performs only four computations (compute $a - 1$, look up two values in I, and compute their difference). The range query is correct, as $I[b]$ counts all elements smaller-or-equal to b, including those that are *not* at-least a, and $I[a-1]$ counts all elements smaller than a (those elements that are *not* at-least a).

P12.1.2 Provide an algorithm to compute your more-efficient data representation in $\mathcal{O}(N + k)$. Argue why your algorithm is correct and has a complexity of $\mathcal{O}(N + k)$.

Solution. We construct the index I using the following algorithm.

```
Algorithm CreateIndex($L$, $k$) :
Sort list $L$ using BucketSort with $k$ buckets.
$I := [0 \mid 0 \leq i \leq k]$.
$i := 0$.
for $j := 0$ upto $|L|$ (excluding $|L|$) do
    /* Invariant: $I[x] = |\{v \in L[0 \dots j) \mid v \leq x\}|, x \leq i$. */
    while $i < L[j]$ do
        $I[i+1], i := I[i], i+1$.
    end while
    $I[i] := I[i+1] + 1$.
end for
$L[y] := L[i]$ for all $i < y \leq k$.
```

The algorithm computes the index from small-to-large by observing that the count $I[x+1]$ is equal to the count $I[x]$ *plus* the number of values equal to $x + 1$. The BucketSort has a complexity of $\Theta(N + k)$. Making the list I has a complexity of $\Theta(k)$ and filling the list I (the **for**-loop and the last line) has a complexity of $\Theta(k + N)$ (j will iterate over every index in L once and i will iterate over every index in I once).

Problem 12.2. Consider the wildcard symbol '•' which we only use to indicate zero-or-more arbitrary symbols.

For example, the pattern "hello•world" matches "helloworld" (the wildcard • interpreted as zero symbols), "hello world" (the wildcard • interpreted as the whitespace symbol), and "hello beautifull world" (the wildcard • interpreted as the string " beautifull ").

P12.2.1 Write an algorithm that, given a source string $\mathcal{S}$ and pattern P, returns the *first* offset in $\mathcal{S}$ at which pattern P begins (if there is such an offset, otherwise return "not found"). Your algorithm should run in $\mathcal{O}(|S|)$.

Solution. Consider a pattern $P = w_1 \bullet w_2 \bullet \cdots \bullet w_n$. Hence, this pattern requires us to search for the sequence of strings $w_1, w_2, \dots, w_n$ (each string in the sequence separated from the others by zero-or-more symbols).

Consider a string S to search in. If the first occurance of string w_1 in S is at offset i, then—to match pattern P—we must try to find w_2 starting at offset $i + |w_1|$ (starting directly after the first occurance of w_1). Hence, the following algorithm will be able to find the first offset of P in S:

```
Algorithm WildcardSearch($S$, $P = w_1 \bullet w_2 \bullet \cdots \bullet w_n$) :
i, r := 0, "not found".
for $j$ := 1 upto $n$ (including $n$) do
    Search for the first offset of $w_j$ in $S[i \dots |S|)$.
    if an offset $k$ was found then
        $i := k + |w_j|$.
        if $j = 1$ then
            $r := k$ (offset of first word of $P$ in $S$).
        end if
    else
        return "not found".
    end if
end for
return $r$.
```

To end up with the claimed complexity, we need a fast string search algorithm. In this case, we can use the Knuth-Morris-Pratt algorithm.

P12.2.2 Explain why your algorithm is correct and runs in $\mathcal{O}(|S|)$.

Solution. The correctness of our algorithm follows from the correctness of KMP. The Knuth-Morris-Pratt (KMP) algorithm has a complexity of $\Theta(|S| + |w|)$ to search for a word w in string S. Here, we note that if the number of symbols in $|P|$ is larger than $|S|$, then the pattern does not fit and we do not have to search. Hence we can always assume $|P|$ is smaller than $|S|$. Furthermore, we notice that the searches for $w_1, w_2, \dots, w_n$ will each inspect non-overlapping parts of the input string S. Hence, we traverse S at-most once during all n uses of KMP.

Bonus Technically, we can have very long patterns P by repeating wildcard symbols, e.g., $P = w_1 \bullet\bullet\bullet\bullet\bullet\bullet\bullet w_2$. In that case, the above solution to bound $\Theta(|P|)$ by $\Theta(|S|)$ no longer works. Such patterns are equivalent to patterns without wildcard-repetition, however. Hence, we can assume that no wildcards are repeated. Students that recognized this limitation one way or another in their solution have a correct solution.