Exercise Session 01

Solve the following exercises. The exercises that are more involved are marked with a star. If you need some guidance for solving the exercise, place your trash bin in front of your group room's door and the first available teaching assistant will come to help you out.

Exercise 1.
Consider the problem of finding the two smallest numbers in a nonempty sequence of numbers $\langle a_1, \ldots, a_n \rangle$ not necessarily sorted.

(a) Formalise the above as a computational problem (be careful to precisely define the input, the output, and their relationship).

(b) Write the pseudocode of an algorithm that solves the above computational problem assuming the sequence is given as an array $A[1\,..\,n]$.

(c) Assume that line i of your pseudocode takes constant time c_i to execute. What is the worst case running time of your algorithm?

Solution 1.

(a) In this exercise it is very important to define what the output must be, when there are numbers occurring more than once in the sequence. In the following, we define the output so that adding duplicates of any elements in the array does not change the answer.

Input: A sequence of $n > 0$ numbers $\langle a_1, \ldots, a_n \rangle$.

Output: A pair (min_1, min_2) such that $min_1 < min_2$ and

$$min_1 = \min\{a_i \colon 1 \leq i \leq n\} \text{ and,}$$
$$min_2 = \min\{a_i \colon 1 \leq i \leq n \text{ and } min_1 \neq a_i\}$$

We will use the convention that $\min \emptyset = \infty$ to cover the case when no min_2 exists, which occurs when all values in the sequence are equal.

(b) Consider the following function FIND-2SMALLEST which takes an array $A[1\,..\,n]$ with $n \geq 1$.

```
FIND-2SMALLEST(A)
1  min1 = A[1]
2  min2 = ∞
3  for i = 2 to A.length
4       if A[i] < min1
5            min2 = min1
6            min1 = A[i]
7       elseif min1 < A[i] < min2
8            min2 = A[i]
9  return (min1, min2)
```

Please remember that there are many possible correct solutions, therefore if your own solution is different it might be still correct. We will discuss in the following lectures different techniques to prove correctness for an algorithm. Intuitively, the above algorithm maintains the property that $min_1 < min_2$ during the whole execution of the algorithm moreover it ensures that the following properties $min_1 = \min\{a_j \colon 1 \leq j \leq i\}$ and $min_2 = \min\{a_j \colon 1 \leq j \leq i \text{ and } min_1 \neq a_j\}$ before each iteration of the for loop. Therefore, when $i = A.length = n$ the property required for the output to be correct hods true.

(c) Let n be the size of the array A and c_i $(i = 1..9)$ the time it takes to execute the statement i in FIND-2SMALLEST. The worst case occurs when the condition $A[i] < min_1$ holds true for each iteration of the for loop. This happens for instance when the sequence of elements is (strictly) increasing, that is $a_1 > a_2 > \cdots > a_n$. In this case the lines **??** and **??** are executed n times, leading to the following worst-case runtime for FIND-2SMALLEST:

$$\begin{aligned} T(n) &= c_3(n+1) + (c_4 + c_5 + c_6)n + c_1 + c_2 + c_9 \\ &= (c_3 + c_4 + c_5 + c_6)n + c_1 + c_2 + c_3 + c_9 \end{aligned}$$

Notice that the above is a linear function in n (i.e., of the form $an + b$ for some constants a and b depending on c_i $(i = 1, \ldots, 9)$).

Exercise 2.

Consider the following pseudocode for the function FIND-ELEMENT takes as input an array $A[1..n]$ and a number a.

```
FIND-ELEMENT(A, a)
1  for i = 1 to A.length
2        if A[i] = a
3              return i
4  return i
```

(a) Count the number of iterations of the for loop in the above pseudocode for the execution of FIND-ELEMENT$([1, 0, 5, 2, 4], 5)$ and report the value returned at the end of the call.

(b) Is the above algorithm solving the element search problem presented in class? Justify your answer.

Solution 2.

(a) When executing FIND-ELEMENT$([1, 0, 5, 2, 4], 5)$ the for loop is executed 3 times because the element 5 is first found when the index counter i holds the value 3. Thus, the value returned by the call is 3.

(b) Recall that for an algorithm to be correct, we require that for any valid input instance it halts with the correct output. This is not the case for FIND-ELEMENT, since for example the call FIND-ELEMENT$([1], 3)$ returns 1 instead of 0.

Exercise 3.

Write a pseudocode of an algorithm to reverse an array of numbers, i.e., the last element should become the first, the second last should become the second, etc. For example, the reverse of $[1, 2, 3, 4]$ is $[4, 3, 2, 1]$.

(a) What is the worst-case running time of your algorithm?

(b) Try now to propose an algorithm that use only a constant amount of extra space. What is the worst-case running time of your algorithm?

Solution 3.

We solve only the point (b) because its solution works in particular as a solution for (a). The algorithm is described below.

```
REVERSE(A)
1  for i = 1 to ⌊A.length / 2⌋
2        key = A[i]
3        A[i] = A[A.length + 1 − i]
4        A[A.length + 1 − i] = key
```

The procedure REVERSE swaps the i-th element from the start of the array with the i-th element from the end, until its reached the index in the middle. Let c_i $(i = 1\,..\,4)$ the cost of executing the i-th statement, then the worst-case running time of the procedure REVERSE is

$$T(n) = c_1(\lfloor n/2 \rfloor + 1) + (c_2 + c_3 + c_4)\lfloor n/2 \rfloor\,.$$

Exercise 4.
Look at the table in Exercise 1-1 in the textbook. Fill in the columns for 1 second and 1 minute. *Hint:* 1 microsecond $= 10^{-6}$ seconds.

★ **Exercise 5.**
Let A and B be two arrays of numbers sorted in non-decreasing order, respectively of length n and m. Write the pseudocode of an algorithm that checks whether the set of elements in A is equal to the set of elements in B, i.e., all elements of A are contained in B and vice versa. What is the worst-case running time of your algorithm?

Solution 5.
We begin by formalising the corresponding computational problem

Input: Two sequences of numbers $\langle a_1, \ldots, a_n\rangle$ and $\langle b_1, \ldots, b_m\rangle$ sorted in non-decreasing order i.e., $a_1 \leq a_2 \leq \cdots a_n$ and $b_1 \leq b_2 \leq \cdots b_m$.

Output: *true* if and only if $\{a_i \colon 1 \leq i \leq n\} = \{b_i \colon 1 \leq i \leq m\}$.

For our algorithm we will use the usual respresentation of sequences by means to two arrays $A[1\,..\,n]$ and $B[1\,..\,m]$. Before start writing the pseudocode, we shall note that n and m may not be equal, and even if $n = m$, it does not suffice to check whether $A[i] = B[i]$ for all $1 \leq i \leq n$ since A and B may have repeated occurrences of the same numbers which are not necessarily positioned in the same index, e.g., $A = [1, 1, 2]$ and $B = [1, 2, 2]$ have the same set of elements but $A[2] = 1 \neq 2 = B[2]$.

```
EQUALSORTEDSETS(A, B)
 1  i = 1
 2  j = 1
 3  while i ≤ A.length and j ≤ B.length and A[i] = B[j]
 4       while i < A.length and A[i] = A[i + 1]
 5            i = i + 1
 6       while j < B.length and B[j] = B[j + 1]
 7            j = j + 1
 8       i = i + 1
 9       j = j + 1
10  return i > A.length and j > B.length
```

The algorithm stores two indices i and j which point at the first occurrence on an element in the arrays A and B respectively. This property is maintained at right before each iteration of the outer while loop, by the two inner loops. Their job is move the indices forward until the last occurrence of an element is found. If the outer while loop terminates because both the indices exceeded the length of the corresponding array, then the two arrays represent the same set of numbers, otherwise one of the two arrays has an element which the other doesn't.

Before diving into the worst-case analysis, we observe that the size of the input depends both on m and n. Therefore we express the running time of EQUALSORTEDSETS as a function in two arguments, i.e., $T(n, m)$.

The worst-case running time occurs when the two arrays are equal, i.e.,

$$\{a_i \colon 1 \leq i \leq n\} = \{v_1, \ldots, v_k\} = \{b_i \colon 1 \leq i \leq m\}$$

for some (pairwise distinct) numbers $v_1\,..\,v_k$. In this case, the number of iterations of the outer loop corresponds to k. We denote respectively by $A(v_i)$ and $B(v_i)$ the number of occurrences of the element

v_i $(i = 1..k)$ in the arrays A and B. For each iteration of the outer loop, the two inner loops perform a number of iterations that corresponds to the amount of occurrences of the current element in each array.

$$\begin{aligned} T(n,m) =& c_1 + c_2 + (k+1)c_3 + c_4\left(\sum_{i=1}^{k} A(v_i) + 1\right) + c_5\sum_{i=1}^{k} A(v_i) + \\ &+ c_6\left(\sum_{i=1}^{k} B(v_i) + 1\right) + c_7\sum_{i=1}^{k} B(v_i) + (c_8 + c_9)k + c_{10} \end{aligned}$$

note that $\sum_{i=1}^{k} A(v_i) = n$ and $\sum_{i=1}^{k} B(v_i) = m$, hence

$$= (c_4 + c_5)n + (c_6 + c_7)m + (c_3 + c_8 + c_9)k + \left(\sum_{i=1}^{4} c_i\right) + c_6 + c_{10}$$

At this point the formula still depends on k. The value of k is maximal when one of the two arrays has no repeated elements. In this case $k = \min\{n, m\}$ and the formula simplifies as follows.

$$= (c_4 + c_5)n + (c_6 + c_7)m + (c_3 + c_8 + c_9)\min\{n, m\} + \left(\sum_{i=1}^{4} c_i\right) + c_6 + c_{10}\,.$$

Exercise Session 02

Solve the following exercises. The exercises that are more involved are marked with a star. If you need some guidance for solving the exercise, place your trash bin in front of your group room's door and the first available teaching assistant will come to help you out.

Exercise 1.
Solve exercises CLRS 3.1–1, and 3.1–4

Solution 1.

CLRS 3.1–1 Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions —we say that $f(n)$ is asymptotically nonnegative if there exists n_0 such that $f(n) \geq 0$ for all $n \geq n_0$—. Using the basic definition of Θ notation, prove that $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

We have to prove that there exists $c_1, c_2 > 0$, and n_0 such that $0 \leq c_1(f(n) + g(n)) \leq \max(f(n), g(n)) \leq c_2(f(n) + g(n))$ for all $n \geq n_0$. We know that $f(n)$ and $g(n)$ are asymptotically nonnegative, then we can choose $n_0 > 0$ such that $f(n) \geq 0$ and $g(n) \geq 0$ for all $n \geq n_0$. Let $c_1 = 1/2$, $c_2 = 1$, and $n \geq n_0$. On the one hand, we have that $\max(f(n), g(n)) \leq f(n) + g(n)$ because the max of two nonnegative numbers is always smaller than their sum. On the other hand, we have $1/2(f(n) + g(n)) \leq \max(f(n), g(n))$ because the average of two numbers is always less than or equal to the maximum of the two.

CLRS 3.1–4 Is $2^{n+1} = O(2^n)$? Yes, because $2^{n+1} = 2 \cdot 2^n = O(2^n)$.

Is $2^{2n} = O(2^n)$? We show that $2^{2n} \neq O(2^n)$. Towards a contradiction, assume that $2^{2n} = O(2^n)$. By definition of big-O notation, there exist $c > 0$ and $n_0 > 0$ such that $0 \leq 2^{2n} \leq c2^n$ for all $n \geq n_0$. Since $2^{2n} = 2^n \cdot 2^n$ we must have $2^n \leq c$ for all $n > n_0$. This cannot be true because c is a constant and 2^n diverges for n that goes to infinity. The correct asymptotic class for 2^{2n} is $2^{2n} = (2^2)^n = O(4^n)$.

~~**Exercise 2.**~~
~~Consider the algorithm SumUpTo that takes as input a natural number $n \in \mathbb{N}$.~~

~~SumUpTo(n)~~
~~1 $s = 0$~~
~~2 **for** $i = 1$ **to** n~~
~~3 $s = s + i$~~
~~4 **return** s~~

~~Prove that given $n \in \mathbb{N}$, SumUpTo terminates and returns $\frac{n(n+1)}{2}$.~~

~~**Solution 2.**~~
~~We prove the following loop invariant: Before each iteration of the outer loop $s = \frac{(i-1)i}{2}$.~~

~~**Initialisation:** Before the first iteration $i = 1$ and $s = 0$. Substituting i with 1 we obtain $\frac{(i-1)i}{2} = 0$.~~

~~**Maintenance:** Before executing line 3 we have that $s = \frac{(i-1)i}{2}$, thus after the assignment we have~~

$$s = \frac{(i-1)i}{2} + i = \frac{i^2 - i + 2i}{2} = \frac{i^2 + i}{2} = \frac{i(i+1)}{2}.$$

~~Therefore, incrementing i for the next iteration preserves the invariant.~~

~~**Termination:** The loop terminates when $i > n$, that is when $i = n + 1$. By substituting i with $n + 1$ in the invariant, we obtain $s = \frac{n(n+1)}{2}$.~~

~~Therefore the function SumUpTo returns $\frac{n(n+1)}{2}$.~~

Exercise 3.
By getting rid of the asymptotically insignificant parts on the expressions, give a simplified asymptotic tight bounds (big-theta notation) for the following functions in n. Here, $k \geq 1$, $e > 0$ and $c > 1$ are constants.

(a) $0.001n^2 + 70000n$

(b) $2^n + n^{10000}$

(c) $n^k + c^n$

(d) $\log^k n + n^e$

(e) $2^n + 2^{n/2}$

(f) $n^{\log c} + c^{\log n}$ (hint: look at some properties of the logarithm at page 56 in CLRS)

Solution 3.

(a) $0.001n^2 + 70000n = \Theta(n^2)$

(b) $2^n + n^{10000} = \Theta(2^n)$

(c) $n^k + c^n = \Theta(c^n)$, because any exponential function dominates any polynomial functions.

(d) $\log^k n + n^e = \Theta(n^e)$ Look at page 57 of CLRS for the comparison of polylogarithmic and polynomial functions.

(e) $2^n + 2^{n/2} = \Theta(2^n)$. Here note that $2^{n/2} = (\sqrt{2})^n$ and $2 > \sqrt{2} \cong 1.4$ therefore the first term dominates over the second.

(f) $n^{\log c} + c^{\log n} = 2 \cdot n^{\log c} = \Theta(n^{\log c})$ because $n^{\log c} = c^{\log n}$. For the same reason we also have $n^{\log c} + c^{\log n} = 2 \cdot c^{\log n} = \Theta(c^{\log n})$.

★ **Exercise 4.**
Consider the following algorithm that takes an array $A[1 \ldots n]$ and rearrange its elements in nondecreasing order.

```
SORT(A)
1  for i = 1 to A.length
2      for j = i + 1 to A.length
3          if A[i] > A[j]
4              key = A[i]
5              A[i] = A[j]
6              A[j] = key
```

(a) Explain informally how the algorithms works;

(b) Prove that Sort solves the sorting problem (hint: determine suitable invariants for both loops);

(c) Determine the asymptotic worst-case running time using the Θ notation.

Solution 4.

(a) Informally, the algorithm may be explained using the analogy of sorting a hand of playing cards. We start with an empty left hand and the other cards face up on the table. Then we select the smallest card in the table an insert it as the rightmost card in the left hand. To find the smallest card in the table we select the leftmost card in the table and we proceed to the right. Every time we encounter a card which is smaller than the first card in the table, we swap the two and continue this way until all cards in the table have been checked. The sub-array $A[1\,..\,i-1]$ constitutes that sorted hand, the sub-array $A[i\,..\,n]$ corresponds to the pile of cards still on the table.

(b) Invariant of the inner for-loop: $A[i]$ is the smallest element of the sub-array $A[i\,..\,j-1]$ and $A[i\,..\,j-1]$ is a reordering of the elements that where originally in $A[i\,..\,j-1]$ before executing the loop.

Initialisation: We show that the loop invariant holds true before the first loop iteration. Before running the inner loop $j = i+1$, therefore $A[i\,..\,j-1] = A[i]$ which is in fact the smallest element and it is its original position before executing the loop.

Maintenance: By hypothesis we know that $A[i]$ is the smallest element of the sub-array $A[i\,..\,j-1]$. If $A[j] < A[i]$ the two elements are swapped. At this point $A[i]$ contains the smallest element in the sub-array $A[i\,..\,j]$, and $A[i\,..\,j]$ is a reordering of the elements that were originally in $A[i\,..\,j-1]$ before executing the loop. Incrementing j for the next iteration preserves the loop invariant.

Termination: The for-loop terminates when $j > A.length$, that is $j = n+1$. By substituting $n+1$ for j in the invariant we obtain that $A[i]$ is the smallest element of the sub-array $A[i\,..\,n]$ and $A[i\,..\,n]$ is a reordering of the elements that where originally in $A[i\,..\,n]$ before executing the loop.

Now that we have proved the above invariant for the inner loop, we can use it to prove the next invariant relative to the outer loop. Before each iteration of the outer loop the sub-array $A[1\,..\,i-1]$ is sorted, all the elements of the sub-array $A[i\,..\,n]$ are greater than or equal to any element of $A[1\,..\,i-1]$, and A is a reordering of the elements that were originally in the array A.

Initialisation: Before the first outer loop iteration $i = 1$, therefore $A[i\,..\,i-1] = []$ and $A[i\,..\,n] = A[1\,..\,n]$. Therefore $A[i\,..\,i-1]$ is trivially sorted and the fact that for all element in $a \in A[i\,..\,i-1]$ and all element in $b \in A[i\,..\,n]$, $a \leq b$ trivially holds because there is no element a to compare. Moreover, A at this point has never been modified.

Maintenance: By hypothesis we know that $A[1\,..\,i-1]$ is sorted and all the elements in $A[i\,..\,n]$ are greater than or equal to any other element of $A[1\,..\,i-1]$. As proven above, at the termination of the inner loop $A[i]$ is the smallest element of the sub-array $A[i\,..\,n]$ and $A[i\,..\,n]$ is a reordering of the elements that where originally in $A[i\,..\,n]$ before executing the loop. Hence, incrementing i for the next iteration preserves the invariant.

Termination: The outer for loop terminates when $i > A.length$, that is $i = n+1$ and the invariant reads as follows. The sub-array $A[1\,..\,n]$ is sorted and it is a reordering of the elements that were originally in the array A.

This concludes the proof of correctness for the procedure SORT.

(c) The worst-case running time occurs when the body of the if-then inside the inner for loop is always executed. This occurs when the array $A[1\,..\,n]$ is such that $A[1] > A[2] > \cdots > A[n]$.

Let c_1 and c_2 be the time it takes for executing line 1 and 2 respectively, and $c_{3\text{-}6}$ the time it takes for executing the entire if construct (lines 3–6). Moreover, we denote by t_i be the number of times the condition of the inner for loop is executed during the i-th iteration of the outer

loop. Then the worst case running time of SORT is

$$\begin{aligned} T(n) &= c_1(n+1) + c_2 \sum_{i=1}^{n} t_i + c_{3\text{-}6} \sum_{i=1}^{n} (t_i - 1) \\ &= c_1(n+1) + c_2 \frac{n(n+1)}{2} + c_{3\text{-}6} \frac{n(n-1)}{2} && (t_i = n - (i-1)) \\ &= \Theta(n^2) \end{aligned}$$

Loop Invariant Extra Exercises

Jim Glenn

January 21, 2020

Problem 1 Prove with a loop invariant that the the following function correctly sums the elements in the array passed to it.

```
def sum(A, n):
    i = 0
    sum = 0
    while i < n:
        sum = sum + A[i]
        i = i + 1
    return sum
```

Preconditions:

- A is an array of real numbers (indexed starting from 0),
- n is the size of the array A (so n is an integer such that $n \geq 0$)

Postcondition: $sum = \sum_{k=0}^{n-1} A[k]$

Solution We use the following invariant:

(a) $sum = \sum_{k=0}^{i-1} A[k]$ (that is, sum is the sum of the first i elements of A)

(b) i is an integer such that $0 \leq i \leq n$

Basis/Initialization:

(a) sum and i are initialized to 0 and with $i = 0$, $\sum_{k=0}^{i-1} A[k] = 0$ because the sum is empty, so $sum = \sum_{k=0}^{i-1} A[k]$ since both are equal to 0.

(b) i is initially 0, which is an integer, and $0 \leq 0 = i$. Also, $i = 0 \leq n$ by the precondition on n. Putting those together gives $0 \leq i \leq n$.

Induction: Suppose the invariant is true before one iteration of the loop and the guard $i < n$ is true.

(a) Since the invariant is true before the loop, we have $sum_{old} = \sum_{k=0}^{i_{old}-1} A[k]$. The first statement inside the loop sets $sum_{new} = sum_{old} + A[i_{old}] = \sum_{k=0}^{i_{old}-1} A[k] + A[i_{old}] = \sum_{k=0}^{i_{old}} A[k]$. The second statement sets $i_{new} = i_{old} + 1$. Substituting that into the result of the first statement gives $sum_{new} = \sum_{k=0}^{i_{new}-1} A[k]$, which is part (a) of the invariant with the new values of the variables.

(b) By the invariant, i_{old} is an integer such that $0 \leq i_{old} \leq n$. Since the guard is true, $i_{old} < n$, which is equivalent to $i_{old} \leq n-1$ since i and n are integers. So $0 \leq i_{old} \leq n-1$ and $0 \leq i_{old} < i_{old}+1 \leq n$. Since the second statement in the loop sets $i_{new} = i_{old}+1$, we can rewrite that as $0 \leq i_{new} \leq n$, which is part (b) of the invariant with the new values of the variables.

Termination: i increases each time through the loop, so eventually $i \geq n$ and the guard becomes false to terminate the loop.

Postcondition: The falsity of the guard when the loop terminates means that $i \geq n$. Part (b) of the invariant says $i \leq n$. The only way $i \geq n$ and $i \leq n$ can both be true is if $i = n$. Plug that into part (a) of the invariant yields $sum = \sum_{k=0}^{n-1} A[k]$, which is the required postcondition for the function.

Problem 2 Prove that insertion sort is correct, using the invariant from class.

```
InsertionSort(A, n)
  i = 1
  while i < n
    insert A[i] into correct location among A[0], ..., A[i-1]
    i = i + 1
```

The "insert..." line is either an inner loop or, as we will treat it, a call to a function `insert(A, i)` that, given an array with the first i elements in sorted order, modifies A so that its first $i+1$ elements are unchanged but are reordered to be sorted, and does not change any of the other elements in A (so `insert` has preconditions 1) A is a non-empty array of real numbers, 2) $0 \leq i < len(A)$, and 3) $A[0] \leq \cdots \leq A[i-1]$; and postconditions 1) $A[0] \leq \cdots \leq A[i]$, 2) those 1st $i+1$ elements are unchanged but may be reordered, and 3) the elements after them are unchanged). (Fun exercise: write `insert` and prove that it is correct using a loop invariant. This is how we tackle nested loops: treat the innermost loop as a separate function, prove that it does what it is supposed to do, and then work on the next outer loop.)

Preconditions:

- A is a non-empty array of real numbers
- n is the size of A (so n is an integer such that $n \geq 1$)

Postcondition: $A[0] \leq \cdots \leq A[n-1]$ and A has the same elements as before (but possibly in a different order).

Invariant:

(a) $A[0] \leq A[1] \leq \cdots \leq A[i-1]$

(b) $A[0], \ldots, A[i-1]$, are the original first i elements of A, possibly in a different order

(c) $A[i], \ldots, A[n]$ contain their original values

(d) i is an integer such that $1 \leq i \leq n$

Solution Basis:

(a) i is initialized to 1, and when $i = 1$ part (a) is vacuously true

(b) i is initialized to 1 and there are no assignments to A before the loop, so $A[0]$ has its original value, which is part (b) of the invariant when $i = 1$.

(c) There are no assignments to A before the loop, so $A[1], \ldots, A[n-1]$ all have their original values, which is part (c) of the invariant when $i = 1$.

(d) i is initialized to 1, which is an integer. The precondition on n yields $n \geq 1$. So $i = 1 \leq n$. Relaxing $1 = i$ to $1 \leq i$ and combining with $i \leq n$ yields $1 \leq i \leq n$.

Induction: Suppose the invariant is true before an iteration of the loop and that the guard $i < n$ is also true.

(a) By part (a) of the invariant, $A[0]_{old} \leq \cdots \leq A[i_{old} - 1]_{old}$, and by part (d) and the guard, $0 \leq i_{old} < n$. Those are the preconditions for the call `insert(A, i)`, so after that call we have $A[0]_{new} \leq \cdots \leq A[i_{old}]_{new}$ by the postconditions of `insert`. By the last statement in the loop, we have $i_{new} = i_{old} + 1$, so we can rewrite the previous results as $A[0]_{new} \leq \cdots \leq A[i_{new} - 1]_{new}$, which is part (a) of the invariant using the new values of the variables.

(b) By part (b) of the invariant, $A[0]_{old}, \ldots, A[i_{old}-1]_{old}$ are the original first i_{old} values from A in some order. From part (c), $A[i_{old}]_{old}$ is its original value. Now the postcondition of `insert` says those elements are reordered but not changed, so the new values are the original first $i_{old} + 1 = i_{new}$ values from A in some order, which is part (b) of the invariant with the new values of the variables.

(c) By part (c) of the invariant, $A[i_{old}+1], \ldots, A[n]$ are their original values. There are no assignments to them in the loop, and the postcondition of `insert` guarantees that they are not changed. So they still hold their original values. Rewriting using $i_{new} = i_{old}+1$ yields $A[i_{new}], \ldots, A[n]$ have their original values, which is part (c) of the invariant using the new values of the variables.

(d) $0 \leq i_{old} \leq n$ by part (d) of the invariant. $i_{old} < n$ since the guard is true and hence $i_{old} \leq n-1$ since i and n are integers (by the part (d) of the invariant and the precondition on n respectively). Therefore, $0 \leq i_{old} < i_{old}+1 \leq n$. Since $i_{new} = i_{old}+1$, we can rewrite that as $0 \leq i_{new} \leq n$, which is part (d) of the invariant with the new values of the variables.

Termination: i increases each time through the loop, so eventually we have $i \geq n$, which breaks the loop.

Postconditions: When the loop terminates, we have $i \geq n$ from the guard being false, and $i \leq n$ from part (d) of the invariant, so $i = n$ at termination. Substituting n for i into parts (a) and (b) of the invariant then yields $A[0] \leq \cdots \leq A[n-1]$ and $A[0], \ldots, A[n-1]$ are their original values, reordered, which are the postconditions for sorting A.

Now that we've done some work with loop invariants including conditions on the loop counters, let's cut that tedious part out. If there is a variable x that is initialized to $x = c_1$ before the loop, the guard on the loop is $x < c_2$ for some $c_2 >= c_1$, c_1 and c_2 are integers, and the only assignment to x inside the loop is $x = x + 1$ (which is not in any other inner loop or conditional), then we may conclude that x is always an integer such that $c_1 \leq x \leq c_2$, and that the loop terminates when $x = c_2$. (Corollary: under the same conditions, but with guard $P \wedge x < c_2$, then the conditions on x still hold, but at the termination of the loop all we know is $\neg P \vee x = c_2$).

Exercise Session 03

Solve the following exercises.

Exercise 1.
Run the merge sort algorithm on the following array of numbers: $[3, 41, 52, 26, 38, 57, 49, 9]$. Give the state of the array after five calls of the algorithm MERGE are performed during the execution of MERGE-SORT.

Solution 1.
The state of the array after five calls to MERGE are performed in MERGE-SORT is $[3, 26, 41, 52, 38, 57, 9, 49]$. One can verify this by performing a step-by-step unfolding of the recursion tree as done in class.

Exercise 2.
Solve exercise CLRS 2.3–3, 2.3–4, and 2.3–6.

Solution 2.
CLRS 2.3–3. Consider the following recurrence for $n = 2^k$ and $k \in \mathbb{N} \setminus \{0\}$.

$$T(n) = \begin{cases} 2 & \text{if } n = 2 \\ 2T(n/2) + n & \text{if } n = 2^k \text{ for } k > 1 \end{cases}$$

We prove that $T(n) = n \lg n$ by induction on k.

Base Case ($k = 1$). For $k = 1$ we have that $n = 2^1 = 2$. By definition of T we have $T(2) = 2$ which is equals to $2 \lg 2 = n \lg n$.

Inductive Step ($k > 1$). Then since $n = 2^k$ we have that $n/2 = 2^{k-1}$. Hence we have

$$\begin{aligned} T(n) &= 2T(n/2) + n && \text{(by def. } T\text{)} \\ &= 2(n/2 \lg(n/2)) + n && \text{(by inductive hypothesis)} \\ &= n \lg(n/2) + n \\ &= n(\lg n - \lg 2) + n && \text{(recall that } \lg(a/b) = \log a - \lg b\text{)} \\ &= n(\lg n - 1) + n && \text{(recall that } \lg 2 = 1\text{)} \\ &= n \lg n - n + n \\ &= n \lg n \end{aligned}$$

This concludes the proof.

Another way to solve the same exercise is the following. Let $n = 2^k$ for some $k \in \mathbb{N} \setminus \{0\}$, we can rewrite the recurrence in terms of k as

$$T(2^k) = \begin{cases} 2 & \text{if } k = 1 \\ 2T(2^{k-1}) + 2^k & \text{if } k > 1 \end{cases}$$

Analogously, the equality $T(n) = n \lg n$, is equivalently restated as $T(2^k) = 2^k \lg 2^k = k \cdot 2^k$. We prove by induction on k that $T(2^k) = k \cdot 2^k$.

Base Case ($k = 1$). Then, $k \cdot 2^k = 2 = T(2^k)$.

Inductive Step ($k > 1$).

$$\begin{aligned} T(2^k) &= 2T(2^{k-1}) + 2^k && \text{(by def. } T\text{)} \\ &= 2((k-1) \cdot 2^{k-1}) + 2^k && \text{(by inductive hypothesis)} \\ &= 2(k \cdot 2^{k-1} - 2^{k-1}) + 2^k \\ &= k \cdot 2^k - 2^k + 2^k \\ &= k \cdot 2^k \end{aligned}$$

CLRS 2.3–4. The recursive variant of the insertion sort algorithm is described in the following pseudocode. The algorithm takes and array $A[1\,..\,n]$, an index $1 \leq p \leq n$, and sorts the subarray $A[1\,..\,p]$.

```
InsertionSort(A, p)
1  if p > 1
2      InsertionSort(A, p − 1)
3      // Insert A[p] into the sorted sequence A[1 .. p − 1]
4      key = A[p]
5      i = p − 1
6      while i > 0 and A[i] > key
7          A[i + 1] = A[i]
8          i = i − 1
9      A[i + 1] = key
```

In this case a meaningful input size n for the algorithm is p, which corresponds to the number of elements to be sorted. The worst-case running time for the above algorithm is described as the solution of the following recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ T(n-1) + \Theta(n) & \text{if } n > 1. \end{cases}$$

The solution of the exercise ends here. For those who are interested, we show that the asymptotic worst-case running time of the recursive version of InsertionSort is still $\Theta(n^2)$. To this end we will prove that $T(n) = O(n^2)$ and $T(n) = \Omega(n^2)$.

To prove $T(n) = O(n^2)$ we show that for some $c > 0$, $T(n) \leq cn^2$ for all $n \geq 1$. We proceed by induction on n

Base Case ($n = 1$) In this case $T(1) = \Theta(1)$, therefore $T(1) \leq c_1$ for some positive constant c_1. By choosing $c \geq c_1$ we have $T(n) \leq c_1 \leq c = cn^2$.

Inductive Step ($n > 1$) In this case $T(n) = T(n-1) + \Theta(n)$. By inductive hypothesis and definition of Θ notation, there exist $c, c_2 > 0$ and $n_0 > 0$ such that $T(n) \leq c(n-1)^2 + c_2 n = cn^2 - 2cn + c + c_2 n$, for all $n \geq n_0$. Therefore we have

$$\begin{aligned} T(p) &\leq cn^2 - 2cn + c + c_2 n \\ &\leq cn^2 - 2cn + c + cn && \text{(choose } c \text{ such that } c \geq c_2) \\ &\leq cn^2 - 2cn + cn + cn && (n \geq 1) \\ &= cn^2 \end{aligned}$$

We prove now that $T(n) = \Omega(n^2)$ by showing that for some $d > 0$, $T(n) \geq dn^2$ for all $n \geq 1$. Again, we proceed by induction on n.

Base Case ($n = 1$) In this case $T(1) = \Theta(1)$, therefore $T(1) \geq d_1$ for some positive constant d_1. By choosing d satisfying $d \leq d_1$ we obtain $T(n) \geq d_1 \geq d = dn^2$.

Inductive Step ($n > 1$) In this case $T(n) = T(n-1) + \Theta(n)$. By inductive hypothesis and definition of Θ notation, there exist $d, d_2 > 0$ and $n_0 > 0$ such that $T(n) \geq d(n-1)^2 + d_2 n = dn^2 - 2dn + d + d_2 n$, for all $n \geq n_0$. Therefore we have

$$\begin{aligned} T(p) &\geq dn^2 - 2dn + d + d_2 n \\ &\geq dn^2 - 2dn + d_2 n && (d \geq 0) \\ &\geq dn^2 && (\text{if } d \leq d_2/2 \text{ then } -2dn + d_2 n \geq 0) \end{aligned}$$

CLRS 2.3–6. Using a binary search for determining the position where to insert the element at each iteration of the INSERTION-SORT does not improve the overall worst-case running time. This is because the insertion procedure still requires one to shift to the left the elements in the subarray to make room for the element prior to its insertion, and this requires a number of steps linear in the size of the currently sorted subarray $A[1\,..\,j-1]$.

Exercise 3.
Consider the problem of finding the smallest element in a nonempty array of numbers $A[1\,..\,n]$.

(a) Write an *incremental* algorithm that solves the above problem and determine its asymptotic worst-case running time.

(b) Write a *divide-and-conquer* algorithm that solves the above problem and determine its asymptotic worst-case running time.

(c) Assume that the length of A is a power of 2. Write a recurrence describing how many comparison operations (among elements of A) your divide-and-conquer algorithm performs, and solve the recurrence using the recursion-tree method.

Remark: count ONLY the comparisons performed among elements in A. E.g., a comparison like $i \leq A.length$ shall not be counted, whereas $A[i] \leq k$ where k is a variable storing some element of A shall be counted. Moreover, if you use expressions like $\min(A[i], A[j])$ for some indices i, j, that also counts as 1 comparison.

Hint: A full binary tree with n leaves has $n-1$ internal nodes (see CLRS B.5.3 pp.1177–1179).

Solution 3.
(a) An incremental algorithm to find the smallest element in an array $A[1\,..\,n]$ is the following.

```
SMALLEST(A)
1  min = A[1]
2  for i = 2 to A.length
3       if A[i] < min
4            min = A[i]
5  return min
```

The worst-case running time occurs when A is sorted in decreasing order, that is $A[1] > A[2] > \cdots > A[n]$, in which case the running time is

$$T(n) = c_1 + nc_2 + (n-1)c_3 + (n-1)c_4 + c_5 = \Theta(n)\,.$$

(b) A *divide-and-conquer* algorithm to find the smallest element in an array is presented in the pseudocode below. It takes an array $A[1\,..\,n]$, two indices p and q such that $1 \leq p \leq q \leq n$, and returns the value of the smallest element in the subarray $A[p\,..\,q]$.

```
SMALLEST(A, p, q)
1  if q = p
2       return A[p]
3  else
4       m = ⌊(p + q)/2⌋
5       min_L = SMALLEST(A, p, m)
6       min_R = SMALLEST(A, m + 1, q)
7       return min {min_L, min_R}
```

A meaningful size for the input is the number of elements of the subarray $A[p\,..\,q]$, i.e., $n = q - p + 1$. To simplify our analysis we will assume that $n = 2^k$ for some $k \in \mathbb{N}$. The asymptotic worst-case running time of SMALLEST is described by the following recurrence

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

where a denotes the running time for lines 1–2, and b denotes the running time for lines 1, 4, and 7. Note that under the assumption that $n = 2^k$, the recursion tree describing the unraveling of the above recurrence is a full binary tree having internal nodes labelled with b and leaves labelled with a. Since the number of leaves corresponds to the number of elements in the array $A[1\,..\,n]$, we have that

$$T(n) = an + b(n-1) = (a+b)n - b = \Theta(n) \tag{1}$$

Equation (1) can be proved by induction on k where $n = 2^k$. Equivalently, we show that for $k \in \mathbb{N}$, $T(2^k) = a2^k + b(2^k - 1)$. Let us rewrite the recurrence in terms of k

$$T(2^k) = \begin{cases} a & \text{if } k = 0, \\ 2T(2^{k-1}) + b & \text{if } k > 0. \end{cases}$$

Base Case ($k = 0$). Then, $T(2^k) = a = a2^k + b(2^k - 1)$.

Inductive Step ($k > 0$). Then,

$$\begin{aligned} T(2^k) &= 2T(2^{k-1}) + b && \text{(by def. of } T\text{)} \\ &= 2(a2^{k-1} + b(2^{k-1} - 1)) + b && \text{(by inductive hypothsis)} \\ &= a2^k + b2^k - b \\ &= a2^k + b(2^k - 1) \end{aligned}$$

(c) The number of comparisons among elements of A for an input instance of size n is described by the following recurrence

$$C(n) = \begin{cases} 0 & \text{if } n \leq 1, \\ 2C(n/2) + 1 & \text{if } n > 1. \end{cases}$$

As discussed before, $C(n)$ corresponds to the number of internal nodes of a full binary tree having n leaves. Therefore, the number of comparisons is $n - 1$.

★ **Exercise 4.**

Solve exercise CLRS 2.3–7.

Solution 4.

We have to describe a $\Theta(n \log n)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

We start by formally defining the above computational problem

Input: A sequence of integer numbers $S = \langle x_1, x_2, \ldots, x_n \rangle$ and an integer x.

Output: TRUE if there exist $i, j \in \{1, \ldots, n\}$ such that $x = x_i + x_j$, FALSE otherwise.

The following algorithm solves the above computational problem. The function F takes as input an array $S[1\,..\,n]$ of integers, and an integer x.

```
F(S, x)
1  MERGE-SORT(S, 1, S.length)
2  for i = 1 to S.length
3       key = x − A[i]
4       j = BIN-SEARCH(S, 1, S.length, key)
5       if j ≠ 0
6            return TRUE
7  return FALSE
```

The worst-case running time of the algorithm occurs when it returns FALSE. The asymptotic worst-case running time is $\Theta(n \log n)$ because line 1 takes $\Theta(n \log n)$, and line 4, which takes $\Theta(\log n)$, is executed n times.

Master Theorem:
Practice Problems and Solutions

Master Theorem

The Master Theorem applies to recurrences of the following form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

There are 3 cases:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with[1] $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$. Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

Practice Problems

For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

1. $T(n) = 3T(n/2) + n^2$

2. $T(n) = 4T(n/2) + n^2$

3. $T(n) = T(n/2) + 2^n$

4. $T(n) = 2^n T(n/2) + n^n$

5. $T(n) = 16T(n/4) + n$

6. $T(n) = 2T(n/2) + n \log n$

[1] most of the time, $k = 0$

7. $T(n) = 2T(n/2) + n/\log n$

8. $T(n) = 2T(n/4) + n^{0.51}$

9. $T(n) = 0.5T(n/2) + 1/n$

10. $T(n) = 16T(n/4) + n!$

11. $T(n) = \sqrt{2}T(n/2) + \log n$

12. $T(n) = 3T(n/2) + n$

13. $T(n) = 3T(n/3) + \sqrt{n}$

14. $T(n) = 4T(n/2) + cn$

15. $T(n) = 3T(n/4) + n\log n$

16. $T(n) = 3T(n/3) + n/2$

17. $T(n) = 6T(n/3) + n^2\log n$

18. $T(n) = 4T(n/2) + n/\log n$

19. $T(n) = 64T(n/8) - n^2\log n$

20. $T(n) = 7T(n/3) + n^2$

21. $T(n) = 4T(n/2) + \log n$

22. $T(n) = T(n/2) + n(2 - \cos n)$

Solutions

1. $T(n) = 3T(n/2) + n^2 \implies T(n) = \Theta(n^2)$ (Case 3)
2. $T(n) = 4T(n/2) + n^2 \implies T(n) = \Theta(n^2 \log n)$ (Case 2)
3. $T(n) = T(n/2) + 2^n \implies \Theta(2^n)$ (Case 3)
4. $T(n) = 2^n T(n/2) + n^n \implies$ Does not apply (a is not constant)
5. $T(n) = 16T(n/4) + n \implies T(n) = \Theta(n^2)$ (Case 1)
6. $T(n) = 2T(n/2) + n \log n \implies T(n) = n \log^2 n$ (Case 2)
7. $T(n) = 2T(n/2) + n/\log n \implies$ Does not apply (non-polynomial difference between $f(n)$ and $n^{\log_b a}$)
8. $T(n) = 2T(n/4) + n^{0.51} \implies T(n) = \Theta(n^{0.51})$ (Case 3)
9. $T(n) = 0.5T(n/2) + 1/n \implies$ Does not apply ($a < 1$)
10. $T(n) = 16T(n/4) + n! \implies T(n) = \Theta(n!)$ (Case 3)
11. $T(n) = \sqrt{2}T(n/2) + \log n \implies T(n) = \Theta(\sqrt{n})$ (Case 1)
12. $T(n) = 3T(n/2) + n \implies T(n) = \Theta(n^{\lg 3})$ (Case 1)
13. $T(n) = 3T(n/3) + \sqrt{n} \implies T(n) = \Theta(n)$ (Case 1)
14. $T(n) = 4T(n/2) + cn \implies T(n) = \Theta(n^2)$ (Case 1)
15. $T(n) = 3T(n/4) + n \log n \implies T(n) = \Theta(n \log n)$ (Case 3)
16. $T(n) = 3T(n/3) + n/2 \implies T(n) = \Theta(n \log n)$ (Case 2)
17. $T(n) = 6T(n/3) + n^2 \log n \implies T(n) = \Theta(n^2 \log n)$ (Case 3)
18. $T(n) = 4T(n/2) + n/\log n \implies T(n) = \Theta(n^2)$ (Case 1)
19. $T(n) = 64T(n/8) - n^2 \log n \implies$ Does not apply ($f(n)$ is not positive)
20. $T(n) = 7T(n/3) + n^2 \implies T(n) = \Theta(n^2)$ (Case 3)
21. $T(n) = 4T(n/2) + \log n \implies T(n) = \Theta(n^2)$ (Case 1)
22. $T(n) = T(n/2) + n(2 - \cos n) \implies$ Does not apply. We are in Case 3, but the regularity condition is violated. (Consider $n = 2\pi k$, where k is odd and arbitrarily large. For any such choice of n, you can show that $c \geq 3/2$, thereby violating the regularity condition.)

Exercise Session 04

.

MAX-HEAPIFY(A, i)

```
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4        largest = l
5  else
6        largest = i
7  if r ≤ A.heap-size and A[r] > A[largest]
8        largest = r
9  if largest ≠ i
10       exchange A[i] with A[largest]
11       MAX-HEAPIFY(A, largest)
```

[1]

Exercise 1

Starting with the procedure MAX-HEAPIFY (CLRS, pp. 154), write pseudocode for the procedure MIN-HEAPIFY(A, i), which performs the corresponding manipulation on a min-heap. How does the running time of MIN-HEAPIFY compare to that of MAX-HEAPIFY?

Solution 1

The procedure for MIN-HEAPIFY(A, i) assumes that the children of i are min-heaps and works similarly to MAX-HEAPIFY by pushing down the value $A[i]$ exchanging it with the smallest element among $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$. The pseudocode for MIN-HEAPIFY is described below.

MIN-HEAPIFY(A, i)

```
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] < A[i]
 4        smallest = l
 5  else
 6        smallest = i
 7  if r ≤ A.heap-size and A[r] < A[smallest]
 8        smallest = r
 9  if smallest ≠ i
10        exchange A[i] with A[smallest]
11        MIN-HEAPIFY(A, smallest)
```

The worst-case running time of MIN-HEAPIFY work can be proved to be $O(\lg n)$ by following the same arguments used for MAX-HEAPIFY.

Exercise 2.

Consider the pseudocode of the PARTITION procedure.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Assume that all elements in the array $A[p\,..\,r]$ are equal, that is, $A[p] = A[p+1] = \cdots = A[r]$. What value will PARTITION(A, p, r) return? How does QUICKSORT perform on arrays that have the same value compared with INSERTION-SORT and MERGESORT?

Solution 2.
Note that if $A[p\,..\,r]$ have the same value, the condition in line 4 is always true and lines 5–6 are executed. Therefore at the end of the for loop the value of i will be $p-1+r-p=r-1$, and the value returned by PARTITION(A,p,r) will be r.
As argued above, if the array A has the same value, procedure PARTITION produces unbalanced partitionings at each recursive call of QUICKSORT causing the algorithm to perform in time $\Theta(n^2)$ (i.e., the worst-case running time). In contrast, INSERTION-SORT performs in time $\Theta(n)$ when the input array is already sorted (i.e., the best-case running time). Hence, for arrays that have the same value insertion sort performs much better than quicksort. The same argument can be trivially generalised for arrays that are already in increasing order.

As for MERGESORT its complexity is not affected the relative order of the elements in A. Therefore it will still run on $\Theta(n \lg n)$.

Exercise 3.

Modify the pseudocode of the PARTITION procedure so that the QUICKSORT algorithm (CLRS, pp. 171) will sort in nonincreasing order. Argument about the correctness of your solution.

Solution 3.
It suffices to modify the partition procedure by replacing in line 4 the condition $A[j] \leq x$ with $A[j] \geq x$. This modification makes sure that all elements in the left partition are greater than or equal to the pivot element x and (dually) all elements that are smaller than x will fall in the right partition. The modified pseudocode for PARTITION is

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≥ x
5            i = i + 1
6            exchange A[i] with A[j]
7       exchange A[i + 1] with A[r]
8  return i + 1
```

Exercise 4.

Consider the pseudocode of COUNTING-SORT (CLRS, pp. 195)

```
COUNTING-SORT(A, B, k)
 1  let C[0 .. k] be a new array
 2  for j − 1 to k
 3       C[i] = 0
 4  for j = 1 to A.length
 5       C[A[j]] = C[A[j]] + 1
 6  for i = 1 to k
 7       C[i] = C[i] + C[i − 1]
 8  for j = A.length downto 1
 9       B[C[A[j]]] = A[j]
10       C[A[j]] = C[A[j]] − 1
```

Modify the above pseudocode by replacing the **for**-loop header in line 8 as

```
8  for j = 1 to A.length
```

Is the modified algorithm stable?

Solution 4.

The modified algorithm is not stable. As for the original COUNTING-SORT algorithm, in the final **for** loop an element equal to one taken from A earlier is placed before the earlier one (i.e., at a lower index position) in the output array B. The original algorithm was stable because an element taken from A later started out with a lower index than one taken earlier. But in the modified algorithm, an element taken from A later started out with a higher index than one taken earlier.

In particular, the algorithm still places the elements with value k in positions $C[k-1]+1$ through $C[k]$, but in the reverse order of their appearance in A.

★ **Exercise 5.**

Use induction to prove that RADIX-SORT is correct. Where does your proof need the assumption that the intermediate sorting procedure is stable? Justify your answer.

Solution 5.

We proceed by induction on the number of digits d.

Base Case ($d = 1$). there is only one digit, so sorting on that digit sorts the array.

Inductive step ($d > 1$). Assuming that radix sort works for $d-1$ digits, we'll show that it works for d digits. Radix sort sorts separately on each digit, starting from digit 1. Thus, radix sort of d digits, which sorts on digits $1, \ldots, d$ is equivalent to radix sort of the low-order $d-1$ digits followed by a sort on digit d. By our induction hypothesis, the sort of the low-order $d-1$ digits works, so just before the sort on digit d, the elements are in order according to their low-order $d-1$ digits. The sort on digit d will order the elements by their d-th digit. Consider two elements, a and b, with d-th digits a_d and b_d respectively.

- If $a_d < b_d$, the sort will put a before b, which is correct, since $a < b$ regardless of the low-order digits.
- If $a_d > b_d$, the sort will put a after b, which is correct, since $a > b$ regardless of the low-order digits.
- If $a_d = b_d$, the sort will leave a and b in the same order they were in, because it is stable. But that order is already correct, since the correct order of a and b is determined by the low-order $d-1$ digits when their d-th digits are equal, and the elements are already sorted by their low-order $d-1$ digits.

If the intermediate sort were not stable, it might rearrange elements whose d-th digits were equal –elements that were in the right order after the sort on their lower-order digits.

Self-Study 1

ALG-CPH-F23: Algorithms and Data Structures (SW2)

Instructions. This exam consists of **four questions** and you have **four hours** to solve them. Note that each question begins on a new page. The maximum possible score is **100 points**, but bear in mind that these points are only advisory, and will be weighted following the exam. Please provide your answers digitally, either in a separate document, directly on this exam PDF, or both.
Remember to upload your digital answers as a single PDF file. It will not be possible to hand in anything physically for this exam. Throughout the exam, make sure to clearly indicate which question you are answering.

- Before starting with the exam, please make sure you read and understand these guidelines.
- For each question in the exam, make sure you read the text carefully, and understand what the problem is before solving it. Terms in bold are of particular importance.
- During the exam, you are allowed to use notes, slides, all content on Moodle, and your internet access / online search. However, you are **not** allowed to use tools which **directly solve** the problem you are working on, or AI-based tools. For instance, using a Master Theorem solver or ChatGPT is not allowed.
- You are **not** allowed to communicate with others during the exam, beyond asking for administrative help from the administrative staff.
- It is highly recommended to have a look at the **entire** exam before starting, so you get an idea of the time each question might take.
- You may answer in English, Danish, Norwegian, Swedish, or in any combination of these.

Question 1. 20 Pts

Identifying asymptotic notation. (Note: lg means logarithm in base 2)

(1.1) [5 Pts] Mark **ALL** the correct answers. $3 \lg n + 5\sqrt{n^4}$ is:

☐ **a)** $\Theta(\lg n)$ ☐ **b)** $\Theta(n)$ ☐ **c)** $\Theta(\sqrt{n})$ ☐ **d)** $\Theta(n^2)$ ☐ **e)** $\Theta(2^n)$

(1.2) [5 Pts] Mark **ALL** the correct answers. $5 \lg n^2 + \sqrt{n^2} + \lg n$ is:

☐ **a)** $O(\lg n)$ ☐ **b)** $O(n)$ ☐ **c)** $O(n \lg n)$ ☐ **d)** $O(n^2)$ ☐ **e)** $O(2^n)$

(1.3) [5 Pts] Mark **ALL** the correct answers. $\sqrt{n^2} \lg n + 30n + 42^2 \lg n$ is

☐ **a)** $\Omega(\lg n)$ ☐ **b)** $\Omega(n)$ ☐ **c)** $\Omega(n \lg n)$ ☐ **d)** $\Omega(n^2)$ ☐ **e)** $\Omega(2^n)$

(1.4) [5 Pts] Mark **ALL** the correct answers. Consider the following recurrence

$$T(n) = \begin{cases} 5 & \text{if n} \leq 1 \\ 5\text{T(n/6)} + \text{n lg}\sqrt{n^2} & \text{if n} > 1 \end{cases}$$

☐ **a)** The master method can be used to solve this recurrence.

☐ **b)** The substitution method can be used to solve this recurrence.

☐ **c)** Using the first case of the master theorem, we can prove that $T(n) = \Theta(n^2)$.

☐ **d)** Using the second case of the master theorem, we can prove that $T(n) = \Theta(n)$.

☐ **e)** Using the third case of the master theorem, we can prove that $T(n) = \Theta(n \lg n)$.

Solution 1.

(1.1) $= n^2 = \Theta(n^2)$. Correct answer: (d).

(1.2) $= n = \Theta(n)$. Correct answers: (b, c, d, e).

(1.3) $= n \lg n$ Correct answers: (a, b, c).

(1.4) Master method, using $a = 5$, $b = 6$ and $f(n) = n \lg n$. Using case 3, since $f(n) = \Omega(n)$. Hence, $T = \Theta(n \lg n)$.

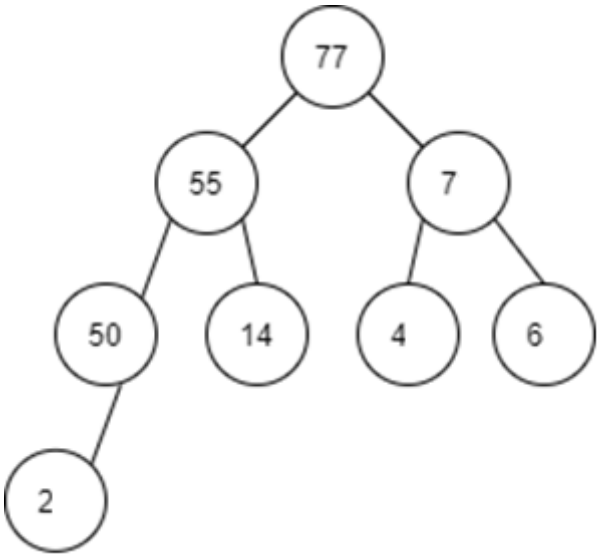
Correct answers: (a, b, e).

Question 2. 31 Pts

(2.1) [4 Pts] Mark **ALL** the correct statements. Consider a modification to QUICK-SORT, which first **reverses** the input-array before sorting. We will call this procedure REVERSE-AND-THEN-QUICK-SORT, or REVQ-SORT for short. Assume that reversing takes constant time.

- ☐ **a)** REVQ-SORT will in practice **some times** have a faster running time than QUICK-SORT.
- ☐ **b)** REVQ-SORT has the same asymptotic time as QUICK-SORT
- ☐ **c)** REVQ-SORT will guarantee that we **always** avoid the worst-case running time of QUICK-SORT.
- ☐ **d)** Given a sorted input, REVQ-SORT will result in the worst-case running time of QUICK-SORT.

(2.2) [5 Pts] Consider the following tree, T, and mark **ALL** the correct answers below:



- ☐ **a)** T satisfies the Max-Heap property
- ☐ **b)** T satisfies the binary search tree property
- ☐ **c)** If all nodes in the tree are coloured *red*, with the exception of the root which is coloured *black*, T satisfies the Red-Black Tree property.
- ☐ **d)** The height of T is 3
- ☐ **e)** T corresponds to a binary tree interpretation of the array $[77, 55, 7, 50, 14, 4, 6, 2]$

(2.3) [12 Pts] Consider the following recursive algorithm, MOST-CATS. The input is an array $A[1 \ldots A.length]$ of pictures, some of which include one or more cats. When the recursion bottoms out, the procedure calls a cat-detection procedure, DETECT-N-CATS, which returns the number of cats present in the picture. Assume that DETECT-N-CATS uses constant time.

```
MOST-CATS(A)
1  if length(A) == 1
2      return DETECT-N-CATS(A[1])
3  else
4      newsize = length(A)/4
5      a = MOST-CATS(A[0 : newsize])
6      b = MOST-CATS(A[newsize : 2 * newsize])
7      c = MOST-CATS(A[2 * newsize : 3 * newsize])
8      d = MOST-CATS(A[3 * newsize : 4 * newsize])
9      return max(a, b, c, d)
```

Assume that the input size at the first call is $n = 4^k$ for some $k \in \mathbb{N}$. Further assume that the notation A[i:j] returns a sub-array of A from index i (inclusive) to index j (exclusive).

(a) [4 Pts] Write the recurrence $T(n)$ describing the running time of MOST-CATS.

(b) [8 Pts] Use the Master Theorem to prove the running time of this recurrence, showing all the steps you use to reach your conclusion.

(2.4) [10 Pts] In this assignment, you will work with a Doubly Linked List. Assume that your list has an attribute L.head which points at the head of the list (*nil* if L is empty). Further assume that each element in the list has the attributes x.next (*nil* if x is the tail) and x.prev (*nil* if x is L.head).

(a) [7 Pts] Give a procedure which takes as input a Doubly Linked List, reverses it, and returns the reversed list. The procedure **must** run in linear time, i.e., $\Theta(n)$. Name the procedure REVERSE-DOUBLY-LINKED-LIST. Write pseudocode and provide a brief description in words how your procedure works.

(b) [3 Pts] Analyse the runtime of your version of REVERSE-DOUBLY-LINKED-LIST.

Solution 2.

(2.1) There are cases where RevQ-SORT may be faster, e.g., if the original input is reversed. However, asymptotically this will be the same, as RevQ-SORT = Θ(QUICK-SORT) + c = Θ(QUICK-SORT). There is no guarantee, however, that this will avoid the worst-case running time. Finally, given a sorted input, we will end up in the worst case.

Correct answers: (a, b, d).

(2.2) The tree does satisfy the Max-Heap property, as all children are smaller than their parents. The height of the tree is 3 since its longest path from the root to a leaf has 3 edges. The tree does coresond to the binary tree interpretation of the array.

Correct answers: (a, d, e).

(2.3) (a) The recurrence should have the form

$$T(n) = \begin{cases} c_1 & \text{if } n \leq 1, \\ 4T(n/4) + c_2 & \text{if } n > 1. \end{cases}$$

(b) This should evaluate to $\Theta(n)$, which intuitively is clear as each element in the array is checked once. Using the Master Theorem, the same conclusion can be reached, as $n^{\log_4^4} \geq f(n) = 1$ matches case 1.

(2.4) (a) The solution must take into account an update of L.head, x.next, x.prev, and correctly use the properties of a doubly linked list. E.g. assuming that the input is an array, or that the length of L is known, or that L can be indexed as an array, is not a correct interpretation of the assignment. A single loop should suffice, starting at L.head, terminating when x.next=nil, and swapping each x.next with x.prev.

(b) The following analysis should therefore also be relatively straight-forward in showing runtime $= \Theta(n)$

Best of luck!
- Johannes

Exercise Session 06

Exercise 1.
In class we have seen that using arrays for implementing stack may lead to underflow and overflow problems. Propose an alternative implementation of the stack operations that use doubly linked lists. What is the worst-case running time for the POP and PUSH operations? May we still incur in situations where the stack underflows or overflows?

Solution 1.
We assume S to be a doubly linked list and we use the attribute *head* to point to the top of the stack. Then, the stack is empty if the list is. Pushing an element on top of the stack corresponds to list insertion as in CLRS. Finally, the pop procedure corresponds to the deletion of the head of the list.

```
STACK-EMPTY(S)
1  return S.head == NIL

PUSH(S, x)
1  LIST-INSERT(S, x)

POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else
4       x = S.head
5       LIST-DELETE(S, x)
6       return x
```

The worst-case running time is $\Theta(1)$ for each operation as the corresponding operations on list used therein have constant worst-case running time too. This new list-based implementation can still produce underflow when trying to pop from an empty stack. However, it does not suffer any longer from overflow problems because the list can be arbitrarily long.

Exercise 2.
Exercises 10.4-2 and 10.4-3 CLRS pp. 248.

Solution 2.

(a) We have to write a $O(n)$ time recursive procedure that, given an n-node binary tree, prints out the key of each node in the tree. The following procedure prints the tree rooted at node x.

```
REC-PRINT(x)
1  if x ≠ NIL
2       print x.key
3       REC-PRINT(x.left)
4       REC-PRINT(x.right)
```

Given a rooted binary tree T we can print it by calling REC-PRINT($T.root$). The worst-case running time of the above algorithm follows the following recurrence

$$T(n) = \max_{0 \le q \le n-1} T(q) + T(n-q-1) + \Theta(1)$$

where the parameter q ranges from 0 to $n-1$ because the two subproblems have total size $n-1$. We guess that $T(n) \le cn$ for some constant $c > 0$. Substituting this guess into the recurrence

we obtain

$$\begin{aligned} T(n) &\leq \max_{0 \leq q \leq n-1} cq + c(n-q-1) + \Theta(1) \\ &= cn - c + \Theta(1) \\ &\leq cn \end{aligned}$$

The last inequality holds by choosing a constant c large enough so that it dominates the $\Theta(1)$ term. Thus $T(n) = O(n)$.

(b) We have to write a $O(n)$ time nonrecursive procedure that, given an n-node binary tree, prints out the key of each node in the tree using a stack as auxiliary data structure. The following procedure prints the tree T without using recursion.

```
ITER-PRINT(T)
1  Initialise an empty stack S
2  PUSH(S, T.root)
3  while ¬STACK-EMPTY(S)
4       x = POP(S)
5       if x ≠ NIL
6            print x.key
7            PUSH(S, x.right)
8            PUSH(S, x.left)
```

Note that the above algorithm runs in linear time, because the body of the while loop takes constant time and the number of iterations is linearly bounded by the number of nodes in T since each node is inserted exactly once in the stack S and at each iteration one element is deleted from S. To be precise the number of insertions in S is $O(n)$ since also the NIL pointers are inserted.

Exercise 3.
Singly linked lists are a variant of linked lists where each element x has two attributes, $x.key$ that stores the key, and $x.next$ that points to the next element in the list. Can you implement the dynamic-set operation INSERT on a singly linked list in $O(1)$ time? How about DELETE?

Solution 3.
The insert operation in singly linked lists is very similar to that for doubly linked lists.

```
LIST-INSERT(L, x)
1  x.next = L.head
2  L.head = x
```

Clearly, the worst-case running time of LIST-INSERT(L, x) is $\Theta(1)$.
As for the procedure LIST-DELETE(L, x), our goal is to skip the element x in L by making its predecessor point to the element that follows x. However, differently from the case for doubly linked lists, we don't have the predecessor of x ready in our hands: we have to retrieve it by performing a linear search in L starting form its head.

```
LIST-DELETE(L, x)
 1  p = NIL
 2  c = L.head
 3  while c ≠ x
 4       p = c
 5       c = c.next
 6  // Now c = x and p is its predecessor
 7  if p ≠ NIL
 8       p.next = x.next
 9  else // x was the head of L
10       L.head = x.next
```

In the above pseudocode we assume that x is an element of L, hence we will eventually encounter it while scanning the list. Once we find it we update the pointers so to skip x in L. The worst-case running time is $\Theta(n)$ where n is the length of L, because if x is the last element in the list the procedure scans all the n elements in L.

Exercise 4.
Give a $\Theta(n)$-time nonrecursive procedure that reverses a *singly* linked list of n elements. The procedure should use no more than constant storage beyond that needed for the list itself.

Solution 4.
The following pseudocode used tree pointers p, c, and n to store respectively the previous, current, and next element in the original list. Initially, p and n are set to nil NIL and c points to the head of the list L. While c does not point to the end of the list, we do the following: store the new next element in n, update the attribute *next* of the current element to point to p. Finally, we move one step forward by updating the values of p and c.

```
REVERSE(L)
1  p = NIL
2  c = L.head
3  n = NIL
4  while c ≠ NIL
5        n = c.next
6        c.next = p
7        p = c
8        c = n
```

The above procedure scans the list once, and each iteration takes constant time. Therefore the running time of REVERSE takes $\Theta(n)$. Furthermore, we use only three additional variables beyond the space needed for the list L.

Exercise 5.
Implement a queue by a singly linked list Q. The operations ENQUEUE and DEQUEUE should still take $O(1)$ time.

Solution 5.
We assume that the singly linked list Q has two attributes: $Q.head$ which is the usual head of the linked list and $Q.tail$ which points to the last element of the linked list (as usual, $Q.tail$ = NIL when the list is empty). The operations ENQUEUE and DEQUEUE are implemented as follows.

```
ENQUEUE(Q, x)
1  if Q.tail ≠ NIL
2        Q.tail.next = x
3  Q.tail = x
```

```
DEQUEUE(Q)
1  if Q.head ≠ NIL
2        x = Q.head
3        Q.head = x.next
4        return x
5  else
6        error "underflow"
```

Clearly, both operations take $O(1)$ time in the worst case.

Exercise Session 07

Instructions. Each exercise is associated with a teaching assistant who is responsible to provide feedback for that exercise. You can ask for feedback by sending your solution by email or by asking questions by video chat using Microsoft Teams (`https://www.its.aau.dk/vejledninger/microsoft-teams/`) You can login to Teams by using your AAU email and password.
The TAs will be available in private chat from 8:15 until 10:00, and will answer all the emails sent from 8:15 to 12:00. Remember to **contact the TA by chat before starting a video chat** to be sure that the TA is ready to pick the call.
Beware that the response time may be slow. In that case, try to solve the following exercises while waiting for one TA to be available.

Exercise 1.
(TA: **Niels Tobias Nørgaard Svendsen** `mailto:ntns15@student.aau.dk`)
[CLRS 11.1-1] Suppose that a dynamic set S is represented by a direct-address table T of length m. Describe a procedure that finds the maximum element of S . What is the worst-case performance of your procedure?

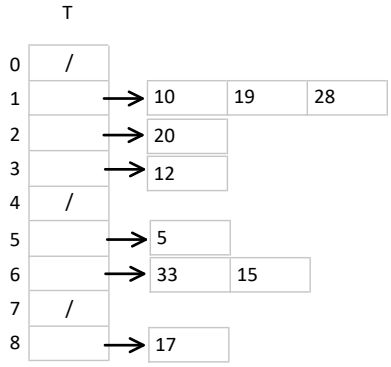
Solution 1.
Recall that direct-address tables work under the assumption that the set of actual keys is $K = \{0, \ldots, m-1\}$, that is, they represent dynamic sets containing elements in the range $0, \ldots, m-1$. The maximum element can be found by performing a linear search starting from the slot $m-1$ down to 0 searching for the first occupied slot in the table. For the following algorithm we assume that the table represents a non-empty set, otherwise the procedure returns NIL.

DIRECT-ADDRESS-MAXIMUM(T)
1 $k = m - 1$
2 **while** $k \geq 1$ and DIRECT-ADDRESS-SEARCH(T, k) = NIL
3 $\quad k = k - 1$
4 **return** DIRECT-ADDRESS-SEARCH(T, k)

Note that the while loop terminates as soon as $k = 0$ or the k-th slot is not empty. Therefore, when whe execute line 4 we return either the first nonempty slot found stating from $m-1$ or whatever element is present in the 0-th slot. The worst-case occurs when the maximum element in T is 0 leading the algorithm to scan all the slots of T. Therefore the worst-case running time of DIRECT-ADDRESS-MAXIMUM(T) is $O(m)$.

Exercise 2.
(TA: **Peter Vergerakis** `mailto:pverge15@student.aau.dk`)
[CLRS 11.2-2] Demonstrate what happens when we insert the keys $5; 28; 19; 15; 20; 33; 12; 17; 10$ into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be $h(k) = k \mod 9$.

Solution 2.
After inserting the keys $5; 28; 19; 15; 20; 33; 12; 17; 10$ into an initially empty hash table with 9 slots, we end up in the following configuration



Exercise 3.

(TA: **Rasmus Grønkjær Tollund** `mailto:rtollu18@student.aau.dk`)

[CLRS 11.2-5] Suppose that we are storing a set of n keys into a hash table of size m. Show that if the keys are drawn from a universe U with $|U| > nm$, then U has a subset of size n consisting of keys that all hash to the same slot, so that the worst-case searching time for hashing with chaining is $\Theta(n)$.

Solution 3.

The Dirichlet's box principle —a.k.a. pigeon hole principle— states that for $n, m \in \mathbb{N}$, if $nm+1$ objects are distributed among m sets, then at least one of the sets will contain at least $n+1$ objects.
In hash tables, the redistribution of objects in the slots is done by the hash function. In this context, the Dirichlet's box principle can be read as follows: regardless from the choice of the hash function h, there exist a subset of keys $K \subseteq U$ such that $|K| > n$ and a slot $0 \leq j \leq m$, such that for all $k \in K$, $h(k) = j$. The worst-case searching time occurs when all the n elements in the hash table belong to K. In this case the list L pointed by the slot j has size n. We know that the worst-case searching time in a list of n elements is $\Theta(n)$, therefore also the worst-case searching time in a hash table is $\Theta(n)$.

Exercise 4.

(TA: **Alessandro Tibo** `mailto:alessandro@cs.aau.dk`)

[CLRS 11.4-1] Consider inserting the keys $10; 22; 31; 4; 15; 28; 17; 88; 59$ into a hash table of length $m = 11$ using *open addressing* with the auxiliary function $h'(k) = k$. Illustrate the result of inserting these keys using linear probing, using quadratic probing with $c_1 = 1$ and $c_2 = 3$, and using double hashing with $h_1(k) = k$ and $h_2(k) = 1 + (k \mod (m-1))$.

Solution 4.

Recall that the hash functions under linear probing, quadratic probing, and double hashing are

$$h(k,i) = (h'(k) + i) \mod m \qquad \text{(LINEAR PROBING)}$$
$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m \qquad \text{(QUADRATIC PROBING)}$$
$$h(k,i) = (h_1(k) + i h_2(k)) \mod m \qquad \text{(DOUBLE HASHING)}$$

Let first consider inserting the keys $10; 22; 31; 4; 15; 28; 17; 88; 59$ using linear probing. Key 10 goes in slot $h(10,0) = 10$ because it is free. Key 22 goes in slot $h(22,0) = 22 \mod 11 = 1$, key 31 goes in slot $h(31,0) = 9$, and key 4 goes in slot $h(4,0) = 4$, all because they find a free slot at the first probe. Key 15 tries to go in slot $h(15,0) = 15 \mod 11 = 4$ but it finds it occupied, then it tries the next probe $h(15,1) = (15+1) \mod 11 = 5$ and finds a free slots where to go. The insertions proceed in this way yielding the final table configuration $T = [22, 88, \text{NIL}, \text{NIL}, 4, 15, 28, 17, 59, 31, 10]$. Overall, the number of unsuccessful probes is 7.
Let now repeat the same insertion sequence using now quadratic probing. The final table configuration is $T = [22, \text{NIL}, 88, 17, 4, \text{NIL}, 28, 59, 15, 31, 10]$ and the total number of unsuccessful probes was 14.
Using double hashing, the final table configuration is $T = [22, \text{NIL}, 59, 17, 4, 15, 28, 88, \text{NIL}, 31, 10]$, and the total number of unsuccessful probes was 7.

Exercise 5.

(TA: **Danny Bøgsted Poulsen** `mailto:dannybpoulsen@cs.aau.dk`)

[CLRS 11.4-3] Consider an open-address hash table with uniform hashing. Give upper bounds on the expected number of probes in an unsuccessful search and on the expected number of probes in a successful search when the load factor is 3/4 and when it is 7/8.

Solution 5.

By Theorem 11.6 in CLRS, the expected number of probes in an unsuccessful search is bounded by $1/(1-\alpha)$. Thus, for $\alpha = 3/4$ we have $1/(1-\alpha) = 4$, and for $\alpha = 7/8$ we have $1/(1-\alpha) = 8$.

By Theorem 11.8 in CLRS, the expected number of probes in a successful search is bounded by $(1/\alpha)\ln(1/1-\alpha)$. Hence, for $\alpha = 3/4$ we have $(1/\alpha)\ln(1/1-\alpha) = (4/3)\ln 4 \cong 1.85$, and for $\alpha = 7/8$ we have $(1/\alpha)\ln(1/1-\alpha) = (8/7)\ln 8 \cong 2.38$.

Exercise Session 08

Instructions. Each exercise is associated with a teaching assistant who is responsible to provide feedback for that exercise. You can ask for feedback by sending your solution by email or by asking questions by video chat using Microsoft Teams (`https://www.its.aau.dk/vejledninger/microsoft-teams/`) You can login to Teams by using your AAU email and password.
The TAs will be available in private chat from 8:15 until 10:00, and will answer all the emails sent from 8:15 to 12:00. Remember to **contact the TA by chat before starting a video chat** to be sure that the TA is ready to pick the call.
Beware that the response time may be slow. In that case, try to solve the following exercises while waiting for one TA to be available.

Exercise 1.
(TA: **Peter Vergerakis** `mailto:pverge15@student.aau.dk`)
[CLRS 12.3-1] Implement a recursive variant of the TREE-INSERT procedure.

Solution 1.
We present two alternative implementations. The first one is described in the following pseudocode

```
TREE-INSERT(T, z)
 1  x = T.root
 2  if x == NIL
 3        z.p = NIL
 4        T.root = z
 5  else
 6        if z.key < x.key
 7              if x.left == NIL
 8                    x.left = z
 9                    z.p = x
10              else
11                    TREE-INSERT(x.left, z)
12        else
13              if x.right == NIL
14                    x.right = z
15                    z.p = x
16              else
17                    TREE-INSERT(x.right, z)
```

An alternative solution can make use of an auxiliary procedure which is called when T is not empty.

```
TREE-INSERT(T, z)
1  if T.root == NIL
2        z.p = NIL
3        T.root = z
4  else
5        TREE-INSERT-AUX(NIL,T.root,z)
```

```
TREE-INSERT-AUX(y, x, z)
 1  if x == NIL
 2        z.p = y
 3        if z.key < y.key
 4              y.left = z
 5        elseif z.key ≥ y.key
 6              y.right = z
 7  elseif z.key < x.key
 8        TREE-INSERT-AUX(x, x.left, z)
 9  else
10        TREE-INSERT-AUX(x, x.right, z)
```

Exercise 2.

(TA: **Niels Tobias Nørgaard Svendsen** `mailto:ntns15@student.aau.dk`)

[CLRS 12.3-3] We can sort a sequence of n numbers by iteratively inserting each number in a binary search tree and then performing an inorder tree walk. Write the pseudocode of this algorithm. What are the worst-case and best-case running times for this sorting algorithm?

Solution 2.

The pseudocode of the suggested sorting procedure is

```
BST-SORT(A)
1  let T be an empty binary search tree
2  for i = 1 to n
3        TREE-INSERT(T, A[i])
4  INORDER-TREE-WALK(T.root)
```

The worst-case running time is $\Theta(n^2)$ and occurs when the tree T in line 4 is a chain that is a tree of height $n-1$. We have seen in class that this may occur e.g., when the array A is already sorted.
The best-case running time is $\Theta(n \log n)$ and occurs when the tree T in line $4 hasheight \Theta(\log n)$. This may occur when T is a randomly built BST.

Exercise 3.

(TA: **Rasmus Grønkjær Tollund** `mailto:rtollu18@student.aau.dk`)

Consider the binary search tree T depicted in Figure 1. Delete the node with *key = 10* from T by applying the procedure TREE-DELETE(T, z) as described in CLRS pp. 298.

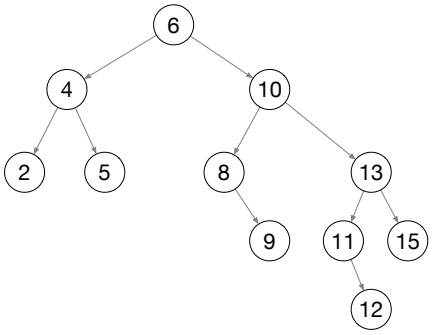


Figure 1: Binary Tree

Solution 3.

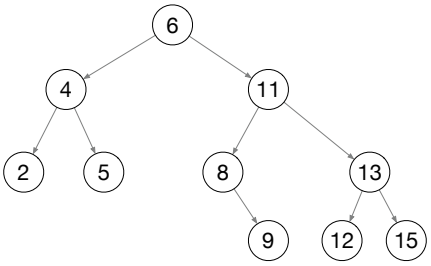
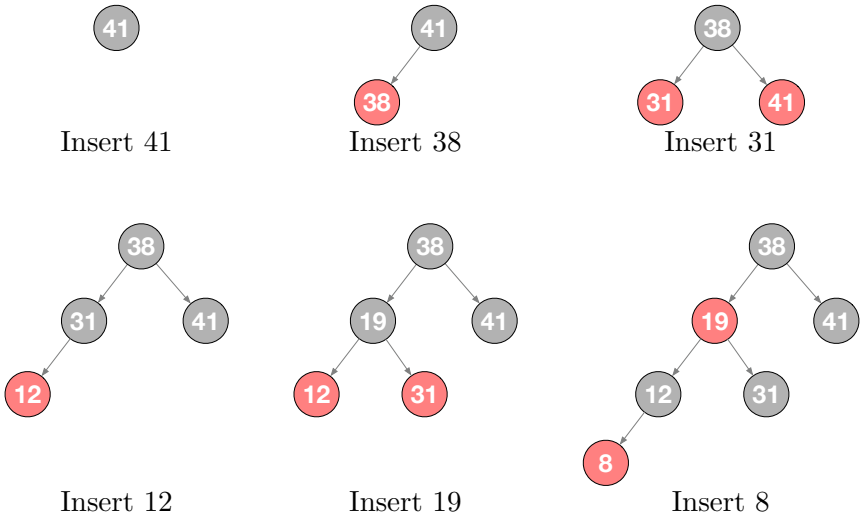


Figure 2: Binary Tree after deletion of node with $key = 10$

Exercise 4.
(TA: **Danny Bøgsted Poulsen** `mailto:dannybpoulsen@cs.aau.dk`)
[CLRS 11.1-1] Show the red-black trees that result after successively inserting the keys 41; 38; 31; 12; 19; 8 into an initially empty red-black tree.

Solution 4.

Step by step insertions



Exercise 5.
(TA: **Alessandro Tibo** `mailto:alessandro@cs.aau.dk`)
Consider the red-black tree T depicted in Figure 3. Insert first a node with $key = 15$ in T, then delete the node with $key = 8$ Show all the intermediate transformations of the red-black tree with particular emphasis on the rotations.

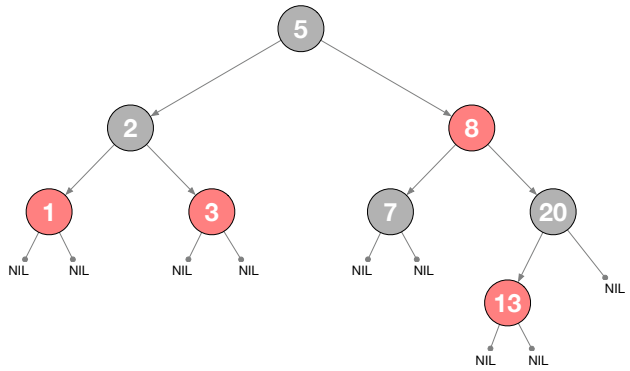
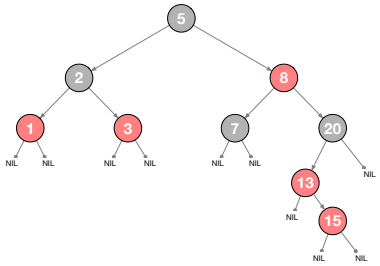


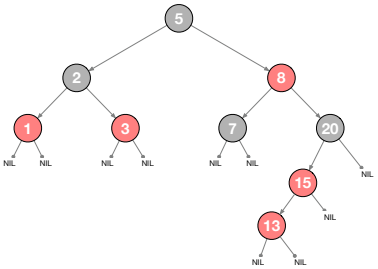
Figure 3: RB-tree

Solution 5.

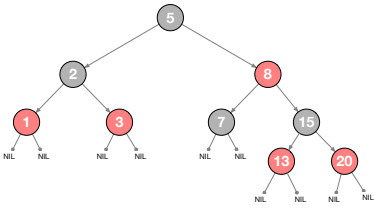
Insertion of 15



insertion

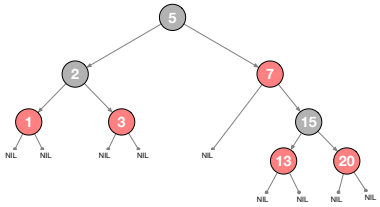


left rotation

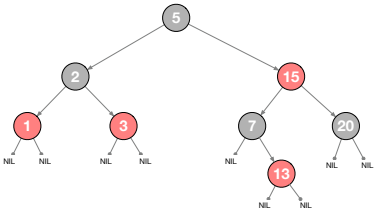


right rotation + re-coloring

Deletion of 8



delete 8 + transplant



left rotation

Exercise Session 10

Instructions. Each exercise is associated with a teaching assistant who is responsible to provide feedback for that exercise. You can ask for feedback by sending your solution by email or by asking questions by video chat using Microsoft Teams (`https://www.its.aau.dk/vejledninger/microsoft-teams/`) You can login to Teams by using your AAU email and password.
The TAs will be available in private chat from 8:15 until 10:00, and will answer all the emails sent from 8:15 to 12:00. Remember to **contact the TA by chat before starting a video chat** to be sure that the TA is ready to pick the call.
Beware that the response time may be slow. In that case, try to solve the following exercises while waiting for one TA to be available.

Exercise 1.
(TA: **Niels Tobias Nørgaard Svendsen** `mailto:ntns15@student.aau.dk`)
[CLRS 22.1-3] The transpose of a directed graph $G = (V, E)$ is the graph $G^T = (V, E^T)$, where $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$. Describe efficient algorithms for computing G^T from G, for both adjacency-list and adjacency-matrix representations of G. Analyse the running times of your algorithms.

Solution 1.
An algorithm for computing the transpose of G represented with adjacency-lists is described below.

AdjacencyList-Transpose(G)
1 Initialise a new graph G^T with $G^T.V = G.V$ and no edges.
2 // Populate adjacency-list for G^T
3 **for each** $u \in G.V$
4 **for each** $v \in G.Adj[u]$
5 List-Insert($G^T.Adj[v], u$)
6 **return** G^T

Executing line 1 takes $\Theta(|V|)$ while executing the for loop in lines 3–5 takes $\Theta(|E|)$ because the List-Insert operation takes $\Theta(1)$ time and the number of executions of line 5 is $|E| = \sum_{u \in V} |Adj[u]|$. Therefore the running time of the procedure AdjacencyList-Transpose is $\Theta(|V| + |E|)$.

An algorithm for computing the transpose of G represented with adjacency-matrix is described below.

AdjacencyMatrix-Transpose(G)
1 Initialise a new graph G^T with $G^T.V = G.V$
2 // Populate adjacency-matrix for G^T
3 **for** $i = 1$ **to** $|G.V|$
4 **for** $j = 1$ **to** $|G.V|$
5 $G^T.A[j, i] = G.A[i, j]$
6 **return** G^T

Executing line 1 takes $\Theta(|V|)$ while executing the for loop in lines 3–5 takes $\Theta(|V|^2)$. Therefore the running time of the procedure AdjacencyMatrix-Transpose is $\Theta(|V|^2)$.

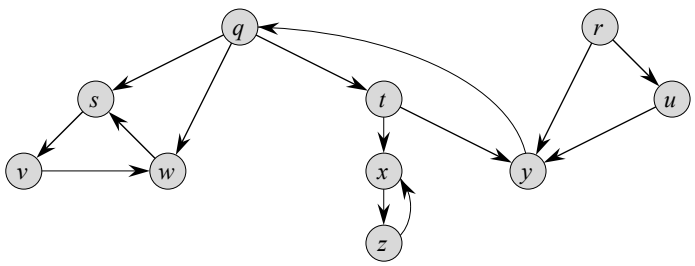
Exercise 2.
(TA: **Peter Vergerakis** `mailto:pverge15@student.aau.dk`)
The diameter of a tree $T = (V, E)$ is defined as $\max_{u,v \in V} \delta(u, v)$, that is, the largest of all shortest-path distances in the tree. Give an efficient algorithm to compute the diameter of a tree, and analyse the running time of your algorithm.

Solution 2.
The following algorithm is obtained by suitably modifying the pseudocode of the breadth-first search procedure BFS (see CLRS pp. 595) and it is assumed to be run given the root of the tree. Specifically, DIAMETER(T, s) closely follows the implementation for BFS(T, s) which explores the graph G stating from a source vertex $s \in V$ visiting all vertices that are reachable from s. While doing so, it updates the variable $diam_s$ satisfying the following loop invariant: before each iteration of the loop in lines 11–20 the value of $diam_s$ is greater than or equal to the attribute $u.d$ of any black vertex u. Analogously to the correctness proof for BFS one can can show that DIAMETER(T, s) returns the distance value $\delta(s, u)$ of a vertex that is at maximal distance from s, which corresponds to the diameter of T.

```
DIAMETER-AUX(T, s)
 1  for each v ∈ T.V \ {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  diam_s = ∞
 9  Q = ∅
10  ENQUEUE(Q, s)
11  while Q ≠ ∅
12       u = DEQUEUE(Q)
13       for each v ∈ T.Adj[u]
14            if v.color == WHITE
15                 v.color = GRAY
16                 v.d = u.d + 1
17                 v.π = u
18                 ENQUEUE(Q, v)
19       u.color = BLACK
20       diam_s = u.d
21  return diam_s
```

The running time of DIAMETER(T, s) is $\Theta(|V| + |E|)$, which can be shown using the same argument used in Lecture 10 for BFS.

Exercise 3.
(TA: **Rasmus Grønkjær Tollund** `mailto:rtollu18@student.aau.dk`)
Consider the graph G depicted below.

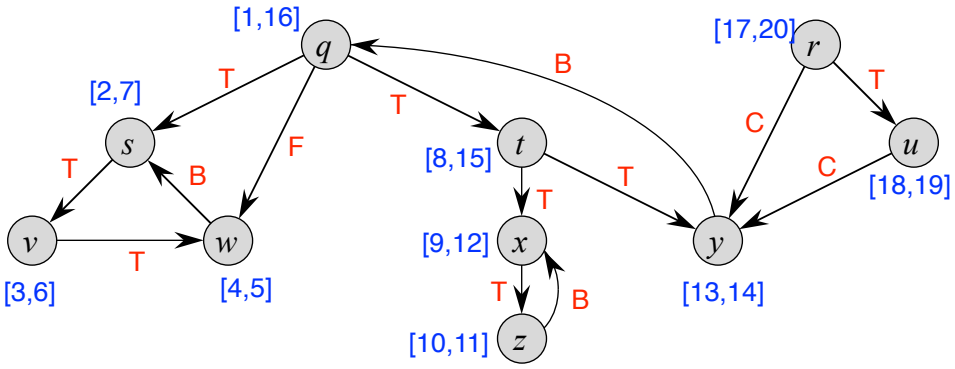


(a) Write the intervals for the discovery time and finishing time of each vertex in the graph obtained by performing a depth-first search visit of G.

(b) Write the corresponding "parenthesization" of the vertices in the sense of Theorem 22.7 in CLRS

(c) Assign with each edge a label T (tree edge), B (back edge), F (forward edge), C (cross edge) corresponding to the classification of edges induced by the DFS visit performed before.

Solution 3.
The figure below shows the intervals for discovery time and finishing time for each vertex (in blue) and the classification of edges corresponding to that specific visit. Note that another choice for the order of visit of vertices may yield a different outcome.



The "parenthesization" corresponding to the DFS visit described in the figure above is

$$(q\,(s\,(v\,(w\,w)\,v)\,s)\,(t\,(x\,(z\,z)\,x)\,(y\,y)\,t)\,q)\,(r\,(u\,u)\,r)\,.$$

Exercise 4.
(TA: **Alessandro Tibo** `mailto:alessandro@cs.aau.dk`)
[CLRS 22.4-5] Another way to perform topological sorting on a directed acyclic graph $G = (V, E)$ is to repeatedly find a vertex of in-degree 0, output it, and remove it and all of its outgoing edges from the graph. Explain how to implement this idea so that it runs in time $O(|V| + |E|)$. What happens to this algorithm if G has cycles?

Solution 4.
The pseudocode implementing the algorithm described above is the following.

```
TOPOLOGICAL-SORT(G)
 1  create the array in-degree[1..|V|]
 2  compute G^T
 3  for each v ∈ V
 4       in-degree[v] = |G^T.Adj[v]|
 5  Q = ∅
 6  for each v ∈ V such that in-degree[v] == 0
 7       ENQUEUE(Q, v)
 8  while Q ≠ ∅
 9       u = DEQUEUE(Q)
10       print u
11       for each v ∈ G.Adj[u]
12            in-degree[v] = in-degree[v] − 1
13            if in-degree[v] == 0
14                 ENQUEUE(Q, v)
```

To find vertices of in-degree 0 we compute the transpose of G then we store the length of each adjacency list in the array *in-degree*. From line 5 we start collecting in a queue Q all vertices having in-degree equal to 0. The while loop in lines 8–14 picks at every iteration a vertex which has in-degree 0 and "deletes" it form the graph by updating the array *in-degree* as if all its outgoing edges where removed. If during this "deletion" some vertex tuns out to have in-degree equal to 0, then the vertex is enqueued in Q (line 14). Notice that vertices which have already in-degree equal to 0 will never be updated again because no other vertex u has one vertex $v \in Adj[u]$ such that $v \in Q$. The use of a queue Q

for storing vertices ensures that they will processed in the same order as they become vertices with 0 in-degree. This guarantees that the printing order of the vertices is correct.
Running time Analysis. Computing the transpose of G takes $\Theta(|V|+|E|)$. The for loop in lines 3–4 takes time $\Theta(\sum_{v\in V}|G^T.Adj[v]|) = \Theta(|E^T|) = \Theta(|E|))$. The initialisation of the queue (lines 5–7) takes $\Theta(|V|)$. As for the while loop in lines 8–14 notice that each vertex enters in the queue at most once and updating the in-degree array is performed on for each vertex in the adjacency list $Adj[u]$ of the vertex u which has been just picked out form Q. Hence, the execution of the while loop takes $O(|V|+|E|)$. Summing up we have that Topological-Sort(G) runs in time $O(|V|+|E|)$.

If the are no cycles, all vertices will be eventually inserted in Q and printed out. If there are cycles, not all vertices will be printed, because some in-degrees never become 0.

Exercise 5.

(TA: **Danny Bøgsted Poulsen** `mailto:dannybpoulsen@cs.aau.dk`)
Assume G is a directed acyclic graph. Give an efficient algorithm to compute the graph of strongly connected components of G, and analyse the running time of your algorithm.

Solution 5.

By definition of strongly connected component two distinct vertices $u, v \in V$ belong to the same component if there is a cycle containing both u and v. Since G is assumed to be a DAG each component will be a singleton containing exactly one vertex. Therefore, returning the graph G will suffice which takes $\Theta(1)$ time.

Exercise 6.

(TA: **Danny Bøgsted Poulsen** `mailto:dannybpoulsen@cs.aau.dk`)
[CLRS 22.5-1] How can the number of strongly connected components of a graph change if a new edge is added?

Solution 6.

Consider the graph of $G = (V, E)$ and one pair of vertices $(u, v) \notin E$ which is added to G. There are tree cases

1. if both u and v belong to the same component then the SCCs after the insertion of (u, v) do not change

2. if u and v belong to two distinct components, say C_u and C_v and the component graph has already an edge (C_u, C_v), then the SCCs after the insertion of (u, v) in G do not change.

3. if u and v belongs to two distinct components, say C_u and C_v and (C_u, C_v) is not an edge of the component graph, then after the insertion of (u, v) the two components will merge forming the SCC $C_u \cup C_v$. All other components remain the same.

Exercise Session 11

Instructions. Each exercise is associated with a teaching assistant who is responsible to provide feedback for that exercise. You can ask for feedback by sending your solution by email or by asking questions by video chat using Microsoft Teams (`https://www.its.aau.dk/vejledninger/microsoft-teams/`) You can login to Teams by using your AAU email and password.
The TAs will be available in private chat from 8:15 until 10:00, and will answer all the emails sent from 8:15 to 12:00. Remember to **contact the TA by chat before starting a video chat** to be sure that the TA is ready to pick the call.
Beware that the response time may be slow. In that case, try to solve the following exercises while waiting for one TA to be available.

Exercise 1.
(TA: **Niels Tobias Nørgaard Svendsen** `mailto:ntns15@student.aau.dk`)
Consider a weighted directed graph $G = (V, E)$ with nonnegative weight function $w\colon E \to \mathbb{N}$. Solve the following computational problems assuming you can solve the single-source shortest-paths problem using e.g., Dijkstra's algorithm or the Bellman-Ford algorithm. Analyse the running time of your solutions.

Single-destination shortest-paths problem: Find a shortest path to a given destination vertex t from each vertex $v \in V$.

Single-pair shortest-path problem: Find a shortest path from u to v for given vertices u and v.

All-pairs shortest-paths problem: Find a shortest path from u to v for every pair of vertices u and v.

Solution 1.
Single-destination shortest-paths problem: It can be solved by solving the single-source shortest-paths problem on the transpose of G, denoted by G^T, with source vertex t. We have already seen that computing G^T takes $O(V + E)$. Thus, using Dijkstra's algorithm[1] we can solve the single-destination shortest-paths problem in $O(V + E) + O(V^2) = O(V^2)$. Whereas, using the Bellman-Ford algorithm, it will take $O(V + E) + O(VE) = O(VE)$.

Single-pair shortest-path problem: It can be solved by simply running a single-source shortest-paths algorithm with u as the source. Then we can print the shortest path from u to v by running PRINT-PATH(G, u, v). Again, using using Dijkstra's algorithm the resulting running time is $O(V^2)$. If instead we use the Bellman-Ford algorithm, the resulting running time is $O(VE)$.

All-pairs shortest-paths problem: It can be solved by running a single-source shortest-paths algorithm $|V|$ times, once for each vertex as the source. Since all edge weights are nonnegative, we can use Dijkstra?s algorithm. Thus the running time is $O(V^3)$. If instead we use the Bellman-Ford algorithm, the resulting running time is $O(V^2E)$.

Exercise 2.
(TA: **Peter Vergerakis** `mailto:pverge15@student.aau.dk`)
Consider a weighted tree $T = (V, E)$ having weight function $w\colon E \to \mathbb{R}$. The diameter of T is defined as $\max_{u,v \in V} \delta(u, v)$, that is, the largest of all shortest-path distances in the tree. Give an efficient algorithm to compute the diameter of a tree, and analyse the running time of your algorithm.

[1] Here we assume the min-priority queue to be implemented using an array, as described in CLRS Section 24.3.

Solution 2.
Assume the tree is rooted in s. We can find the diameter of the weighted tree by first solving the single-shortest paths problem from s, then determine the maximal shortest path estimate $v.d$ – which at this point is equal to $\delta(s, v)$ – for v ranging in V. Note that a tree is a directed acyclic graph, therefore we can use DAG-SHORTEST-PATH(T, w, s) for solving the single-shortest paths problem.

```
WEIGHTED-DIAM(T, w)
1  let s be the root of T
2  DAG-SHORTEST-PATH(T, w, s)
3  diam = 0
4  for each v ∈ T.V
5        diam = max(diam, v.d)
6  return diam
```

Running DAG-SHORTEST-PATH(T, w, s) takes $O(|V|+|E|)$ and the for loop in lines 4–5 takeds $O(|V|)$. Hence the running time of WEIGHTED-DIAM(T, w) is $O(|V| + |E|)$.

Exercise 3.
(TA: **Rasmus Grønkjær Tollund** `mailto:rtollu18@student.aau.dk`)
[CLRS 24.5-4] Let $G = (V, E)$ be a weighted, directed graph with source vertex s, and let G be initialised by INITAILIZE-SINGLE-SOURCE(G, s). Prove that if a sequence of relaxation steps sets $s.\pi$ to a non-NIL value, then G contains a negative-weight cycle.

Solution 3.
Whenever RELAX sets π for some vertex, it also reduces the vertex's d value. Thus if $s.\pi$ gets set to a non-NIL value, $s.d$ is reduced from its initial value of 0 to a negative number. But $s.d$ is the weight of some path from s to s, which is a cycle including s. Thus, there is a negative-weight cycle.

Exercise 4.
(TA: **Alessandro Tibo** `mailto:alessandro@cs.aau.dk`)
[CLRS 24.3-3] Consider the pseudocode for Dijkstra?s algorithm presented in CLRS pp. Suppose we change guard of the while loop in line 4 as $|Q| > 1$. This causes the while loop to execute $|V| - 1$ times instead of $|V|$ times. Is this proposed algorithm still correct?

Solution 4.
Yes, the algorithm still works. Let u be the leftover vertex that does not get extracted from the priority queue Q. If u is not reachable from s, then by the no-path property (Corollary 24.12) $d[u] = \delta(s, u) = \infty$. If u is reachable from s, there is a shortest path $p = s \rightsquigarrow x \rightarrow u$. When the vertex x was extracted, $d[x] = \delta(s, x)$ and then the edge (x, u) was relaxed; thus, by the convergence property (Lemma 24.14) $d[u] = \delta(s, u)$.

Exercise 5.
(TA: **Danny Bøgsted Poulsen** `mailto:dannybpoulsen@cs.aau.dk`)
[CLRS 24-3] Arbitrage is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that 1 U.S. dollar buys 49 Indian rupees, 1 Indian rupee buys 2 Japanese yen, and 1 Japanese yen buys 0.0107 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy $49 \cdot 2 \cdot 0,0107 = 1.0486$ U.S. dollars, thus turning a profit of 4.86 percent. Suppose that we are given n currencies $c_1, c_2, \ldots c_n$ and an $n \times n$ table R of exchange rates, such that one unit of currency c_i buys $R[i, j]$ units of currency c_j.

(a) Give an efficient algorithm to determine whether or not there exists a sequence of currencies $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} \rangle$ such that $R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_k, i_1] > 1$. Analyse the running time of your algorithm.

(b) Give an efficient algorithm to print out such a sequence if one exists. Analyse the running time of your algorithm.

Solution 5.

(a) Let $G = (V, E)$ be a weighted graph with $V = \{c_1, \ldots, c_n\}$ and weight function

$$w(c_i, c_j) = -\ln R[i, j]\,. \tag{1}$$

Notice that for a sequence $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k}\rangle$ we have that $R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_k, i_1] > 1$ iff $\ln R[i_1, i_2] + \ln R[i_2, i_3] + \cdots + \ln R[i_k, i_1] > 0$ (by applying the logarithm on both sides). Taking the negative logarithm this becomes

$$(-\ln R[i_1, i_2]) + (-\ln R[i_2, i_3]) + \cdots + (-\ln R[i_k, i_1]) < 0 \tag{2}$$

The above inequality holds when $p = \langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} c_{i_1}\rangle$ corresponds to a negative cycle in G. Therefore we can conclude that if we can find a negative cycle in G, then we can conclude there exists an opportunity for currency arbitrage.

In this particular problem we can assume that any two vertices u and v are connected by some edge, i.e., $E = V \times V$. This means that any vertex can reach any other with some path, and in particular any negative cycle can be reached starting from any source.

For this reason, the Bellman-Ford algorithm can be used to easily detect negative weight cycles in $O(VE)$ time stating from a source vertex arbitrarily chosen from V.

The algorithm described above is summarised by the following pseudocode.

```
ARIBITRAGE(R)
1  Construct G with weight function w defined as in Eq. (1)
2  pick some v ∈ G.V
3  return ¬BELLMAN-FORD(G, w, v)
```

The worst-case running time of the above algorithm is $O(V^3)$ because we are calling one instance of the Bellman-Ford algorithm algorithm —which runs in $O(VE)$ time— on a graph with n vertices and n^2 edges.

(b) Recall that BELLMAN-FORD, after relaxing all edged $|V| - 1$ times, all estimates represent the weight of a shortest path from the source in case there are no negative cycles. The last loop checks if there is still opportunity to "shorten" a path (i.e., decrease the shortest path weight estimate) by relaxing another edge. If this is the case for some edge $(u, v) \in E$, then we know, that there is a negative cycle from $u \to v \rightsquigarrow u$. To print such a cycle it suffices to print the the path $v \rightsquigarrow u$ induced by the predecessor graph, that is, call PRINT-PATH(G, v, u).

```
PRINT-ARBITRAGE(R)
1  Construct G with weight function w defined as in Eq. (1)
2  pick some v ∈ G.V
3  if ¬BELLMAN-FORD(G, w, v)
4       determine the edge (u, v) ∈ E such that v.d > u.d + w(u, v)
5       PRINT-PATH(G, v, u)
6  else
7       return FALSE
```

Notice that If a cycle is found it will have at most $|V|$ edges, thus executing line 5 takes $O(n)$ time. Line 4 can be integrated in the BELLMAN-FORD procedure at an additional cost $\Theta(1)$. Therefore, the worst-case running time of PRINT-ARIBITRAGE(R) is $O(n^3)$.

Exercise 1

[CLRS 25.1-2] Why do we require that $w_{ii} = 0$ for all $1 \leq i \leq n$?

Exercise 2

Consider a weighted directed graph $G = (V, E)$ representing the map of a city where the weight of each edge $(v_i, v_j) \in E$ represents the distance from location v_i to v_j. The subset of vertices $B \subset V$ represents all bars in the city, and the remaining locations are denoted by $H = (V \setminus B)$. Assume that:

- The graph is complete, i.e., there exists an edge between each vertex $(v_i, v_j) \in V \times V$.
- The graph has negative weights, but no negative cycles.

1. Describe an algorithm that computes the **length** of the shortest route between all pairs of vertices $(v_i, v_j) \in H \times H$, with the requirement that all routes must go via **exactly one** of the bars in B. I.e., the shortest path from v_i to v_j with exactly **one** intermediate vertex in B. Analyse the running time of your algorithm.
2. You decide to go on a bar-crawl. Describe an algorithm which computes and returns a **route** (e.g. as an array A containing the order of vertices), which allows you to visit **all** of the bars in B **exactly once**. Note that the route via **all** bars does **not** need to be the shortest possible route. However, it should be a shorter route than simply going to each bar in the ordering denoted by the vertex indexation (we assume this order to be random). Suggest at least one heuristic which should give you a shorter path than this random ordering. Analyse the running time of your algorithm.

1. This is almost exactly the same as an exercise from the exercise session ("Rick and Morty"). One solution is to run Floyd-Warshall first (on line 1), and then continue as described:

```
OneBar(H, B)
1  D = Floyd-Warshall(H)
2  D' = ∞ matrix
3  for each v_i ∈ H
4       for each v_j ∈ H
5            for each b ∈ B
6                 d'_ij = min(d'_ij, d_ik + d_kj)
7
8  return D'
```

   This is $\Theta(|V|^3)$

2. This one gives some room for interpretation. The simplest would be a linear solution by vertex ordering (which would not give points, as indicated in the text), such as:

```
BarCrawl(B)
1  let A be an array representing our path
2  for each v_i ∈ B
3       A[i] = v_i
4  return A
```

   A different straight-forward / greedy solution, would be

BarCrawl(B)

```
1  let A be an array representing our path
2  let i = 0
3  A[i] = B[i]
4  while B ≠ ∅
5        for each v_j ∈ B
6              if d(v_i, v_j) < d(v_i, A[i + 1])
7                    A[i + 1] = v_j
8        B = B ∖ A[i + 1]
9        i = i + 1
10 return A
```

Which should be $\Theta(|B|^2)$.