Re-Exam - August 2021

Algorithms and Data Structures

Instructions. This exam consists of **five questions** and you have time until 13:00 to submit your solution in digital exam. You can answer the questions directly on this paper, or use additional sheets of paper which have to be hand-in as a **single pdf file**. You are encouraged to mark the multiple choice answers as well as the labelling of graphs directly in this exam sheet.

- Before starting solving the questions, read carefully the exam guidelines at `https://www.moodle.aau.dk/mod/page/view.php?id=1173709`.
- Read carefully the text of each exercise. Pay particular attention to the terms in bold.
- **CLRS** refers to the textbook T.H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (3rd edition).
- You are allowed to refer to results in the textbook as well as exercise or self-study solutions posted in Moodle to support some arguments used in your answers.
- Make an effort to use a readable handwriting and to present your solutions neatly.
- Before submission, check that your answer sheet is readable by a standard pdf-viewer (e.g., Adobe Reader, Preview, etc.)

Question 1. 15 Pts

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

(1.1) [5 Pts] Mark **ALL** the correct answers. $n^2\sqrt{n} + n^2 \lg n^5 + n \lg 100^n$ is

☐ **a)** $\Theta(n^2 \lg n)$ ☐ **b)** $\Theta(n^2)$ ☐ **c)** $\Theta(n^{2.5})$ ☐ **d)** $\Theta(n \lg n)$ ☐ **e)** $\Theta(n)$

(1.2) [5 Pts] Mark **ALL** the correct answers. $n^2\sqrt{n} + n \log_3 9^n + n \lg n^n$ is

☐ **a)** $\Theta(n^5)$ ☐ **b)** $\Omega(n)$ ☐ **c)** $\Theta(n^{2.5})$ ☐ **d)** $\Omega(\sqrt{n})$ ☐ **e)** $O(n^5)$

(1.3) [5 Pts] Mark **ALL** the correct answers. $2^n + n^n + \frac{1}{200} n \lg n^n + n^{1000}$ is:

☐ **a)** $\Omega(n \lg n)$ ☐ **b)** $\Omega(n^{999})$ ☐ **c)** $O(2^n)$ ☐ **d)** $\Omega(n^2 \lg n)$ ☐ **e)** $O(n^2 \lg n)$

Solution 1.

(1.1)

$$n^2\sqrt{n} + n^2 \lg n^5 + n \lg 100^n = n^{2.5} + 5n^2 \lg n + n^2 \lg 100 = \Theta(n^{2.5})$$

Therefore **c** is the only correct answer

(1.2)

$$n^2\sqrt{n} + n \log_3 9^n + n \lg n^n = n^{2.5} + 2n^2 + n^2 \lg n = \Theta(n^{2.5})$$

Therefore **b**, **c**, **d**, and **e** are correct.

(1.3)

$$2^n + n^n + \frac{1}{200} n \lg n^n + n^{1000} = 2^n + n^n + \frac{1}{200} n^2 \lg n + n^{1000} = \Theta(n^n)$$

Therefore **a**, **b**, and **d** are correct.

Question 2. 20 Pts

Consider the following recurrences

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n/2) + T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases} \qquad Q(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot Q(n-1) & \text{if } n > 0 \end{cases}$$

Answer the questions below concerning these two recurrences. For each question, pay close attention to whether it concerns $Q(n)$ or $T(n)$.

(2.1) [5 Pts] Mark **ALL** correct answers.

- ☐ **a)** $T(n)$ can be solved using Case 1 of the Master Theorem
- ☐ **b)** $T(n)$ can be solved using Case 2 of the Master Theorem
- ☐ **c)** $T(n)$ can be solved using Case 3 of the Master Theorem
- ☐ **d)** $T(n)$ cannot be solved using the Master Theorem

(2.2) [5 Pts] Mark **ALL** correct answers.

- ☐ **a)** $T(n) = \Theta(n^2)$
- ☐ **b)** $T(n) = O(n \lg n)$
- ☐ **c)** $T(n) = O(2^n)$
- ☐ **d)** $T(n) = \Omega(n^2)$

(2.3) [10 Pts] Prove that $\lg Q(n) = O(n \lg n)$ using the substitution method.

Solution 2.

(2.1) Note that the recurrence is of the form $T(n) = aT(n/b) + f(n)$ where $a = 2$, $b = 2$, and $f(n) = n$. We can solve the recurrence using the master method. This recurrence falls into the second case, because $f(n) = \Omega(n) = \Omega(n^{\log_b a})$. Therefore, by the Master Theorem (Case 2) we can conclude that $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n \lg n)$.

Therefore, the only correct answer is **b**.

(2.2) We have seen that $T(n) = \Theta(n \lg n)$. Therefore the correct answers are **b** and **c**.

(2.3) Note that we have already seen a similar recurrence in Exercise Session 4, namely Exercise 4.c, indeed $Q(n)$ is exactly the factorial of n, more commonly denoted as $n!$. In what follows we recall the proof given in Exercise Session 4.

To prove that $\lg n! = O(n \lg n)$ it suffices to show that for all $n \geq 1$, $\lg n! \leq n \lg n$ for some suitable constant $c > 0$.

Base Case ($n = 1$). $\lg n! = \lg 1 = 0 = n \lg n$. Thus, for $n = 1$, $\lg n! \leq cn \lg n$ holds for any $c > 0$.

Inductive Step ($n > 1$). We have that

$$\begin{aligned} \lg n! &= \lg(n \cdot (n-1)!) && \text{(def. factorial)} \\ &= \lg n + \lg((n-1)!) \\ &\leq \lg n + c(n-1)\lg(n-1) && \text{(inductive hypothesis)} \\ &\leq \lg n + c(n-1)\lg n && \text{(lg is monotone)} \\ &\leq c \lg n + c(n-1)\lg n && \text{(assuming } c \geq 1) \\ &= cn \lg n \end{aligned}$$

Thus, for $c \geq 1$ we have that $\lg n! \leq cn \lg n$ for all $n \geq 1$ from which we conclude $\lg n! = O(n \lg n)$.

Question 3. 25 Pts

Understanding of known algorithms.

(3.1) [5 Pts] Mark **ALL** the correct statements.

- ☐ **a) In the worst case** MergeSort runs faster than QuickSort
- ☐ **b)** MergeSort runs **always** faster than InsertionSort
- ☐ **c)** One can decide in time $\Theta(V + E)$ if the graph $G = (V, E)$ is acyclic or not
- ☐ **d)** The Bellman-Ford algorithm returns False when the graph $G = (V, E)$ has a cycle
- ☐ **e)** Quicksort and MergeSort work in-place.

(3.2) [4 Pts] Mark **ALL** the correct statements. Consider the array $A = [5, 3, 6, 0, 4, 2]$ and assume that $A.\textit{heap-size} = A.\textit{length}$.

- ☐ **a)** A satisfies the max-heap property
- ☐ **b)** The binary tree interpretation of A satisfies the binary search tree property
- ☐ **c)** The result of Max-Heapify$(A, 1)$ is $[6, 4, 5, 0, 3, 2]$
- ☐ **d)** The pair $(A, 1)$ is not a valid instance of Max-Heapify

(3.3) [6 Pts] Consider the hash table $H = 99, 100, 65, \text{Nil}, \text{Nil}, \text{Nil}, \text{Nil}, 61, \text{Nil}$. Insert the keys $87, 74, 62$ in H using *open addressing* with the auxiliary function $h'(k) = k$.

Mark the hash table resulting by the insertion of these keys using **linear** probing.

- ☐ **a)** $99, 100, 65, \text{Nil}, \text{Nil}, 74, 87, 61, 62$
- ☐ **b)** $99, 100, 65, 74, \text{Nil}, \text{Nil}, 87, 61, 62$
- ☐ **c)** $99, 100, 65, \text{Nil}, \text{Nil}, 62, 87, 61, 74$
- ☐ **d)** none of the above

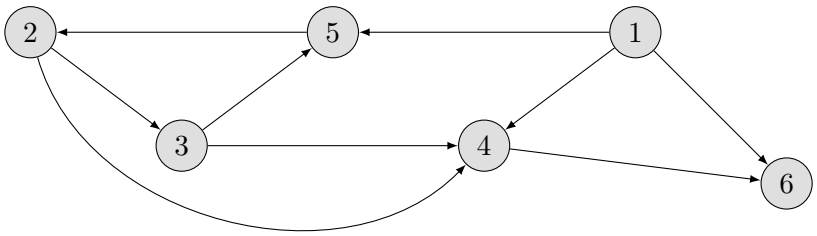
Mark the hash table resulting by the insertion of these keys using **quadratic** probing with $c_1 = 2$ and $c_2 = 4$.

- ☐ **a)** $99, 100, 65, \text{Nil}, \text{Nil}, 74, 87, 61, 62$
- ☐ **b)** $99, 100, 65, 74, \text{Nil}, \text{Nil}, 87, 61, 62$
- ☐ **c)** $99, 100, 65, \text{Nil}, \text{Nil}, 62, 87, 61, 74$
- ☐ **d)** none of the above

Mark the hash table resulting by the insertion of these keys using **double hashing** with $h_1(k) = k$ and $h_2(k) = 1 + (k \mod (m - 1))$.

- ☐ **a)** $99, 100, 65, \text{Nil}, \text{Nil}, 74, 87, 61, 62$
- ☐ **b)** $99, 100, 65, 74, \text{Nil}, \text{Nil}, 87, 61, 62$
- ☐ **c)** $99, 100, 65, \text{Nil}, \text{Nil}, 62, 87, 61, 74$
- ☐ **d)** none of the above

(3.4) [10 Pts] Consider the directed 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 (see CLRS sec.22.3).
Remark: If more than one vertex can be chosen, choose the one with smallest label.

(b) Mark the corresponding "parenthesization" of the vertices in the sense of CLRS Theorem 22.7 resulting from the DFS visit performed before

☐ **a)** (1 (5 (2 (4 (6 6) 4) 2) (3 3) 5) 1) ☐ **b)** (1 (4 (6 6) 4) (5 (2 2) (3 3) 5) 1)

☐ **c)** (1 (4 (6 6) 4) (5 (2 (3 3) 2) 5) 1) ☐ **d)** none of the above

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

(d) If G admits a topological sorting, then show the result of TOPOLOGICAL-SORT(G) (see CLRS sec.22.4). If it doesn't admit a topological sorting, briefly argue why.

Solution 3.

(3.1) (a) The statement is correct because, in the worst case, MERGESORT runs in $\Theta(n \lg n)$ whereas QUICKSORT runs in $\Theta(n^2)$.

(b) The statement is not correct because, when the input array is already sorted, INSERTION-SORT runs in $\Theta(n)$ (this is the best case), whereas MERGESORT runs in $\Theta(n \lg n)$.

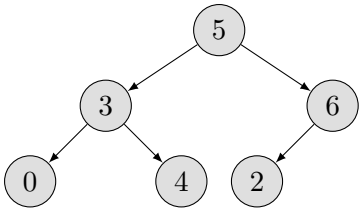
(c) The statement is correct because, one can decide if a graph is acyclic by running a DFS visit and check if no edges have been classified as *black* (see Theorem). Recall that the running time of the DFS procedure is $\Theta(V + E)$.

(d) The statement is not correct for two reasons. First, the BELLMAN-FORD algorithm requires as input a weighted graph and a source node; second, the algorithm returns FALSE only if a **negative** cycle is found.

(e) The statement is not correct, because QUICKSORT works in place but MERGESORT doesn't.

(3.2) (a) The statement is not correct because the root of the heap is 5 but the maximal element of the array is 6.

(b) The statement is not correct. Indeed, the binary tree interpretation of A is



The fact that node 2 belongs to the right subtree of 5 violates the binary search tree property.

(c) The statement is not correct, because $(A, 1)$ is not a valid instance of MAXHEAPIFY.

(d) The statement is correct, because MAXHEAPIFY requires that both the left and the right subtree of the given index —here is 1— satisfy the max-heap property. As one can see from the picture above, only the right subtree satisfies the max-heap property.

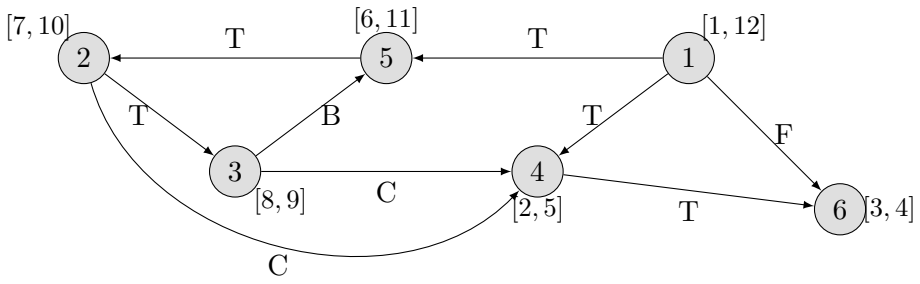
(3.3) The resulting hash tables are respectively:

linear probing: $99, 100, 65, 74, \text{NIL}, \text{NIL}, 87, 61, 62$

quadratic probing: $99, 100, 65, \text{NIL}, \text{NIL}, 62, 87, 61, 74$

double hashing: $99, 100, 65, \text{NIL}, \text{NIL}, 74, 87, 61, 62$

(3.4) The correct answers for (a) and (c) are depicted in the graph below. There, each vertex $v \in V$ is associated with the interval $[v.d, v.f]$ as computed by DFS, and each edge is labelled according to the corresponding classification.



(b) The correct answer is **c**.

(d) The graph is not acyclic, therefore it does not admit topological sorting. The fact that the graph has a cycle is in line with the presence of the *back* edge $3 \to 5$ in the DFS classification, which spots out the cycle $5 \to 2 \to 3 \to 5$.

Question 4. 20 Pts

Asymptotic runtime analysis.

(a) Prof. Algo wrote the following algorithm which takes two arrays of numbers $A[1\,..\,n]$ and $B[1\,..\,m]$ **sorted in non-decreasing order** possibly having repeated elements and checks whether the set of elements in A is equal to the set of elements in B.

```
EQUAL(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
```

(a.1) [5 Pts] Perform an asymptotic analysis of the **worst-case** running time of EQUAL(A, B). Motivate your answer.

(a.2) [5 Pts] Perform an asymptotic analysis of the **best-case** running time of EQUAL(A, B). Motivate your answer.

(b) [10 Pts] Prof. Algo discovered another way to perform topological sorting on a directed acyclic graph $G = (V, E)$. The intuition behind the algorithm goes as follows: it repeatedly finds a vertex of in-degree 0, prints it, and deletes it together with all its outgoing edges from the graph.

```
ALGOTOPOLOGICAL-SORT(G)
 1  Create the array in-degree[1 .. |V|]
 2  Let G^T be the transpose of G
 3  for each v ∈ V
 4       in-degree[v] = |G^T.Adj[v]|
 5  Let Q = ∅ be an empty queue
 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)
```

Perform an asymptotic analysis of the worst-case run-time of ALGOTOPOLOGICAL-SORT(G). Motivate your answer.

Solution 4.

(a) We have seen this algorithm in Exercise Session 1. Here we recall the exact worst-case run-time analysis done back then, however, the exercise can be solved more briefly than what follows since the question asks for an **asymptotic** analysis.

How the algorithm works. 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 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, that are the length of the two arrays respectively. Hence we express the running time of EQUAL as a function in two arguments, namely $T(n, m)$.

Worst-case run-time analysis. The worst-case running time occurs when the two arrays are equal, that is

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

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. Therefore, using the usual notation that c_i denotes the constant time required to execute line i, we have that

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

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.

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

Best-case run-time analysis. The best-case runtime analysis occurs when the two arrays differ in their first element, that is $a_1 \neq b_1$. In this case the outer while loop is skipped and the algorithms terminates in after a constant amount of steps, that is $T(n, m) = \Theta(1)$.

(b) We have solved this exercise in Exercise Session 10. Here we recall the worst-case runtime analysis performed back then.

Recall that the each operations ENQUEUE, DEQUEUE, and empty queue creation take each $\Theta(1)$ time (see CLRS pp. 235). Computing the transpose of G takes $\Theta(|V|+|E|)$ (see Exercise Session 10, Exercise 1). 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 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 $O(|V| + |E|)$.

Question 5. 20 Pts

Solving computational problems.

The organisers of Roskilde festival have shortlisted n candidate bands $B_1, \ldots, B_n$ to perform in next year festival. Each band B_i $(i = 1 \,..\, n)$ is associated to the expected revenue it generates in tickets sales, denoted by r_i, and the costs the organisers have to anticipate to hire the band, denoted by c_i. The organisers want to find an *affordable* selection $S \subseteq \{1, \ldots, n\}$ of bands to hire for the festival so that the *total expected revenue* $\sum_{i \in S} r_i$ is maximal. A selection S is affordable if the costs the organisation have to anticipate do not exceed the allocated budget C, that is $\sum_{i \in S} c_i \leq C$.

(a) [10 Pts] Describe a **bottom-up** dynamic programming procedure RoskildeRevenue(r, c, C) that takes as input the arrays $r[1 \,..\, n]$ and $c[1 \,..\, n]$, as well as the allocated budget $C > 0$ and returns the total expected revenue of an optimal selection of bands.

(b) [10 Pts] Describe a procedure RoskildeFlyer(r, c, C) that prints an optimal selection of bands in **non-decreasing order** of expected revenue.

Remarks:

- The description of the algorithmic procedures must be given **both** by providing the pseudocode and by explaining in detail how it works.
- Try to execute your algorithm on the following example. You may catch some errors you did not foresee while designing your algorithm.

  Example: Consider a problem instance with $c = [15, 20, 29]$, $r = [138, 100, 125]$ and $C = 50$. An optimal solution is the set of indices $S = \{1, 3\}$ with total expected revenue $138 + 125 = 263$ and the cost the organisation has to anticipate is $15 + 29 = 44 \leq 50$. Therefore RoskildeRevenue$(r, c, C) = 263$ and RoskildeFlyer(r, c, C) prints out B_3, B_1.

Solution 5.

(a) Note that this problem is an instance of the *knapsack problem* that we have solved in Exercise Session 9. Here the values r_i and c_i respectively correspond to the item value and the item weight, while the budget C corresponds to the maximal weight that knapsack can carry. We can simply reuse the procedure Knapsack from Exercise Session 9 to find the optimal expected revenue.

```
RoskildeRevenue(r, c, C)
1  (R, K) = Knapsack(r, c, C)
2  return R
```

(b) To print out an optimal selection in non-decreasing order of expected revenue we modify the procedure Print-Knapsack described in Exercise Session 9. When a band belonging to the optimal selection is found we store it into a binary search tree with associated *key* equal to its revenue. When the entire selection has been inserted in the tree we print its content by calling the Inorder-Tree-Walk procedure.

```
RoskildeFlyer(r, c, C)
 1  (R, K) = Knapsack(r, c, C)
 2  j = W
 3  n = c.length
 4  Let T be an empty binary search tree
 5  for i = n downto 1
 6       if K[i, j]
 7            B_i.key = r[i]
 8            Tree-Insert(T, B_i)
 9            j = j − c[i]
10  Inorder-Tree-Walk(T.root)
```