Algorithms and Data Structures (IBC027)
Solutions and Correction Guidelines

January 20, 2020

Whenever an algorithm is required (assignments 2-6), it can be described in pseudocode or plain English, and its **running time and correctness must always be justified. Your explanations may be informal but should always be clear and convincing**. *For problems where an algorithm must be defined (assignments 2-6), 20% of the points is for the correctness argument, and 20% for the analysis of the running time. The grade equals the sum of the scores for all 6 problems below (plus possibly a bonus score for the homework assignments). Success!*

General correction guideline Students may earn the maximal score for the complexity analysis of an algorithm that is incorrect.

1 Quiz (2 points)

Justify your answers.

Solutions: Questions have been taken from `https://www.geeksforgeeks.org/data-structure-gq/`. Each correct answer is worth 0.25pts. Deduct 0.1pts if no (sensible) justification is provided. Justification does not need to be complete. If justification clearly shows that student did not understand the problem, give 0pts.

Question 1 What is the worst case time complexity for search, insert and delete operations in a general binary search tree?

(A) $\mathcal{O}(n)$ for all

(B) $\mathcal{O}(\log n)$ for all

(C) $\mathcal{O}(\log n)$ for search and insert, and $\mathcal{O}(n)$ for delete

(D) $\mathcal{O}(\log n)$ for search, and $\mathcal{O}(n)$ for insert and delete

Solution: A. In skewed Binary Search Tree (BST), where every node has at most one child, all three operations can take $\mathcal{O}(n)$.

Question 2 The following numbers are inserted into an empty binary search tree in the given order: 10, 1, 3, 5, 15, 12, 16. What is the height of the binary search tree (the height is the maximum number of edges from a leaf node to the root)?

(A) 2

(B) 3

(C) 4

(D) 6

Solution: B. The constructed binary search tree will look as follows:

```
      10
     /     \
    1       15
    \      /  \
     3    12   16
       \
        5
```

0.1pts for students who give the right BST but the wrong answer.

Question 3 A data structure is required for storing a set of integers such that each of the following operations can be done in $\mathcal{O}(\log n)$ time, where n is the number of elements in the set.

1. Deletion of the smallest element
2. Insertion of an element if it is not already present in the set

Which of the following data structures can be used for this purpose?

(A) A heap can be used but not a balanced binary search tree

(B) A balanced binary search tree can be used but not a heap

(C) Both balanced binary search tree and heap can be used

(D) Neither balanced search tree nor heap can be used

Solution: B. A balanced binary search tree (like, for instance, an AVL-tree or a read black tree) containing n items allows the lookup, insertion, and removal of an item in $\mathcal{O}(\log n)$ worst-case time. Since its a BST, we can easily find the minimum element in $\mathcal{O}(\log n)$.

Since a heap is a balanced binary tree (or almost complete binary tree), insertion complexity for heap is $\mathcal{O}(\log n)$. Also complexity to get minimum in a min heap is $\mathcal{O}(\log n)$ because removal of root node causes a call to heapify (after removing the first element from the array) to maintain the heap tree property. But a heap cannot be used for the above purpose as the question says – insert an element if it is not already present. For a heap, we cannot find out in $\mathcal{O}(\log n)$ if an element is present or not.

Question 4 Which of the following is true about Red Black Trees?

(A) The path from the root to the furthest leaf is no more than twice as long as the path from the root to the nearest leaf

(B) At least one children of every black node is red

(C) Root may be red

(D) A leaf node may be red

Solution: A. Properties C and D directly violate the axioms of a Red Black tree. Property B is wrong since a Red Black tree may consist of just black nodes. Property A is true and has been proven in the lecture.

Question 5 A hash table of length 10 uses open addressing with hash function h(k)=k mod 10, and linear probing. After inserting 6 values into an empty hash table, the table is as shown below.

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

Which one of the following choices gives a possible order in which the key values could have been inserted in the table?

(A) 46, 42, 34, 52, 23, 33

(B) 34, 42, 23, 52, 33, 46

(C) 46, 34, 42, 23, 52, 33

(D) 42, 46, 33, 23, 34, 52

Solution: C. Answer (C) The sequence (A) doesnt create the hash table as the element 52 appears before 23 in this sequence. The sequence (B) doesnt create the hash table as the element 33 appears before 46 in this sequence. The sequence (C) creates the hash table as 42, 23 and 34 appear before 52 and 33, and 46 appears before 33. The sequence (D) doesnt create the hash table as the element 33 appears before 23 in this sequence.

Question 6 Consider a hash table with 100 slots. Collisions are resolved using chaining. Assuming simple uniform hashing, what is the probability that the first 3 slots are unfilled after the first 3 insertions?

(A) $(97 \times 97 \times 97)/10^6$

(B) $(99 \times 98 \times 97)/10^6$

(C) $(97 \times 96 \times 95)/10^6$

(D) $(97 \times 96 \times 95)/(3! \times 10^6)$

Solution: A. Simple Uniform hashing function is a hypothetical hashing function that evenly distributes items into the slots of a hash table. Moreover, each item to be hashed has an equal probability of being placed into a slot, regardless of the other elements already placed. The probability that the first insertion does not use any of the first 3 slots is 97/100. The probabilities that the second and third insertions do not use any of the first three slots is also 97/100. Thus the correct answer is $(97/100)^3$.

Question 7 Consider a binary max-heap implemented using an array. Which one of the following arrays represents a binary max-heap?

(A) 25,12,16,13,10,8,14

(B) 25,14,13,16,10,8,12

(C) 25,14,16,13,10,8,12

(D) 25,14,16,13,10,8,12

Solution: C. A tree is max-heap if the value at every node in the tree is greater than or equal to the values of its children.

In array representation of heap tree, a node at index i has its left child at index $2i$ and right child at index $2i+1$. For item (C) we get the following max heap, for the other items we obtain a tree that violates the max-heap property.

```
      25
    /   \
   /     \
  14      16
 / \      / \
/   \    /  \
13  10  8    12
```

Question 8 Which of the following data structures is best suited for efficient implementation of priority queue? (A) Array (B) Linked List (C) Heap (D) Stack

(A) Array

(B) Linked List

(C) Heap

(D) Stack

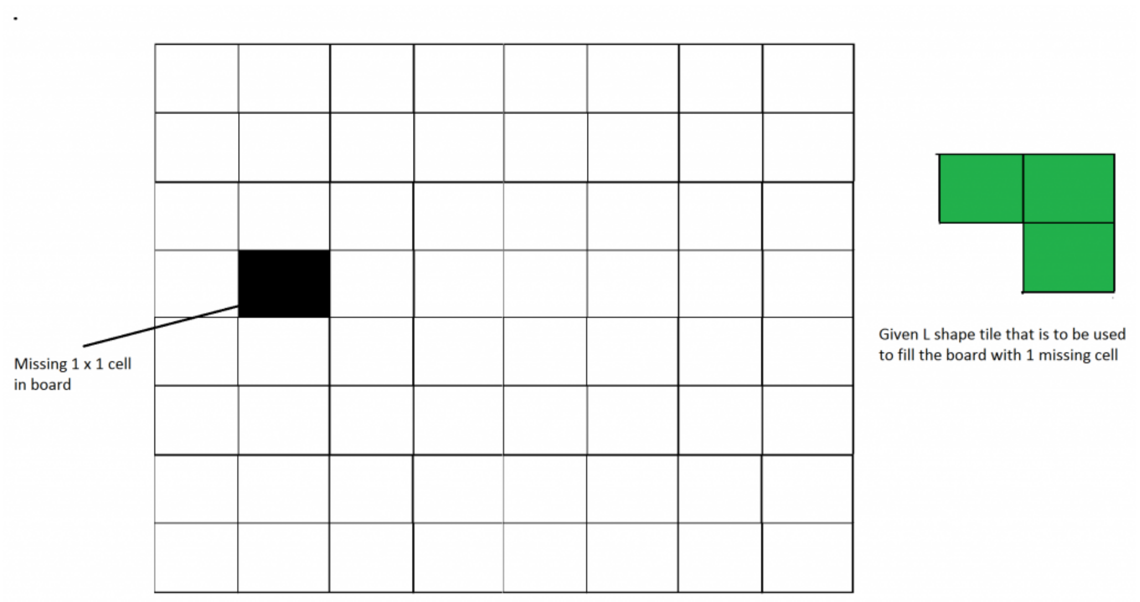
Solution: C. See lecture on heaps.

2 Divide and conquer (1.5 points)

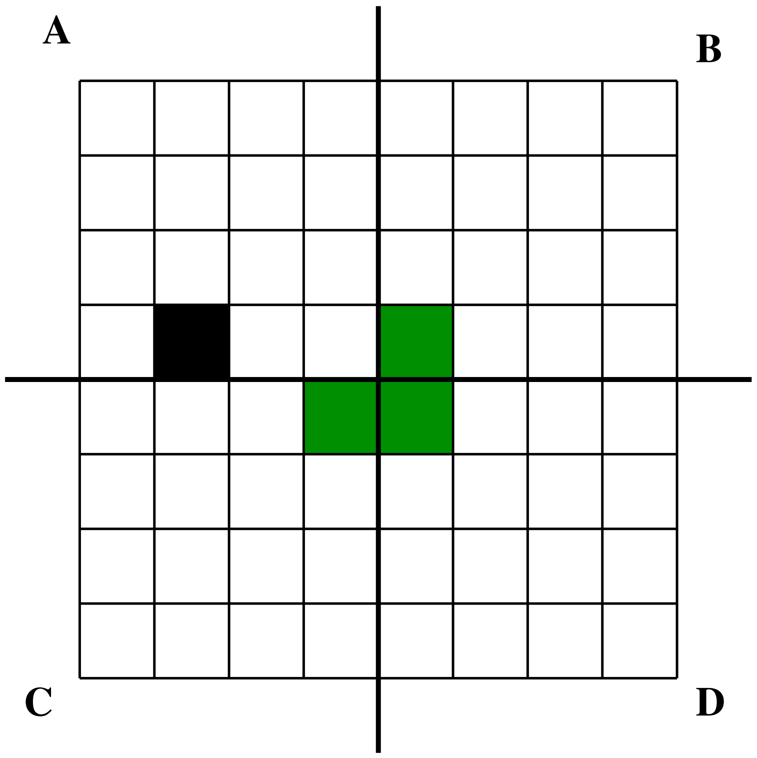
Let us consider an extremely simplified version of Tetris, with a unique simplified L-shaped tile and a board that consists of $2^n \times 2^n$ cells, for some positive integer n. The game is to place as many tiles (the shape is given below, they can be rotated) as possible on the board. We suppose that there is exactly 1 designated cell on the board that is "missing" and that cannot be covered. Describe and analyze a divide-and-conquer algorithm to fully cover the board (except for the missing cell) with L-tiles. The input of the algorithm consists of the value for n and the coordinates of the missing cell, and the output is a list of tiles, where each tile is represented by the three cells on the board that it covers. For instance, for input $(1,(2,2))$ the corresponding output is $(\{(1,1),(1,2),(2,1)\})$.

Solution: The L-shapes from this assignment are called *trominoes* in the literature. They were introduced in S. W. Golomb, *Checker Boards and Polyominoes*, Amer. Math. Monthly, 61(1954), 675–682. In this article, Solomon Golomb proved that deficient squares whose side length is a power of two can be tiled. Later, Golomb wrote a whole book *polyominoes*, the shapes formed by connecting some number of unit squares edge-to-edge. This book is "universally regarded as a classic in recreational mathematics" (see `https://en.wikipedia.org/wiki/Polyominoes:_Puzzles,_Patterns,_Problems,_and_Packings`). For more information on polyominoes we also refer to `https://www.ics.uci.edu/~eppstein/junkyard/polyomino`.

For the base case where $n=1$ the solution is easy: we just place a single tile on the three tiles of the board that are not "missing".


Missing 1 x 1 cell in board
Given L shape tile that is to be used to fill the board with 1 missing cell

Now suppose $n > 1$. We divide the board in 4 equal parts or "quadrants" of size $2^{n-1} \times 2^{n-1}$. The missing cell must occur in one of these 4 parts, and for this part we can solve the problem via a recursive call. Now we place a single tile at the very center of the original $2^n \times 2^n$ board in such a way that it covers a single cell of each of the 3 remaining parts. The idea is illustrated in the figure below. In this way we create a missing cell for each of the 3 remaining parts! This allows



us to solve the problem for these parts via recursive calls. Using the solutions for the four parts, we can easily construct a solution for the overall board: we just need to add 2^{n-1} to some x or y coordinates:

- We add 2^{n-1} to all y-coordinates of cells in quadrant A, moving them up.
- We add 2^{n-1} to both the x- and the y-coordinates of cells in quadrant B, moving them up and to the right.
- We add 2^{n-1} to the x-coordinates of cells in quadrant D, moving them to the right.

Correctness is straightforward (formally via induction on n).

As for the time complexity, we can find a constant $c > 0$ such that the following recursion equations hold for the total time $T(n)$ required to solve a board with parameter n:

$$\begin{aligned} T(1) &\leq 4 \cdot c \\ T(n) &\leq 4 \cdot T(n-1) + c \cdot 2^{2n} \quad \text{if } n > 1 \end{aligned}$$

The first equation holds since the amount of work to solve a board with $n = 1$ is constant. The second equation holds since in order to solve the problem for a board with $n > 1$, we first need to solve 4 times the problem for a board of size $n - 1$, and then we need to print out 3 cells for each of the tiles, that is, altogether $2^{2n} - 1$ cells. Using the substitution method we derive $T(n) = \mathcal{O}(n \cdot 2^{2n})$. For this, we prove by induction on n that $T(n) \leq c \cdot n \cdot 2^{2n}$.

1. Induction base. If $n = 1$ then
$$T(n) = T(1) \leq 4 \cdot c = c \cdot 1 \cdot 2^2 = c \cdot n \cdot 2^{2n}$$
2. Induction step. Suppose $n > 1$. Then
$$\begin{aligned} T(n) &\leq 4 \cdot T(n-1) + c \cdot 2^{2n} \qquad \text{(by definition } T(n)) \\ &\leq 4 \cdot c \cdot (n-1) \cdot 2^{2n-2} + c \cdot 2^{2n} \qquad \text{(by induction hypothesis)} \\ &= c \cdot (n-1) \cdot 2^{2n} + c \cdot 2^{2n} \\ &= c \cdot n \cdot 2^{2n} - c \cdot 2^{2n} + c \cdot 2^{2n} \\ &= c \cdot n \cdot 2^{2n} \end{aligned}$$

From this $T(n) = \mathcal{O}(n \cdot 2^{2n})$ follows immediately. If we write $N = 2^{2n}$ for the total number of cells on the board, then the resulting time complexity of our algorithm is $\mathcal{O}(N \log N)$. So although the divide and conquer algorithm is exponential in n, it is close to linear in the size (number of cells) of the board.

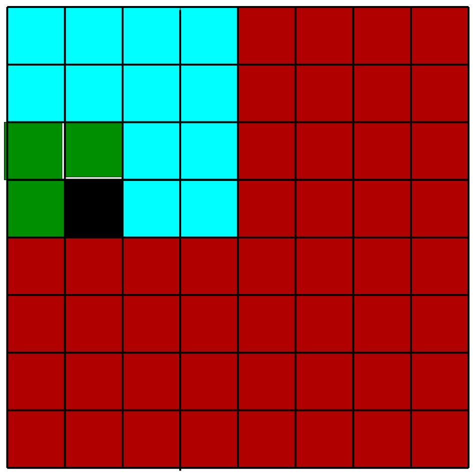
If students clearly describe the key ideas they get most of the points: (a) case $n = 1$ trivial (0.2pts), (b) divide board into 4 parts and add single tile in the middle (0.5pts), (c) combine solutions for parts by coordinate shift (0.2pts). For correctness, any explanation, for instance some text mentioning the word induction, is enough to get the maximal number of points. For the complexity, students get 0.2pts if the write down the correct recursion equation for $T(n)$, and 0.1pts for the induction proof.

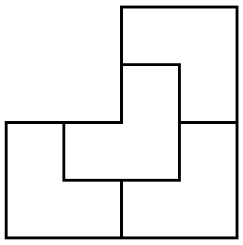
It is interesting to note that divide-and-conquer does not give us the fastest possible algorithm: using a different approach we may obtain a $\mathcal{O}(2^{2n})$ algorithm, that is, an algorithm that is linear in the size of the board. Call the original tile a tile of order 1. The idea is that using 4 tiles of order i we may create a tile of order $i + 1$ (see figure), which has exactly the same shape but with length twice as big. We can obtain a tiling of the board of size n by first adding a tile of order 1, then a tile of order 2, etcetera, and finally a tile of order n (see figure).

3 Graphs (1.5 points)

Let $G = (V, E)$ be an undirected graph. For $u, v \in V$, let $d(u, v)$ be the length of the shortest path from u to v if it exists, and ∞ otherwise. Let $U \subseteq V$ be a nonempty set of vertices. Then $d(U, v)$ denotes how far set U is from vertex v:

$$d(U, v) \quad = \quad \min_{u \in U} \ d(u, v).$$





The *eccentricity* of U, written $\epsilon(U)$, denotes how far set U is from the vertex in V that is most distant from it in the graph:

$$\epsilon(U) \quad = \quad \max_{v \in V} \; d(U, v).$$

Intuitively, if G is a country and U is the part of it that you know, $\epsilon(U)$ gives the largest distance from the places you know to a place you haven't seen yet. Describe and analyze an efficient algorithm to compute the eccentricity of U.

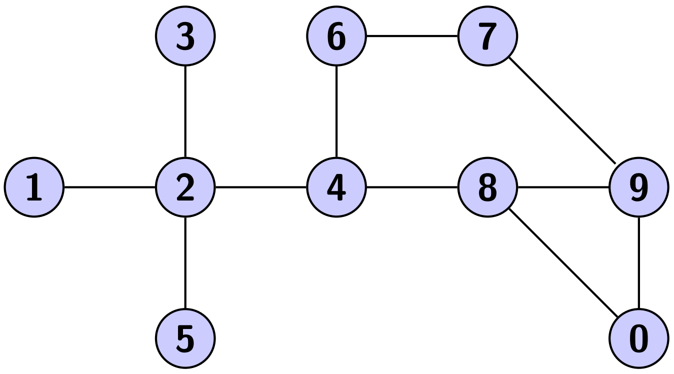


Figure 1: In this graph $d(1,0) = 4$, $\epsilon(\{1\}) = 4$ and $\epsilon(\{1,2,4\}) = 2$.

Solution: There are several ways to solve this problem. Here we describe an approach that conceptually is most simple. Let $G' = (V', E')$ be the graph obtained from G by adding a fresh vertex s and edges between s and all the vertices in U:

$$\begin{aligned} V' &= V \cup \{s\}, \\ E' &= E \cup \{(s,u) \mid u \in U\}. \end{aligned}$$

For $x, y \in V'$, let $d'(x,y)$ denote the length of the shortest path from x to y in G' if it exists, and ∞ otherwise. Then, for any vertex $v \in V$, the shortest path from s to v must go to a vertex in U. Moreover, the shortest path in G' from v to any node in U is also the shortest path in G from v to any vertex in U (if the shortest path in G' would pass through s then there would be a shorter path to a vertex in U). Hence:

$$d'(s,v) \;=\; 1 + \min_{u \in U} d'(u,v) \;=\; 1 + \min_{u \in U} d(u,v) \;=\; 1 + d(U,v),$$

and thus

$$\epsilon(U) \quad = \quad \max_{v \in V}\ d(U,v) = \max_{v \in V}\ d'(s,v) - 1.$$

Using Breadth-First-Search we may compute the shortest distance from s to any $v \in V$ (as discussed in the lecture on Dijkstra's algorithm) in time $\mathcal{O}(|V'| + |E'|) = \mathcal{O}(|V| + |E|)$. The eccentricity of U can then be computed in time $\mathcal{O}(|V|)$ by taking the maximum of $d'(s,v)$, for all $v \in V$, and subtracting 1.[1] Thus the overall time complexity of computing the eccentricty of U is $\mathcal{O}(|V| + |E|)$.

Correctness of this algorithm follows from the correctness of BFS and the above derivations.

Note that in an implementation we do not need to construct graph G': we may just adapt the code for BFS by setting $d[u] = 0$, for all $u \in U$, and placing all vertices from U in the FIFO queue. An alternative solution is to replace all the vertices in U by a single fresh vertex s, which has edges to all the vertices in $v \in V \setminus U$ that are connected to some vertex in U, and then run BFS from s in the modified graph and take the maximum of all distances.

Students may earn 0.9pts for the description of their algorithm, 0.3pts for the correctness argument, and 0.3pts for the complexity argument. Deduct 0.3pts if the algorithm is nonlinear.

4 Dynamic programming (2 points)

In the *longest increasing subsequence* problem, the input is a sequence of numbers $s = a_1, \ldots, a_n$. A *subsequence* is any sequence obtained from s by omitting zero or more elements, and an *increasing* subsequence is one in which the numbers are getting strictly larger. The task is to find an increasing subsequence of greatest length.

For instance, the longest increasing subsequence of 5, 2, 8, 6, 3, 6, 9, 7 is 2, 3, 6, 9.

1. Describe and analyze an $\mathcal{O}(n^2)$-time dynamic programming algorithm for the longest increasing subsequence problem.

2. Describe and analyze (possibly using a different approach than dynamic programming) an $\mathcal{O}(n \log n)$-time algorithm for the longest increasing subsequence problem.

Solution:

1. We want to be able to use solutions for a prefix of s to compute a solution for s "substructure property". Now a problem is that a longest increasing subsequence of a prefix $a_1 \cdots a_i$ does not necessarily contain a_i. In fact, we will not know what the final number of the longest increasing subsequence will be. But this makes it hard to use the solution for the prefix when constructing a solution for the full sequence. So therefore we slightly change the problem: we look for the longest increasing subsequence of $a_1 \cdots a_i$ that ends with a_i. By doing this we will be able to exploit substructure. Moreover, once we know, for each i, the longest increasing subsequence that ends in a_i, we can solve our original problem by simply taking the longest of these subsequences.

   Hence, for $1 \leq i \leq n$, let $L(i)$ denote the length of the longest increasing subsequence of $a_1 \cdots a_i$ that contains a_i as its final element. Let L denote the length of the longest increasing subsequence of s. Then we have the following recursion equations:

$$\begin{aligned} L(i) &= \max_{j<i, a_j<a_i} L(j) + 1 \\ L &= \max_j L(j) \end{aligned}$$

   (By definition, the maximum of the empty set is 0; so for instance $L(1) = 1$.) In order to see that the first equation holds, observe that since the list up to a_i must contain a_i

[1]We use the convention that $\infty = \infty - 1 = \infty + 1$.

its length is at least 1. Now suppose that there is a preceding element a_j. Then the length of the subsequence up to a_j must be $L(j)$: if the sequence up to a_j would not the longest increasing subsequence up to a_j, then the sequence up to a_i would not be the longest increasing subsequence either. This means we may compute $L(i)$ by adding 1 to the maximum of all $L(j)$ with $j < i$ and $a_j < a_i$. The longest increasing subsequence of s will contain at least one element. Let j be the final element then the value of the longest increasing subsequence of s will be $L(j)$. This means we may compute L by taking the maximum of all $L(j)$.

We may compute L by first computing $L(1)$, then $L(2)$, etc and then taking the maximum, see Algorithm 1. It is easy to modify the algorithm such that it returns the actual longest increasing subsequence: we introduce an array p of pointers that specifies for each index i the preceding element (if any) in the longest increasing subsequence that ends with a_i. All $p[i]$ values are set to NIL initially, and whenever we assign $L[i] \leftarrow \max(L[i], L[j])$, we also assign $p[i] \leftarrow j$. The optimal subsequence can then be reconstructed by following these backpointers. Due to the two nested for-loops, since in order to compute $L(i)$ we need to take the maximum over at most $i - 1$ preceding $L(j)$ values, the overall time complexity of our algorithm will be $\mathcal{O}(n^2)$.

Algorithm 1: Longest Increasing Subsequence

```
input  : a sequence of numbers $s = a_1, \ldots, a_n$
output: length of the longest increasing subsequence of $s$
foreach $i \in \{1, \ldots, n\}$ do
    $L[i] \leftarrow 0$
    foreach $j \in \{1, \ldots, i-1\}$ do
        if $a_j < a_i$ then
            $L[i] \leftarrow \max(L[i], L[j])$
    $L[i] \leftarrow L[i] + 1$
$L \leftarrow \max_{1 \leq i \leq n} L[i]$
return $L$
```

Students may earn 1pt for this item: 0.3pts for the recursion equations, 0.3pts for the description of the algorithm (either top down or bottom up), 0.2pts for the correctness analysis, and 0.2pts for the complexity analysis. Deduct 0.1pts if students only describe how to compute the length of the longest increasing subsequence, but not the subsequence itself. Students do not need to describe recursion equations explicitly; they may for instance also be embedded in pseudocode.

Several alternative solutions exist. Thus, for instance, rather then going from left to right in the array, we may also go from right to left. The textbook of Dasgupta, Papadimitriou and Vazirani describes a solution where we first compute a directed acyclic graph with vertices $\{1, \ldots, n\}$ and an edge (i, j) iff $a_i < a_j$. The goal then simply becomes to compute the longest path in this DAG. This is basically Problem 4 from the A&D exam of 2018!! Finally, we may use dynamic programming to compute $K(i, k)$, for $1 \leq i \leq n$ and k a number: the longest increasing subsequence starting at a_i in which every element is larger than k. Then we have the following recursion equations:

$$K(i,k) \;=\; \begin{cases} 0 & \text{if } i = n \text{ and } a_n \leq k \\ 1 & \text{if } i = n \text{ and } a_n > k \\ K(i+1,k) & \text{if } i < n \text{ and } a_i \leq k \\ \max(K(i+1,a_i)+1, K(i+1,k)) & \text{if } i < n \text{ and } a_i > k \end{cases}$$

The goal then becomes to compute $K(1, -\infty)$. In order to compute this value, we only need to consider the values k that actually occur in sequence s, and $-\infty$. Thus we need to fill a matrix of size $n \times (n+1)$ at most, and filling an entry in the matrix takes constant time. Thus the overall time complexity of a top down or bottom up implementation will be $\mathcal{O}(n^2)$.

2. This is definitely the hardest question from the exam. An $\mathcal{O}(n \log n)$ algorithm can be obtained by making a single pass over list s in combination with a binary search. For a good description (and animation) of the solution see `https://en.wikipedia.org/wiki/Longest_increasing_subsequence`.

5 Network flow (1.5 points)

Suppose you are organizing a conference where researchers present articles they have written. Researchers who want to present an article send a paper to the conference organizers. The conference organizers have access to a set A of reviewers who are each willing to read up to m_A articles. Let B the set of papers to review: each gets reviewed by up to m_B reviewers. Moreover, each submission has a particular topic and each reviewer has a specialization for a set of topics, so papers on a given topic only get reviewed by those reviewers who are experts on that topic. The conference organizers need to decide which reviewers will review each article (or equivalently, which articles will be reviewed by which reviewers). Describe and analyze an efficient algorithm that solves this problem.

Solution: We construct a flow network as follows. Besides a source s and a sink t, the network contains a node for each reviewer in A and a node for each paper in B. For each reviewer $\alpha \in A$, add an edge (s, α) with capacity m_A. Add a directed edge (α, β) to the network if some reviewer $\alpha \in A$ is expert on the topic of a paper $\beta \in B$. Set the capacity of that edge to be 1. Finally, for each $\beta \in B$, add an edge (β, t) with capacity m_B. Run Ford-Fulkerson to get the maximum flow for this network. Due to the integrality theorem, we know that the maximum flow is integer valued. If in the maximal flow there is a flow 1 along edge (α, β), we ask reviewer α to review article β. (Note that we may also reverse the network and connect the source s with the articles, and the sink t with the reviewers; the resulting maximal flow will be the same.)

Since the capacity of the edge (s, α) is m_A, reviewer α will read at most m_A articles. Moreover, since the capacity of the edge (β, t) is m_B, paper β will be read by up to m_B reviewers. It does not make sense that a single reviewer writes more than one review about the same paper (submitting the same review twice does not give any additional information). This constraint is not listed explicitly in the problem statement, but follows implicitly from the description of the problem domain, and is satisfied through our requirement that in the flow network each edge (α, β) has capacity 1. Finally, since we only add an edge (α, β) when reviewer α is an expert on the topic of article β, all papers get reviewed by experts. Therefore, all the constraints listed in the problem statement will be satisfied and our solution is correct. Moreover, the solution is optimal in the sense that there is no other assignment in which all the constraints are met and more papers are reviewed. If the maximal flow equals $\mid B \mid \times m_B$, then each paper gets reviewed by m_B reviewers and the organizers of the conference can be happy: they did a perfect job. If the max flow is less, then it may occur that some article does not get reviewed at all. This is of course problematic, but since the assignment didn't mention a lower bound on the number of reviews per paper we may ignore this issue here.

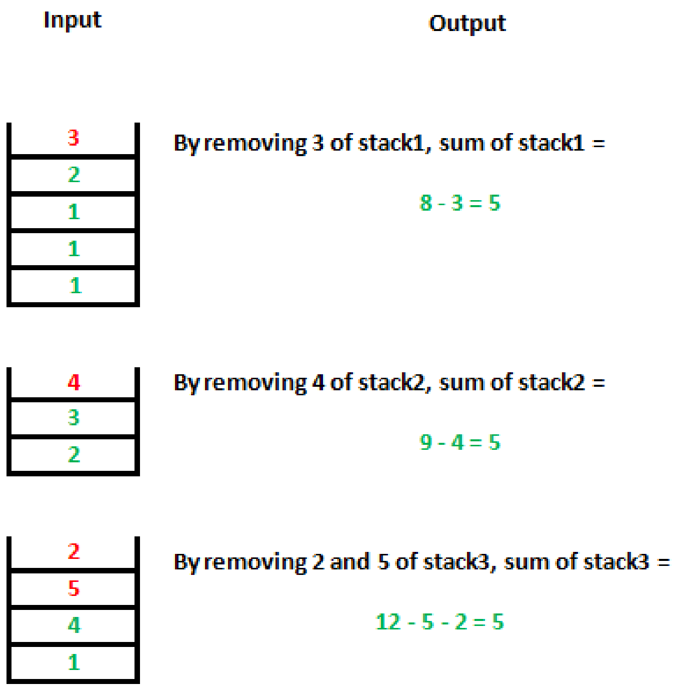
The running time of Ford-Fulkerson is $\mathcal{O}(mnC)$, where m is the number of edges, n is the number of vertices, and C is the maximal capacity. In our case, $m \leq |A| + |B| + |A| \times |B|$, $n = |A| + |B| + 2$, and $C = \max(m_A, m_B)$. Assuming that $|A|$, $|B|$, m_A and m_B are all positive, this leads to an overall time complexity of $\mathcal{O}(|A| \times |B| \times (|A| + |B|) \times \max(m_A, m_B)$. (Note that the complexity of Ford-Fulkerson dominates the complexity of constructing the flow network and computing the final assignment based on the maximum flow.) It is also ok (and in fact even better) when students argue that the complexity of Ford-Fulkerson is $\mathcal{O}(mf)$, where f is the value of the maximum flow. In our case, the maximum flow f can, for instance, be bounded by $|B| \times m_B$, leading to an improved overall time complexity $\mathcal{O}(|A| \times |B|^2 \times m_B)$. It is also ok if students use an alternative maximum flow algorithm, for instance Edmonds-Karp, and derive a complexity bound based on that.

Grading:

- 0.9pts for the correct algorithm: 0.6pts for the correct flow network, and 0.3pts for a description how paper assignment can be computed using a max flow algorithm. If a flow network does not satisfy any of the four requirements discussed above, the student gets 0pts for this part. (He/she still may earn points for the description of how the paper assignment is computed from the max flow.)
- 0.3pts for correctness. Students should explain why all three requirements that are explicitly mentioned in the assignment are met, and discuss why the organizers want a max flow.
- 0.3pts for the time complexity. This needs to be described in terms of the input parameters of the problem, just writing $\mathcal{O}(mnC)$ (or a similar expression) without explaining what m, n and C are will not yield any points.

6 Greedy (1.5 points)

You are given three stacks that contain positive numbers. You are allowed to remove, repeatedly, the top element of one of the stacks. Your task is to do this in such a way that the sum of the remaining numbers for each of the stacks is equal and maximal. Note that if we remove all the numbers from all the stacks, the sum of the remaining numbers (there aren't any) in all of the stacks is equal to 0; often it will be possible to obtain an equal sum that is larger by removing fewer elements. The diagram below shows an example. Describe and analyze a greedy algorithm to compute the maximum equal sum.


Input
Output
3
2
1
1
1
By removing 3 of stack1, sum of stack1 =
8 - 3 = 5
4
3
2
By removing 4 of stack2, sum of stack2 =
9 - 4 = 5
2
5
4
1
By removing 2 and 5 of stack3, sum of stack3 =
12 - 5 - 2 = 5

Solution: For each stack i, introduce a variable sum_i, compute the sum of all the numbers contained in stack i, and assign this value to sum_i.

Now our greedy algorithm proceeds as follows. While the values of sum_1, sum_2 and sum_3 are not all equal, we pop the top element from a stack i whose value is maximal and decrease the value of sum_i with the value of this top element.

Clearly, when our algorithm terminates, the values of sum_1, sum_2 and sum_3 will be equal. Since eventually all three stacks will be empty, in which case $sum_i = 0$ for all i, the algorithm is guaranteed to terminate. Let S be the unique value such that, after popping zero or more elements from each of the three stacks, the sum of the remaining numbers for each of the stacks is equal and maximal. We claim that at any point during execution of the while loop, $sum_i \geq S$, for all i. In order to see that this invariant holds, observe that the invariant trivially holds at the start of the loop when sum_i is equal to the sum of all numbers contained in stack i. Now whenever we pop an element from say stack i, there is another stack, say stack j, such that $sum_i > sum_j$. By the invariant, we know $sum_j \geq S$. Hence, before popping $sum_i > S$. But since S can be obtained as the sum of the remaining elements of stack i after popping zero or more elements, and since all numbers on the stack are positive, we conclude that we can safely pop from stack i, and that after decreasing the value of sum_i with the value of the top element, the invariant $sum_i \geq S$ will still hold. The invariant immediately implies that upon termination of the while loop, $sum_i = S$, for all i.

For $i = 1, 2, 3$ let n_i denote the number of elements originally contained in stack i. Computing the sum of the numbers contained in each of the stacks can be done in time $\mathcal{O}(n_1 + n_2 + n_3)$. Since in each iteration of the while loop we pop an element from a stack (except in the final iteration), the total number of iterations is less than $n_1 + n_2 + n_3$. Each iteration takes a constant amount of time, and thus the while loop will take $\mathcal{O}(n_1 + n_2 + n_3)$ time. We conclude that the overall time complexity of the algorithm is $\mathcal{O}(n_1 + n_2 + n_3)$.

As usual, students may earn 0.9pts for the algorithm, 0.3pts for the correctness analysis, and 0.3pts for the complexity analysis. Given that this problem is really easy, students only get points for correctness if the analysis is really convincing.