Algorithms and Data Structures (IBC027)

January 13, 2021

Whenever an algorithm is required (assignments 2-6), it can be described in pseudocode or plain English. **The explanations of your algorithms may be informal but should always be clear and convincing**. *The grade equals the sum of the scores for all 6 problems below divided by 10 (plus possibly a bonus score for the homework assignments). Good luck!*

1 Quiz (27 points)

The quiz consists of 9 questions, and for each question that is answered correctly you may earn 3 points. If students give a direct answer (e.g. "A" for Question 2) then we ignore any explanation/justification of this answer (even in case the answer is wrong but the justification is correct). If students only give a justification but no direct answer, they may get at most 2 pts for a question.

Question 1 Is $2^{n+1} \in \mathcal{O}(2^n)$? Is $2^{2n} \in \mathcal{O}(2^n)$?

Solution Yes and No.
For the first question, let $n_0 = 0$ and $c = 2$. Then for all $n \geq n_0$,

$$2^{n+1} = 2 \cdot 2^n \leq c \cdot 2^n$$

For the second question, suppose that there exists natural number n_0 and positive real number c such that, for all $n \geq n_0$,

$$2^{2n} \leq c \cdot 2^n$$

Since $2^{2n} = 2^n \cdot 2^n$ and $2^n > 0$, this is equivalent to $2^n \leq c$, which in turn is equivalent to $n \leq \log_2 c$. But this inequality clearly does not hold for all n. Thus we have derived a contradiction.

Marking Students may earn 1pt for a correct answer for the first item, and 2pts for a correct answer for the second item (which is harder).

Question 2 In a binary max heap containing n numbers, the smallest element can be found in time

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

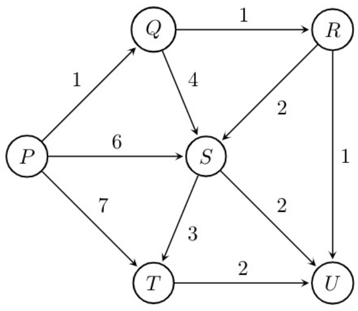
(B) $\mathcal{O}(\log n)$

(C) $\mathcal{O}(\log \log n)$

(D) $\mathcal{O}(1)$

Solution (A).
In a max heap, the smallest element is always present at a leaf node. So we need to check for all leaf nodes for the minimum value. Worst case complexity will be $\mathcal{O}(n)$.

Question 3 Suppose we run Dijkstra's shortest-path algorithm on the following edge-weighted directed graph with vertex P as the source.



In what order do the nodes get included into the set of vertices for which the shortest path distances are finalized?

(A) P, Q, R, S, T, U

(B) P, Q, R, U, S, T

(C) P, Q, R, U, T, S

(D) P, Q, T, R, U, S

Solution The correct answer is (B).
Dijkstra's algorithm maintains a set S of nodes, initially equal to $\{P\}$, and in each step adds the node which has the smallest distance to any node in S. So first Q is added, which has distance 1 to P, then R is added, which has distance 1 to Q, then U is added, which has distance 1 to R, then S is added, which has distance 2 to R, and finally T is added.

Question 4 Discuss three differences between Dijkstra's algorithm and the Floyd-Warshall algorithm.

Solution We mention four differences:

1. Dijkstra's algorithm computes shortest paths from a give source to other nodes in the graph, whereas Floyd-Warshall computes the shortest path between any pair of nodes.
2. Unlike Dijkstra's algorithm, the Floyd-Warshall algorithm also works for graphs with negative edge weights.
3. Dijkstra's algorithm has lower time complexity. For a graph with n vertices and m edges, represented with an adjacency matrix representation, the time complexity of Dijkstra's algorithm is $\Theta(m \log n)$ (if a priority queue is used) whereas Floyd-Warshall requires $\Theta(n^3)$.
4. Dijkstra's algorithm is an example of a greedy algorithm, whereas Floyd-Warshall is an example of dynamic programming.

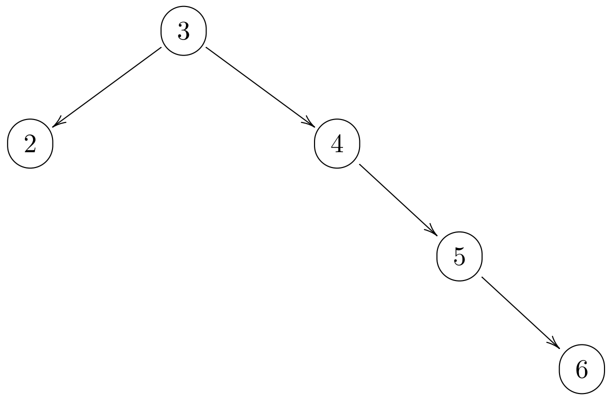
Marking Students may earn 1pt for each sensible difference that the mention.

Question 5 Consider the following sequence of operations on a binary search tree which is initially empty:

insert(3); insert (1); insert (2); insert(4) insert(5); insert(6); delete(1)

What is the height of the resulting tree?

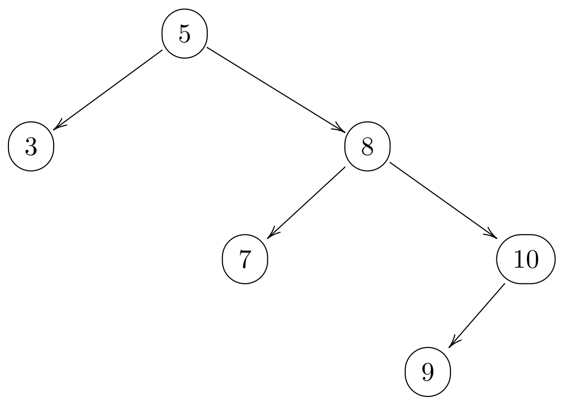
Solution 3. The resulting tree looks as follows. Recall that the *height* of a tree is the number of edges on the longest downward path from the root to a leaf.



Marking

- 3pt for "3"
- 2pt for "4" (many students get the definition of height wrong)
- 0pt otherwise

Question 6 Consider the following binary tree.



Which of the following statements are true? (Multiple answers possible!)

(A) This is a binary search tree;

(B) This can be colored as a red-black tree;

(C) This is an AVL tree;

(D) This is a complete binary tree;

Solution Only (A) and (B) are true.

Marking $\max\{0, c-1\}$ where c is the number of statements correctly classified as true or false.

Question 7 Suppose we have a red-black tree, and insert an element into it using the standard procedure for binary search trees. We color the new node red. Which red-black tree properties are potentially violated after this operation?

Solution

- that the root should be coloured black
- that every red node has only black children

Marking

- 2pt for "red node black children" property
- 1pt for "root coloured black".
- Deduct 1pt for each "wrong" RB-property listed (i.e., which is not violated by insertion.

Question 8 The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function $h(k) = k \mod 10$ and linear probing. What is the resulting hash table?

(A)

0	
1	
2	2
3	23
4	
5	15
6	
7	
8	18
9	

(B)

0	
1	
2	12
3	13
4	
5	5
6	
7	
8	18
9	

(C)

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

(D)

0	
1	
2	12, 2
3	13, 3, 23
4	
5	5, 15
6	
7	
8	18
9	

Solution The correct answer is (C).
Answer A is wrong because in hash table you don't overwrite previous values. Answer B is wrong because in a hash table you don't ignore values if an entry in the table is already filled. Answer C is right because in a hash table with linear probing, you search for the next free entry in case the entry specified by the hash function is filled, which is exactly what has been done here. Answer D is wrong because in a hash table with linear probing, each entry in the table contains at most one entry.

Question 9 Discuss three advantages/disadvantages of hashing using chaining when compared with hashing using open addressing.

Solution We mention five differences:

1. Open addressing is usually faster than chained hashing when the load factor is low because you don't have to follow pointers between list nodes.
2. Open addressing gets very, very slow if the load factor approaches 1, because you end up usually having to search through many of the slots in the bucket array before you find either the key that you were looking for or an empty slot.
3. With open addressing, you can never have more elements in the hash table than there are entries in the bucket array. With chained hashing the number of elements in the table is a priori unlimited.
4. With open addressing, deletion is problematic. For instance, if you use tombstones then performance may degrade. With chained hashing using a doubly linked list, deletion is constant time.
5. It is simpler to make a chaining-based hash table concurrent, since you can lock each chain separately.

Marking Students may earn 1pt for each sensible difference that the mention.

2 Binary search trees (10 points)

Describe an efficient algorithm for constructing a binary search tree of height $\mathcal{O}(\lg n)$ from a sorted array of n integers. Include a complexity argument.

Solution If the input array is empty, return; otherwise, take the element at the middle position of the array (making a choice between the middle two elements if the array if of even length) and construct a BST with that element as its root; recursively process the left half of the array, and make that the left sub-tree (no child if this is empty); similarly for the right half of the array. In this way, every element of the input array is visited once, and we obtain a linear time algorithm.

Alternative which is $\mathcal{O}(n \lg n)$: if the input array is empty, return; otherwise, insert the element at the middle position of the array first (making a choice between the middle two elements if the array if of even length), then recursively process the left and the right half.

The number of recursive calls is in $\mathcal{O}(2n)$: we make 2 calls for every element of A. We insert n elements into a balanced tree, each insertion is in $\mathcal{O}(\lg n)$, so the overall complexity is $\mathcal{O}(n \lg n)$.

Another $\mathcal{O}(n \lg n)$ alternative: insert everything from left to right into a red-black tree. This is of the given complexity, since insertion in a red-black tree is in $\mathcal{O}(\lg n)$, so that inserting n elements yields $\mathcal{O}(n \lg n)$. We know from the lecture that red-black trees are in $\mathcal{O}(\lg n)$ (it is important to make this remark, if this approach is taken).

Marking

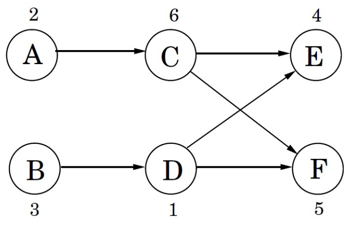
- 7pt for a clear and correct $\mathcal{O}(n)$ algorithm
  - deduct 1pt for forgetting empty array case (which is quite important in most implementations)
  - deduct 1-2pt for small mistakes or too much imprecision (for instance, describing the algorithm more or less correctly but not stating e.g. what becomes left/right subchild, or what is called recursively)
  - no deduction for small rounding errors in selecting indexes (this makes grading too hard, depends on conventions on starting index of arrays, etc)
- 3pt for complexity analysis
  - 2pt correct complexity (1pt if $\mathcal{O}(n \lg n)$ is given for a $\mathcal{O}(n)$ algorithm)
  - 1pt explanation
- If an $\mathcal{O}(n \lg n)$ algorithm is given: max 5pt (out of 7) for the algorithm. Complexity analysis can still be 3pt, if it matches the given algorithm, so maximal score is 8pt in this case.

3 Graph algorithms (16 points)

You are given a directed graph $G = (V, E)$ (in adjacency list format) in which each node $u \in V$ has an associated *price* p_u which is a positive integer. Define an array cost as follows, for each $u \in V$,

$\mathsf{cost}[u] =$ price of the cheapest node reachable from u (including u itself).

For instance, in the graph below (with prices shown for each node), the cost values of the nodes A, B, C, D, E, F are 2, 1, 4, 1, 4, 5, respectively.



Your goal is to design an algorithm that fills the *entire* cost array (i.e., for all vertices).

1. Give a linear-time algorithm that works for directed *acyclic* graphs.
2. Extend this to a linear-time algorithm that works for all directed graphs.

Include a clear explanation and discuss correctness and runtime complexity for both algorithms.

Solution

1. Run a topological sort on G and reverse the resulting list of vertices (so vertices with smallest finishing time come first). As explained in the lecture, this can be done in a time that is linear in the size of G. Now we go through the list of vertices, and for each vertex u, we set

$$\mathsf{cost}[u] \quad = \quad \begin{cases} p_u & \text{if } u \text{ has no outgoing edges} \\ \min(p_u, \min_{(u,v)\in E}(\mathsf{cost}[v]) & \text{otherwise} \end{cases} \tag{1}$$

Note that, by definition of a topological ordering, $(u, v) \in E$ implies that v occurs before u in the reversed list, and thus $\mathsf{cost}[u]$ is always defined in terms of entries of array cost that have already been filled before. Since each vertex is visited once, the time complexity of filling the array is linear in the size of G. Thus our algorithm, which fills the entire cost array, is linear in the size of G. The algorithm is correct since we may prove (by induction on the length of the longest path from u) that the price of the cheapest node reachable from u is equal to $\mathsf{cost}[u]$.

Instead of first doing a topological sort (which basically is a modification of DFS) and then compute the cost array, once may also define a direct adaptation of DFS as shown in Algorithm 1. This solution is more complicated, since one is sort of reinventing the wheel. For acyclic graphs correctness follows from the following property, that one may establish by induction on the length of the longest path from u: whenever u is colored black then $\mathsf{cost}[u]$ gives the price of the cheapest node reachable from u. In the proof of the induction step, we use that whenever DFSVisit explores an edge (u, v), vertex v will be colored either white or black (since G is acyclic it does not contain back edges, see Lemma 22.11 from the slides). Moreover, we know that after DFSVisit(G, v), vertex v will be black. Since it is a minor adaptation of DFS, Algorithm 1 is clearly linear.

Algorithm 1: Computing the cost array for acyclic graphs using an adaptation of DFS

```
Procedure DFS($G$)
    foreach vertex $u$ of $G$ do
        $color[u] \leftarrow white$ ;
    foreach vertex $u$ of $G$ do
        if $color[u] = white$ then
            DFSVisit($G$, $u$) ;
Procedure DFSVisit($G$, $u$)
    $color[u] \leftarrow gray$ ;
    $cost[u] \leftarrow p_u$ ;
    foreach $v \in Adj[u]$ do
        if $color[v] = white$ then
            DFSVisit($G$, $v$) ;
        $cost[u] \leftarrow \min(cost[u], cost[v])$ ;
    $color[u] \leftarrow black$ ;
```

2. Note that the value of array cost is the same for all nodes that are part of the same strongly connected component. Therefore, in order to compute cost for an arbitrary directed graph G, we first compute the strongly connected components of G using the linear time algorithm that has been presented during the lecture. Given the SCCs, we can then compute the component graph in linear-time. If node u in the component graph corresponds to a SCC C from G, then the price associated to u is the minimum of the prices of nodes from C. By a lemma presented during the lecture, the component graph is a DAG. This means that we may compute, again in linear time, the cost array for the component graph. Finally, we fill in linear time the cost array for the original graph G by associating to each node u of G the cost of the corresponding node (the node for the SCC of u) in the component graph.

   Note that Algorithm 1 does not always produce the correct results for graphs that contain cycles. A counterexample is shown in Figure 2. Suppose vertices v_1, v_2, v_3 and v_4 have

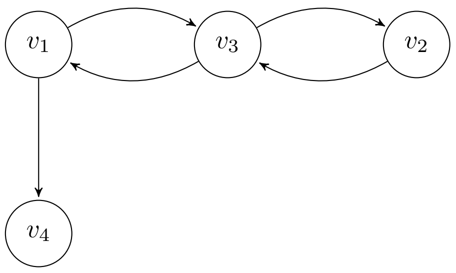


Figure 1: Counterexample showing that DFS adaptation does not work for cyclic graphs.

   prices 9, 7, 8 and 4, respectively. Suppose further that the DFS algorithm visits vertices in the order v_1, v_2, v_3 and v_4, and edges in lexicographic order. Then procedure DFS first makes a call to DFSVisit(G, v_1). After this call all nodes are black, cost$[v_1]$ = cost$[v_4] = 4$ and cost$[v_2]$ = cost$[v_3] = 7$. Subsequent calls to DFSVisit(G, v_2), DFSVisit(G, v_3) and DFSVisit(G, v_4) will not change the value of cost$[v_2]$.

Marking

- 9pts for the first item:

  - 5pts for the description of a correct, linear algorithm (2pts for equation (1) and 3pts for correct graph traversal)
  - no points for a nonlinear solution (such as running DFS from every vertex)!
  - deduct 1pt for small mistakes or too much imprecision (for instance, describing the algorithm more or less correctly but not reversing the topological sorting)
  - 2pts for the correctness argument (in case topological sorting is used a reference to CLRS or the slides suffices, in case an adaptation of DFS is used some extra argument is required)
  - 2pts for the the complexity argument (you can get 2pts even when algorithm is incorrect or nonlinear)

- 7pts for the second item:
  - 5pts for the description of a correct, linear algorithm
  - 1pt for the correctness argument
  - 1pt for the complexity argument

4 Dynamic programming (16 points)

Radboud University plans the construction of a new building. Inspired by the reputation of the Erasmus building as an architectural highlight, they want to build another tower, which has to be as big as possible. But of course space on campus is limited. It is your task to help them find a suitable location, which needs to be a square. Based on a map of the campus, you have to decide the largest square area which is available for this grand new tower.

The input is an $m \times n$ matrix A consisting of zeros and ones, where $A_{i,j} = 1$ means that the location at row i and column j is available, and $A_{i,j} = 0$ means it is not. The problem is to determine the size of the largest square of ones present in this matrix (by size we mean the length of the sides). For instance, if the input A is

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

then the largest such square has size 3. Describe an algorithm for this problem using dynamic programming. Include a clear explanation and a complexity analysis. Hint: compute a matrix S such that $S_{i,j}$ is the size of the largest square of ones in A with (row i, column j) as its bottom right corner.

Solution. We recursively compute an $m \times n$ matrix S such that $S_{i,j}$ is the size of the largest square of ones with i, j as its bottom right corner. The key idea is that there is a square of ones of size k in A with i, j as its bottom right corner if only if $A_{i,j} = 1$ and there are squares of ones of size $k-1$ at coordinates $A_{i-1,j}$, $A_{i-1,j-1}$ and $A_{i,j-1}$. Thus, to compute $S_{i,j}$, the largest such square, we check that $A_{i,j} = 1$ (otherwise no square at all is possible), and then add one to the minimum of $S_{i-1,j}$, $S_{i-1,j-1}$, $S_{i,j-1}$. The base case is when $i = 1$ or $j = 1$; in that case the maximum size is 1, which is achieved iff $A_{i,j} = 1$. Altogether this leads to:

$$S_{i,j} = \begin{cases} A_{i,j} & \text{if } i = 1 \text{ or } j = 1 \\ \min\{1 + S_{i-1,j}, S_{i-1,j-1}, S_{i,j-1}\} & \text{if } i > 1,\ j > 1 \text{ and } A_{i,j} = 1 \\ 0 & \text{otherwise} \end{cases}$$

This immediately yields an algorithm: either implement this recursive computation top-down using memoization, or compute it bottom-up, for instance column by column from left to right (and each column from top to bottom).

Having computed S, the desired size of the maximum square of ones is simply the maximal value in S, which we can find with a single pass through the array.

The complexity of computing each entry in $S_{i,j}$ is $\mathcal{O}(1)$, so in total $\mathcal{O}(nm)$ for computing S. The final scan of S for the maximal value is also in $\mathcal{O}(nm)$, so the complexity of our algorithm is $\mathcal{O}(nm)$.

Correctness follows from correctness of the recursive equation, for which we have argued above.

Marking

- 7pt for recursive equation (doesn't need to be formally a recursive equation, can be written text or given as a bottom-up implementation directly; as long as it's precise), including 2pt for getting the base cases right.
- 3pt for explanation/correctness of the equation
- 2pt for explaining how this yields an algorithm (either bottom-up, in that case an order should be mentioned; or top-down w/memoization)
- 4pt complexity
  - 2pt for correct expression
  - 2pt for explanation—deduct 1 pt if the final scan for the maximal element is ignored. Also it should be mentioned that computing an entry is in $\mathcal{O}(1)$ for 1pt.

Some special cases:

- 0pt in case an (inefficient) algorithm is given that doesn't use DP.
- No need to explicitly mention they use DP if it's clear they do.
- I had one solution which interpreted size as surface and deducted 1pt.

5 Divide and conquer (16 points)

Let $A = [a_1, a_2, \dots, a_n]$ be an integer array of length n. An *inversion* is a pair of indices i, j with $i < j$ such that $a_i > a_j$. For instance, in the array

$$[5, 4, 6, 2]$$

there are 4 inversions: $(1, 2), (1, 4), (2, 4)$ and $(3, 4)$.

Describe an efficient algorithm based on divide and conquer, which takes an integer array A as input, and returns the number of inversions in A. Explain your algorithm, and its correctness and runtime complexity.

Solution. A key idea is that the number of inversions in an array A is the sum of:

1. the number of inversions in the left half of A;
2. the number of inversions in the right half of A;
3. the number of inversions between an element in the left half and an element in the right half.

This suggests a recursive implementation, and in fact to implement it efficiently a good strategy is to integrate the counting of inversions in Mergesort, as counting the number of inversions between elements of the left and the right half is done efficiently if both halves are already sorted: a crucial point is that sorting both halves does not change the number of inversions between the two halves.

Indeed, we can integrate the computation of the number of inversions between an element in the left and right half with the merging procedure: while merging two arrays A_1 and A_2, with pointers i and j, we keep track of the number of inversions as a variable s which is initialized to 0, and every time $A_1[i] > A_2[j]$ we add $\text{length}(A_1) - i + 1$ to s (as all the remaining elements of A_1 are inversions with $A_2[j]$). The merge procedure then returns this sum s.

Finally, our variant of Mergesort splits and sorts, but apart from sorting it returns the number of inversions, by recursively computing the number of inversions in the two halves (if one of the two arrays is empty the number of inversions is 0), and the inversions between the two halves while merging as explained above, and returning the sum of these three values. Here's an implementation:

```python
def countInv(list):
    if(len(list) <= 1):
        return list, 0
    else:
        mid = int(len(list)/2)
        left, a = countInv(list[:mid])
        right, b = countInv(list[mid:])
        result, c = mergeAndCount(left, right)
        return result, (a + b + c)

def mergeAndCount(left, right):
    result = []
    count = 0
    i,j = 0,0

    while(i < len(left) and j < len(right)):
        if(left[i] <= right[j]):
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            count += len(left)-i
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result, count
```

Complexity: This does not affect the complexity of Mergesort, as it adds only one extra operation to every comparison in the merging procedure, and one extra operation in the main body of Mergesort. Therefore, the complexity is still $\mathcal{O}(n \lg n)$.

Correctness: this hinges on the following main points:

- the fact that the recursive characterisation of inversions that we established in the beginning is correct, and that sorting the two halves does not change the number of inversions between the two halves, since that is exactly what is computed within our adaptation of mergesort; and
- the correctness of the way we compute the number of inversions between the halves A_1 and A_2 during merging.

The former is obvious; we focus on the second point, i.e., the correctness of merging. To this end, observe that since A_1 is sorted, given an index j in the bounds of A_2, the inversions (i, j) between indices i of A_1 and our fixed index j are exactly those elements on the right of (and including) the first i for which $A_1[i] > A_2[j]$. This is exactly what is counted during our adapted merging procedure.

Marking

- 3pt complexity
- 3pt correctness (give 2pt for a reasonable story, 3pt for a really convincing argument)
- 10pt algorithm, explanation, including:
  - 3pt for using mergesort/simultaneous sorting idea (mergesort doesn't need to be mentioned explicitly)
  - 3pt for the right divide & conquer idea (splitting; doesn't need mention D&C explicitly but it should be clear it's used)
  - 4pt remaining points for clear and well-described algorithm, merge procedure etc.

Special cases:

- 0pt for an algorithm that doesn't use D&C at all (as above, it doesn't need to be mentioned explicitly; but it needs to be used)
- max 12 pt for an algorithm that uses the D&C idea but doesn't sort, making it n^2 anyway (take the points from the 3 for mergesort and one from the 4pts for the algorithm); note that in this case correctness and complexity may still be correct, as long as they match whatever the solution is doing.

Algorithm 2: Computation of a minimum spanning tree.

```
Function MST($G$)
    sort the edges of $G$ according to their weights ;
    for each edge $e$ from $G$, in decreasing order of $c_e$ do
        if $e$ is part of a cycle of $G$ then
            $G \leftarrow G - e$ (that is, remove $e$ from $G$) ;
    return $G$ ;
```

6 Greedy algorithms (15 points)

Consider Algorithm 2 below, which takes as input an undirected, connected graph $G = (V, E)$ (in adjacency list format) with edge costs c_e:

1. Explain/prove why this algorithm returns a minimum spanning tree.
2. On each iteration, the algorithm must check whether there is a cycle containing a specific edge e. Give a linear-time algorithm for this task and justify its correctness.
3. What is the overall time complexity of this algorithm, in terms of $|E|$?

Solution.

1. The algorithm can be viewed as yet another instance of the generic greedy algorithm discussed during the lecture. This time, the red rule is being applied for an edge e when it is being removed from the graph by Algorithm 2 from the graph, and the blue rule otherwise. Whenever Algorithm 2 removes an edge e from a cycle then we know that e is the maximal weight edge on that cycle: edges of G have been sorted on their weight, and all edges with larger weight have either been remove from G in a previous iteration, or are not part of a cycle. Removing an edge in Algorithm 2 corresponds to coloring the edge red in the greedy algorithm. The red rule from the greedy algorithm can only be applied on cycles with no red edges, but those are exactly the cycles that remain when we have removed the "red" edges in Algorithm 2. Whenever Algorithm 2 does not remove an edge e then this edge is not part of a cycle in the reduced graph. Therefore, e induces a cut, with e the only element of the cutset in the reduced graph. In the original graph G, the cutset may contain other edges, but these have all been colored red and have greater or equal weight. Therefore, e is the edge in the cutset with minimal weight and it can be colored blue.

   Instead of proving correctness of Algorithm 2 by reduction to the greedy algorithm from the lecture, we may also give a direct proof. For this the key idea is to show that whenever the algorithm removes an edge e there exists a minimum spanning tree that does not contain e.

   For our discussion, let graph G_0 denote the original input of function MST. Initially $G = G_0$, and since G_0 is a connected weighted graph, there is a subgraph T of G that is a minimal spanning tree of G_0. We show that after each iteration of the for-loop this invariant is preserved. If the edge e that is considered in an iteration is not part of a cycle of G then G remains unchanged, so clearly T will still be a subgraph of G that is a minimal spanning tree of G_0. A second case is where edge e is part of a cycle but not part of T. Also in this case, after removing e from G, T will still be a subgraph of G that is a minimal spanning tree of G_0. The third case is where edge e is both part of a cycle and part of T. Now if we remove edge e from T, we obtain a cut S and $V - S$ of the nodes of G. As discussed during the lecture, a cycle and a cutset intersect in an even number of edges. So there is another edge f that is on the cycle and part of the cutset connecting S and $V - S$. Now the weight of f cannot be larger than the weight of e since otherwise (being part of a cycle) f would have been removed earlier. But this means that the graph T' obtained by replacing e by f is a minimal spanning tree of G_0 that is a subgraph of the graph obtained by removing e from G.

Since we remove all edges that are on a cycle, the resulting graph G will contain no cycles. But since by the invariant it contains a minimal spanning tree T, and adding any edge to a spanning tree will create a cycle, G and T must actually be equal.

2. Suppose $e = \{v, w\}$. Remove e from G and use BFS to check whether there is a path from v to w in the reduced graph. This can be done in linear time. If there is a path then the original graph contains a cycle that contains e, and if there is no such path then there is no cycle containing e.

3. The overall time complexity of Algorithm 2 is $\mathcal{O}(|E|^2)$, as we need $|E|$ iterations of the for-loop, with each iteration taking $\mathcal{O}(|E|)$ time: as shown in item 2 we can check in linear time whether e is on a cycle (note that since G is connected, $\mathcal{O}(|V| + |E|) = \mathcal{O}(|E|)$), and removing an edge from a graph is linear. The quadratic bound on the for-loop dominates the $\mathcal{O}(|E| \log |E|)$ time complexity of the initial sorting step.

Marking

- 9pts for the first item
  - 4pts for an argument that the resulting graph is a acyclic
  - 2pts for an argument that the resulting graph is connected (so together with previous statement this establishes that algorithm returns a spanning tree)
  - 3pts extra for an argument that is is a *minimal* spanning tree
- 3pts for the second item
- 3pts for the third item; deduct 1pt if no BigO notation is used, 1pt if expression is not simplified (no points are deducted when $\mathcal{O}(V + E)$ is not simplified to $\mathcal{O}(E)$, even though this holds for connected graphs), and 1pt if complexity is not phrased in terms of input parameters.