University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn

Algorithm Theory

22nd of March 2023, 09:00 - 11:00

Name: .....

Matriculation No.: .....

Signature: .....

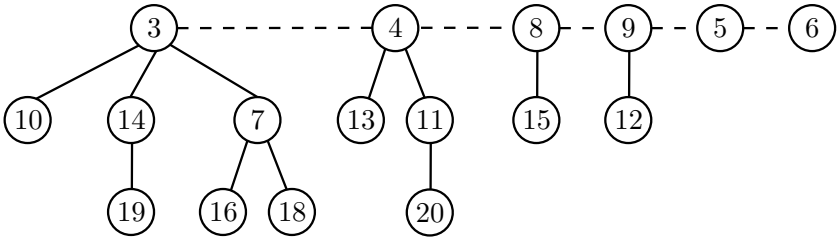
Do not open or turn until told so by the supervisor!

- Write your **name** and **matriculation number** on this page and **sign** the document.
- Your **signature** confirms that you have answered all exam questions yourself without any help, and that you have notified exam supervision of any interference.
- You are allowed to use a summary of **six handwritten A4 pages**.
- **No electronic devices** are allowed.
- Write legibly and only use a pen (ink or ball point). Do **not use red**! Do **not use a pencil**!
- You may write your answers in **English or German** language.
- Only **one solution per task** is considered! Make sure to strike out alternative solutions, otherwise the one yielding the minimal number of points is considered.
- **Detailed steps** might help you to get more points in case your final result is incorrect.
- The keywords **Show...**, **Prove...**, **Explain...** or **Argue...** indicate that you need to prove or explain your answer carefully and in sufficient detail.
- The keywords **Give...**, **State...** or **Describe...** indicate that you need to provide an answer solving the task at hand but without proof or deep explanation (except when stated otherwise).
- You may use information given in a **Hint** without further explanation.
- **Read each task thoroughly** and make sure you understand what is expected from you.
- **Raise your hand** if you have a question regarding the formulation of a task or if you need additional sheets of paper.
- A total of **45 points** is sufficient to pass and a total of **90 points** is sufficient for the best grade.
- Write your name on **all sheets**!

Task	1	2	3	4	5	6	**Total**
Maximum	42	24	16	16	12	10	120
Points							

Task 1: Short Questions *(42 Points)*

(a) Execute a `delete-min` operation (including the `consolidate` part) on the following Fibonacci heap and give its final state. Intermediate steps might be helpful in case the final result is wrong. The degree of each node is equal to its rank. *(8 Points)*



(b) You are rich and have an unlimited supply of credit cards, but each has a limit of ℓ. You need to purchase k items which cost $c_1, \ldots, c_k$, respectively such that $c_i \leq \ell$, for all $i \in \{1, 2, ..., k\}$. Each item has to be paid from a credit card, but you must not exceed its limit ℓ. You want to minimize the number of credit cards that you need to use to make all k purchases. You decide on the following 'First Fit' strategy:

- For i from 1 to k do the following.
- If there is a credit that was already used but still has sufficient credit left to pay c_i, purchase the item with this credit card.
- Else, use a yet unused credit card and increase the number of used credit cards by one.

Show that this strategy gives you a 2-approximation of the minimum number of used credit cards to purchase the items. *(10 Points)*

Hint: Show that at any point for any two distinct cards A and B that have been used to pay at least one item, the sum over the costs of items payed with A and B is at least ℓ. If you are not able to show this, you can assume it in order to solve the problem and get partial points.

(c) Let A denote the $n \times n$ matrix filled with distinct positive integer values. An element is a *peak element* if it is greater than or equal to its four neighbors, left, right, top and bottom e.g. neighbors for $A[i][j]$ are $A[i][j-1], A[i][j+1], A[i-1][j]$, and $A[i+1][j]$. For corner elements, missing neighbors are considered of negative values. The task is to find the index of a peak element. Devise an algorithm that finds the index of a peak element in $O(n \log n)$, argue correctness and running time. *(14 Points)*
Hint: First show that if we pick an arbitrary column j and if we then pick a row i such that $A[i][j] \geq A[i'][j]$ for all $i' \neq i$, then $A[i][j]$ is a peak element or there exists a peak element in a column $j' \neq j$. If you cannot prove this fact, you can still use it to devise your algorithm.

(d) Let $G = (V, E)$ be a graph and $w : E \to \mathbb{R}_+$ be a function which assigns a weight to each of the edges of G. Then the weight $w(F)$ of a subset $F \subseteq E$ of the edges of G is defined as $w(F) := \sum_{e \in F} w(e)$. Now the *weighted matching problem* is to find a matching M in G that has maximum weight.

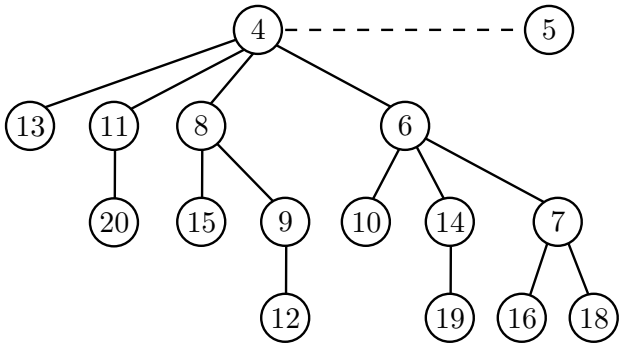
Algorithm 1 `Greedy Weighted Matching` ▷ Given $G = (V, E), w : E \to \mathbb{R}_+$

```
1: M := ∅
2: while E ≠ ∅ do
3:     let e be the heaviest edge in E
4:     add e to M
5:     remove e and all edges incident to e from E
```

Show that the greedy algorithm has an approximation ratio of 1/2. *(10 Points)*

Sample Solution

(a) This is one possible solution



(b) Let x_A be the costs payed with card A by First Fit (FF). W.l.o.g let A be a card that is 'filled' before B by FF. When we pay the first item with card B, the item cost c is more then the remaining space left on card A, cause otherwise it would have been added to A. Hence $x_A > \ell - c$ and $x_B \geq c$ what proofs the hint $x_A + x_B > \ell$.

Let k be the number of cards used by NF: Since there are $\binom{k}{2}$ possible pairs of cards and each pair has value $> l$, it follows $l\binom{k}{2} < (k-1)\sum_A x_A$ (each card occurs in exactly $k-1$ pairs) and so $\sum_A x_A > l \cdot k/2$. Note that OPT can not better than using the full limit l on each card: $OPT \geq \sum_A x_A/l > k/2$. This implies a 2-approximation.

(c) **Algorithm Idea:** is to first find the maximum element in the middle column, i.e. in column $A[, n/2]$. Suppose the maximum column element is located at $A(i; n/2)$. If it is peak then we are done. Otherwise there are two cases:
Case 1: $A[i; n/2 - 1] > A[i; n/2]$. In this case we recursively find the peak in the submatrix containing the n rows and the first $n/2$ columns.
Case 2: $A[i; n/2 + 1] > A[i; n/2]$. In this case we recursively find the peak in the submatrix containing the n rows and the last $n/2$ columns.
Finally, for the base case, if we only have one column left then we simply return the maximum element in that column.

Correctness: First, we notice that if we pick an arbitrary column j, and assume that the global maximum element of that column is at row i i.e. $A[i][j]$, then if also $A[i][j]$ is a peak element, we are done. Otherwise, a peak element must exist in the left (resp. right) side of $A[i][j]$ in case $A[i][j-1] > A[i][j]$ (resp. $A[i][j] < A[i][j+1]$). To see this assume w.l.o.g. that $A[i][j] < A[i][j+1]$, then we iteratively go through each column $k = j+1., ..., m$ and find the global maximum element of column k. Now we have two cases:
Case 1: it is a peak element, thus we are done and we stop.
Case 2: it is smaller than its neighbor to the right (but will be larger than its left, top,and bottom neighbors). Thus we continue this same procedure with the next column $k+1$.

Finally, we notice that if we continued this procedure until the last column m (i.e. we still haven't found a peak element), then the global maximum element of column m must be a peak element. The reason is that since going from column $m-1$ to m means that the global maximum element of column $m-1$ (say at row i i.e. $A[i][m-1]$) was smaller than its right neighbor i.e. $A[i][m]$. Thus we notice that if say the global maximum element of column m is at row i', then $A[i'][m] > A[i][m] > A[i][m-1]$ and since $A[i][m-1]$ is the global maximum of column $m-1$, then $A[i'][m] > A[, m-1]$. Therefore, $A[i'][m]$ will be larger than its left neighbor; moreover, it is also larger than its top and bottom neighbor; and since its right missing neighbor is negative, thus it is still larger than it, we conclude $A[i'][m]$ is a peak element. Note that the case where

$A[i][j] < A[i][j+1]$ is symmetric.

And now to see why the recursion in the algorithm eventually gives a peak element, we notice the following that: after doing the recursive divide and conquer step, we are always checking if the global maximum of the middle column of the subarray we are working with is a peak element or not and if it wasn't, then we continue recursing on the bigger half side. The idea now is that this bigger half side* is a subarray that we can show that a peak element (for the whole input array) who is also a global maximum of its column always exists there and the algorithm is trying to find them in this binary search fashion.
To show this we roughly explain how: These bigger half side subarrays are the subarrays we are working with after each divide and conquer step of the algorithm, and they will have their left and right boundaries to be at most two (previously considered) "middle"' columns (of some previous subarrays considered in prior divide and conquer steps); and for these "middle" columns we have already checked whether their corresponding global max was a peak element or not and it wasn't. Thus we can now continue similarly to the argument in the observation above and reach that in the worst case the global maximum of either a boundary column of the original array or a column neighboring a "middle" boundary column will be a peak element in such subarray* and hence the original input array.

Running Time: It takes $O(n)$ time to find the max element on a column. Since the space where a potential peak is, gets halved in each iteration, only $O(\log n)$ steps are needed (this argument is similar to a binary search) to find the column of a peak. Hence $O(n \log n)$ iterations are required. Additional Note: There is an slightly different algorithm that quarters the matrix in each step and hence will find the peak in linear time.

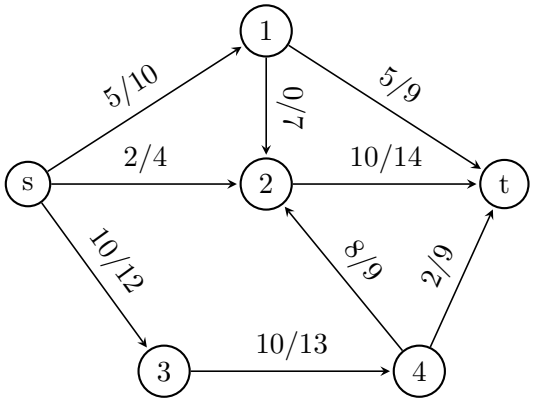
(d) Every edge e removed by greedy can remove at most 2 optimal edges incident to it (say we have two optimal edges e_1,e_2), hence $w(e) \geq w(e_i)$, where $i \in \{1, 2\}$.

And since the greedy algorithm is returns a maximal matching, then every optimal edge has to be incident to some greedy edge (or is a greedy edge). So $2w(M) = \sum_{e \in M} 2w(e) \geq \sum_{e' \in OPT} w(e') = w(OPT)$ and the proof is done.

Task 2: Miscellaneous Maximum Flow Questions *(24 Points)*

Note that the following three tasks are independent from each other.

(a) Consider the following flow network, where for each edge, the capacity (second number) and a current flow value (first number) are given. Draw the residual graph with all the residual capacities! In addition, give a best possible augmenting path (an augmenting path that improves the current flow from s to t by as much as possible) and the resulting flow value of the network (after augmenting along that path). *(6 Points)*



(b) Let G be a network with source s, sink t, and integer capacities. Prove or disprove the following statements:

- If all capacities are even then there is a maximum flow f such that $f(e)$ is even for all edges e. *(4 Points)*
- If all capacities are odd then there is a maximum flow f such that $f(e)$ is odd for all edges e. *(4 Points)*

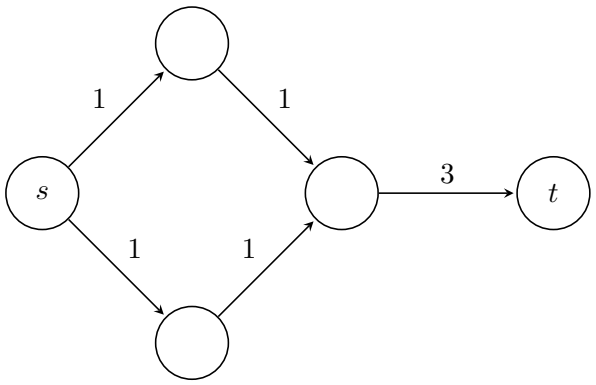
(c) Given a flow network with unit capacity edges i.e., a directed graph $G = (V, E)$, a source $s \in V$, and a sink $t \in V$ and $c_e = 1$ for every $e \in E$. You are also given a parameter $k \in \mathbb{N}$. The goal is to delete k edges so as to reduce the maximum $s - t$ flow in G by as much as possible. In other words, you should find a set of edges $F \subseteq E$ so that $|F| = k$ and the maximum $s - t$ flow in $G' = (V, E \backslash F)$ is as small as possible subject to this. Give a polynomial-time algorithm to solve this problem, argue correctness, and running time. *(10 Points)*

Sample Solution

(a) The best possible augmenting path is: $s \to 1 \to 2 \to 4 \to t$. This path increases the flow by 5. Hence, the resulting flow value is 22.

(b)
- Dividing all capacities by two, we obtain a network with integral capacities. Therefore, it has an integral maximum flow. Multiplying this flow by 2, we get an even maximum flow on the original network.

  Alternatively: think of Ford Fulkerson and notice that each time we find an augmenting path to augment its smallest residual edge capacity must be even. Thus in every step of the Ford Fulkerson algorithm, we add an even number to our current flow until no augmenting paths exist, that's when the Ford Fulkerson algorithm terminates producing an even flow.
- Counterexample:



(c) Find a min st-cut of the given network $G = (V, E)$. Let F be the set of cut edges.

We know that max flow in G has value $f = |F|$. If $|F| \leq k$, then by removing these edges we drop the max flow in G to 0. Otherwise,if $|F| > k$, we delete any subset of k edges of F from G. This results in a new flow network graph G' with a cut of capacity $f - k$ which is also the new min cut size, and hence G has max flow at most $f - k$.
This is true since otherwise if by removing k edges the new min cut size is $< f - k$, then this means that after adding the k edges back this means that in G we have a cut size of $< f$, hence there exists a smaller cut than our min cut of size f, a contradiction. It follows that reducing the max flow in G from value f to $f - k$, achieved by our removal of k edges of the min cut F of G, is indeed the biggest possible.

The runtime of the algorithm to find this subset of k edges is polynomial in the number of nodes, since we simply run the Ford-Fulkerson algorithm on G to find a min cut (since we have unit edge capacities, then whichever implementation give a polynomial in n algorithm).

Task 3: Amortized Analysis *(16 Points)*

Assume we have a sequence $a_1, \ldots, a_n$ of tree elements that we want to store in a balanced binary tree. That is, for any $i = 1, \ldots, n$, the height of the tree after inserting element a_i equals $\lfloor \log i \rfloor$. We hence fill a (initially empty) binary tree with the elements level by level. Additionally, we want to store in each node the current height of the tree. For this purpose, whenever the height of the tree increases (i.e., a new level is started), we increase the counter of each node by 1.
So we have two operations: If inserting a_i does not increase the height of the tree, we simply insert it by applying `insert`(a_i). If inserting a_i increases the height of the tree, we apply `insert_and_raise`(a_i) which inserts a_i and increases the counter of each node.

Use either the accounting method or the potential function method to show that both operations have constant amortized cost.

Hint: We do not require any additional property for the tree like the min-heap property or being a binary search tree. Hence, inserting an element takes $O(1)$ by using e.g. the array-implementation of balanced binary trees. For simplicity, you may assume that `insert` *has cost exactly 1 and* `insert_and_raise`(a_i) *has cost* $1 + i$.

Sample Solution

First observe that we apply `insert_and_raise`(a_i) iff i is a power of two. If at step i we have an `insert_and_raise` operation, the number of `insert` operations since the last `insert_and_raise` is $i/2 - 1$.

Accounting method: For each `insert` we pay an extra amount of 2. Then before an `insert_and_raise` operation at step i we have $2(i/2-1) = i-2$ on our bank account. The actual cost of `insert_and_raise` is $1 + i$. If we use the money from the bank account, we still need to pay 3. So we have that both operations have amortized cost 3.

Potential Function: We define the potential function as twice the number of nodes on the bottom level. Each `insert` operation increases the potential by 2. Before an `insert_and_raise` operation at step i, the potential equals i. `insert_and_raise` sets the potential to 2, i.e., decreases it by $i - 2$. As the actual cost of `insert` is 1 and the cost of `insert_and_raise` is $1+i$, we have that both operations have amortized cost 3.

Task 4: Online Independent Set *(16 Points)*

Given an undirected, simple graph $G = (V, E)$, an **independent set** $I \subseteq V$ is a subset of nodes such that for all $u, v \in I$ we have $\{u, v\} \notin E$. The maximum independent set problem is the problem of finding an independent set of maximum size.

(a) Show that a maximum independent set of a tree with n nodes has size at least $n/2$. *(4 Points)*

Now we consider an online version of this problem: Initially, we are given a set of nodes V without edges. Then edges of G arrive one by one in an online fashion and we have to maintain an independent set I of the graph. We start with $I = V$ and after each arrival of an edge, we can decide to remove nodes from I. If a node has been removed from I, we can not add it again later.

(b) Show that even if we know that G is a tree with n nodes, no deterministic online algorithm for this problem has a better competitive ratio than $n/2$. *(12 Points)*

Hint: Use part (a) and show that for any $n > 0$, no deterministic online algorithm can guarantee an independent set of size > 1.

Sample Solution

(a) A tree is a bipartite graph. In a bipartite graph $(A \cup B, E)$, both A and B are independent sets and one has size at least $n/2$.

(b) Let $\mathcal{A}$ be a deterministic algorithm. We connect the nodes in V to a tree T such that in each step, the set I which $\mathcal{A}$ maintains contains at most 1 node from T (that is, at most 1 non-isolated node from V). Finally, when there are no isolated nodes left, we have $|I| = 1$ but a maximum independent set has size at least $n/2$ (by a)). We construct T inductively. Initially, $T = \emptyset$. Now assume T contains $k - 1$ edges and $|I \cap T| \leq 1$. If $|I \cap T| = 0$, we connect a node $u \in V \setminus T$ with an arbitrary node in T. After this step, if $u \in I$, we have $|I \cap T| = 1$, otherwise $|I \cap T| = 0$. Now assume $|I \cap T| = 1$ (before step k). Then we connect a node $u \in V \setminus T$ with the node $v \in I \cap T$. If u was in I before, $\mathcal{A}$ has to remove either u or v (or both) from I such that either 0 or 1 nodes remain in $I \cap T$. As we never add an edge between two tree nodes, we do not obtain cycles and hence T is indeed a tree.

Task 5: Set of Strangers *(12 Points)*

Consider a group of n people. Suppose there are $2n$ pairs of people of this group that know each other. We may model this situation as an undirected graph $G = (V, E)$, in which the vertices are people and we draw an edge between two people if and only if they know each other. The total number of edges is therefore $2n$. We want to find a huge group of strangers, i.e., a set of nodes where no edges connect nodes within this set. To achieve this we use the following randomized algorithm:

- First, every node joins a set $S \subseteq V$ independently with probability $1/4$.
- Let $E_S \subseteq E$ be the set of edges where both endpoints are in S.
- Now, iterating through the edges $e \in E_S$ and deleting one arbitrary endpoint of e from S (if not already deleted) gives us the final set of strangers S'.

Show that $E[|S'|] \geq n/8$.

Hint: The number of people getting deleted from S can not be more than $|E_S|$.

Sample Solution

It is clear by the construction that $E[|S|] = n/4$. For an edge $e = (u, v)$, the probability that $e \in E_S$ is, due to independence, the product of the probabilities that $u \in S$ and $v \in S$, i.e., $P(e \in E_S) = 1/16$. Summing over all $2n$ edges gives $E[|E_S|] = 2n \cdot 1/16 = n/8$. Using the statement of the hint, we can end the proof as follows $E[|S'|] \geq E[|S|] - E[|E_S|] = n/8$.

Task 6: Dynamic Programming *(10 Points)*

Given an array A of length n containing integer values. We define a *longest increasing subsequence* (LIS) of A to be a sequence $0 \leq i_0 < i_1 < ... < i_\ell < n$ such that $A[i_0] < A[i_1] < ... < A[i_\ell]$ and ℓ is as long as possible. For example, if $A = [10; 13; 2; 4; 6; 4; 9]$, then an LIS is $i_0 = 2; i_1 = 3; i_2 = 4; i_3 = 6$ which is of length 4 and corresponding to the subsequence 2; 4; 6; 9. Notice that a longest increasing subsequence need not need to be unique.

Devise a dynamic programming algorithm that returns the length of a longest increasing subsequence in A and runs in $O(n^2)$. Argue its correctness and running time (and don't forget to write down the recursive relation).

Sample Solution

We notice that the LIS of a nonempty array $[A(0); ...; A(i)]$ i.e. $D[i]$ will have length at least one, hence it has a last element $A(j)$ for some $0 \leq j \leq i$. Thus denote $D'[i]$ to be the length of the LIS of an array $[A(0); ...; A(i)]$ such that the last element of the array i.e. $A[i]$ is used. So we conclude that $D[i] = \max_{0 \leq j \leq i} D'[j]$ where $D'[j] = 1 + \max_{0 \leq k < j, A[k] < A[j]} D'[k]$ and if no such k exists then $D'[j] = 1$.
A proposed dp algorithm using memorbrebrebrebization is the following:

Algorithm 2 `LIS`(n) ▷ Given an array A with n elements and assume we have a global dictionary `memo` initialized with `Null`

if $n = 1$ **then return** 1 ▷ base case
if `memo[`n`]` $\neq$ `Null` **then return** `memo[`n`]` ▷ LIS was computed before
`memo[`n`]` $\leftarrow \max_{0 < j \leq n-1} \left(\max_{0 \leq k < j, A[k] < A[j]} 1 + \texttt{LIS}(k+1) \right)$ ▷ Memoization
return `memo[`n`]`

Correctness: We argue by induction on i and prove the following statement: on input $[A[0]; ...; A[i]]$ $D'[i]$, the length of the LIS including $A[i]$,- is correct. This will prove correctness because the algorithm returns at the end $\max_{0 \leq j \leq i} D'[j]$, i.e. the longest increasing subsequence that ends at any location.

Base step: $D'[0] = 1$ is correct, because we can take the first element to be a subsequence of itself, with is of length one.
Induction step: Suppose $D'[0], ..., D'[i-1]$ are all correct. Now for $D'[i]$ either the longest subsequence is $A[i]$ itself which is of length 1, or $D'[i]$ uses a subsequence that starts earlier and ends at i, with the prior index included before i is $k \geq 0$, and this can only occur if $A[k] < A[i]$, and if so, its length is $D'[k] + 1$. The recursion takes the maximum over these possibilities, so $D'[i]$ is correct. And since we take the maximum of $D'[i]$ over all i at the end and each $D'[i]$ is correct, one of them should be the LIS of the input.

Running time: Due to memoization we compute each value `memo`$[k]$ for $k \in \{1, .., n\}$ at most once. Each computation of `memo`$[k]$ costs at most $\mathcal{O}(n)$ in the current step (determining the maximum of at most n numbers) not counting the cost of recursive calls. The total cost is therefore $\mathcal{O}(n^2)$.