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

Algorithm Theory

22nd of March 2022, 9:00 - 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 **five handwritten, single-sided 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	**Total**
Maximum	31	40	18	15	16	120
Points						

Task 1: Short Questions I *(31 Points)*

(a) Assume you have a freshly initialized data structure D that provides some operation `op`, which modifies D in some way and has an *amortized* running time of $\mathcal{O}(1)$. Consider a series of n executions of `op` on D. State how large the asymptotic *worst case* running time of a single one of the n `op` operations can be *at most* as a function of n. *(4 Points)*

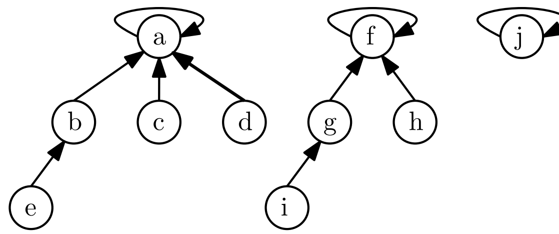
(b) Consider a *complete* binary tree T with $n = 2^k - 1$ nodes for some $k \in \mathbb{N}$ (that means, each level of T has the maximum number of nodes). Nodes have identifiers $\{1, \dots, n\}$. Assume we have two operations `op1`(i) and `op2`(i). The running time of `op1`(i) is linear in the number of *ancestors* of node i in T. The running time of `op2`(i) is linear in the number of *descendants* of node i (i.e., the number of all nodes which have node i as ancestor). Consider the following sequences of operations:

(i) `op1(1)`, `op1(2)`, ..., `op1`(n)

(ii) `op2(1)`, `op2(2)`, ..., `op2`(n)

Prove that each sequence has a total running time of $\Theta(n \log n)$. *(10 Points)*

(c) Consider the following state of a union-find data structure represented as disjoint-set forest where the *current* rank of each node equals its in-degree.



Conduct the following operations *sequentially* on the union-find data structure (the result of the prior operation is the input of the next). Give the state of the data structure after each operation.

(i) `union`(f, j) using the *union-by-rank* heuristic *(3 Points)*

(ii) `union`(b, g) using the *union-by-rank* heuristic *(3 Points)*

(iii) `find`(i) using *path compression* *(3 Points)*

(d) Let A be an array of length $n \geq 2$ containing integers. Given an algorithm that computes the largest difference of two values $A[j] - A[i]$ with $i, j \in \{0, \dots, n-1\}$ *and* $j > i$ within a running time of $\mathcal{O}(n \log n)$. Explain why your algorithm is correct and has the claimed runtime. (Two bonus points are granted for a solution in $\mathcal{O}(n)$). *(8 Points)*

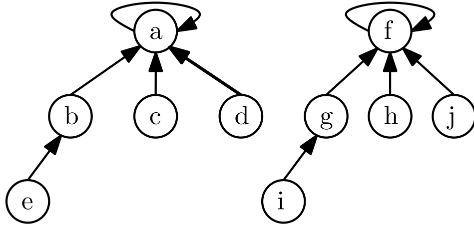
Sample Solution

(a) $\mathcal{O}(n)$.

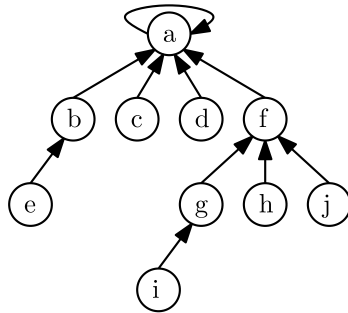
(b) Since T is complete, it has height $k \in \Theta(\log n)$. Therefore the number of ancestors per node is $\mathcal{O}(\log n)$ thus a single `op1`(i) takes at most $\mathcal{O}(\log n)$, which means the total running time of sequence (i) can be upper bounded by $\mathcal{O}(n \log n)$. Furthermore notice that T has $\frac{n+1}{2} \in \Theta(n)$ leafs, each of which has $\Theta(\log n)$ ancestors. This means that the total running time of sequence (i) is lower bounded by $\Omega(n \log n)$.

Finally, the running time of sequence (ii) can be seen as follows. We divide the running time of `op2`(i) by the number of descendants of node i and charge each descendant its *constant* part of running time of `op2`(i). Then in the whole sequence (ii) each node gets charged at most $\mathcal{O}(1)$ by each of its ancestors, i.e., the total running time of sequence (ii) is asymptotically the same as that of sequence (i).

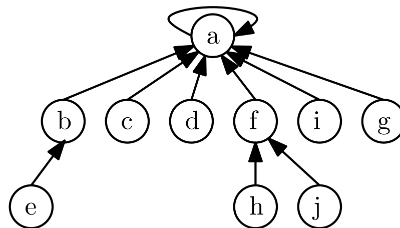
(c) (i) after `union`(f, j) :



(ii) after `union`(b, g):



(iii) after `find`(i):



(d) Consider the following divide and Conquer approach. If $n = 2$ return the difference of the two elements. Let $m := \lfloor \frac{n}{2} \rfloor$. Consider the two partial arrays $B := A[0, m]$ and $C := A[m+1, n-1]$. Solve the problem recursively on A and B, leading to solutions b, c respectively. Then (naively) compute the maximum entry in B and the minimum entry in C and let a be the difference. Return the maximum of $\{a, b, c\}$. The running time is $T(n) = 2T(n/2) + \mathcal{O}(n)$ which solves to $T(n) \in \mathcal{O}(n \log n)$ using the master theorem.

Note that we can improve the running time in each recursion to $O(1)$ as follows. In each recursive call we also return the maximum and minimum in B and C from which we can then compute the value a and the new maximum and minimum in constant time. Instead of defining (and copying) new arrays B and C we can maintain left and right pointers in each recursion which denote the subarray of A we are currently considering. This improves the running time to $T(n) = 2T(n/2) + \mathcal{O}(1)$ which solves to $T(n) \in \mathcal{O}(n)$.

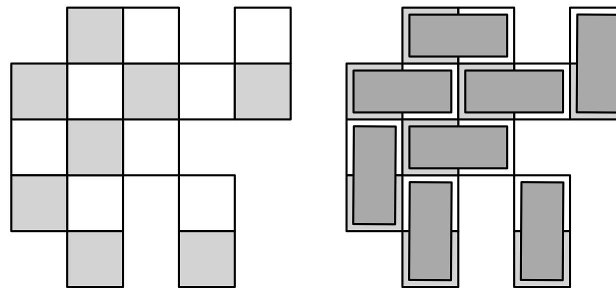
Task 2: Short Questions II *(40 Points)*

(a) Let A be an array of length n containing pairwise distinct, positive integers. We define a *non-adjacent* subset of A as a selection elements from A such that no elements in the subset are next to each other in A. A *maximum* non-adjacent subset has the additional property that its sum of elements is *maximum* among all non-adjacent subsets. The goal is to compute the sum of a maximum *non-adjacent* subset. Give an algorithm that accomplishes this in $\mathcal{O}(n)$ time. *(9 Points)*

(b) Let $G = (V, E)$, be an unweighted, undirected graph and $u, v, w \in V$. Give an *efficient* algorithm that determines whether there is a walk from u to w that passes through v such that no edge of G is traversed more than once. Explain why your algorithm is correct. *(8 Points)*

(c) Consider an incomplete $n \times n$ checkerboard, i.e., where some tiles are cut out. The incomplete checkerboard is given by an $n \times n$ array C with $C[i][j] = 0$ if the tile at position $(i, j) \in \{0, \ldots, n-1\}^2$ has been cut out, else $C[i][j] = 1$. We want to answer the question whether we can place domino pieces, each of which covers *exactly* two adjacent tiles on the checkerboard, such that all tiles are covered (for instance in the example below the answer is yes). More precisely, we want to cover every tile (i, j) with $C[i][j] = 1$ with some domino piece, such that domino pieces *do not overlap* and *only cover existing tiles* ($C[i][j] = 1$ with $i, j \in \{0, \ldots, n-1\}^2$). Give an efficient algorithm that answers this question and argue why it is correct. *(7 Points)*

Remark: You may use algorithms from the lecture as black-box.



(d) Consider the knapsack problem with n items and a knapsack of size W, where each item i has value v_i and weight w_i of *at most* $W/2$. Give a $\frac{1}{2}$-approximation algorithm with running time $\mathcal{O}(n \log n)$ for this special case of the knapsack problem. Prove the approximation ratio. *(11 Points)*

(e) We are given a finite sequence of positive integer numbers in an online fashion and would like to select a number from it that is as large as possible. The numbers are presented one by one and after we have seen a number we immediately and irrevocably have to decide whether to select or reject it, without knowing the remaining numbers.

Show that there is no deterministic strictly c-competitive online algorithm for any $c > 0$. *(5 Points)*

Sample Solution

(a) **Algorithm 1** `rec`(A, n) ▷ assume we have a global dictionary `memo` initialized with `Null`

```
if n = 0 then return 0
if n = 1 then return A[0]                                    ▷ base cases
if memo[n] ≠ Null then return memo[n]                        ▷ result was computed before
memo[n] ← max(A[n] + rec(A, n−2), rec(A, n−1))               ▷ Memoization
return memo[n]
```

(b) Let v correspond to the source and create a new sink node t that has an edge to both u and w then solve the max flow problem. If and only if the flow has size at least 2 there is an edge disjoint uw-path via v. If the flow is at least 2, then such a path is obtained by concatenating the edge

disjoint two flow paths from v to t leaving out the last edge to t. Else, there is a cut of size at most 1 that separates v from u and w (by the max-flow min-cut theorem). As any uw-path via v, has to cross that cut twice, no such path can be edge disjoint.

(c) The problem corresponds to finding a perfect matching in a bipartite graph, where the nodes of the bipartition correspond to the white and black tiles of the checkerboard respectively with an edge between horizontally or vertically adjacent tiles. The incomplete checkerboard can be covered if and only if there is a perfect matching. We have seen in the lecture that the question whether a bipartite graph has a perfect matching can be answered efficiently by transforming it into a flow problem and using the Ford-Fulkerson algorithm.

(d) We use a simple greedy strategy: For each item we compute the value-weight ratio $r_i = v_i/w_i$. W.l.o.g. we assume that the items are sorted by r_i (sorting can be done in $\mathcal{O}(n \log n)$). Then, we pick items $1, \ldots, k$ such that $\sum_{i=1}^{k} w_i \leq W$ but $\sum_{i=1}^{k+1} w_i > W$ or $k = n$. This is possible in $\mathcal{O}(n)$, by packing items until the next item does not fit anymore, or all items are packed.

Let $G = \{1, \ldots, k\}$ be our greedy solution and let $O \subseteq \{1, \ldots, n\}$ be the optimal solution. Let $w_O = \sum_{i \in O} w_i$, $w_G = \sum_{i \in G} w_i$ and $v_O = \sum_{i \in O} v_i$, $v_G = \sum_{i \in G} v_i$ be the total weights and values of both solutions. If all items fit into the knapsack, then clearly the greedy strategy is optimal. If not, then $w_G \geq W/2$, as otherwise in our special knapsack variant the next item would still fit into the knapsack. As we chose our items greedily, the value-weight ratio $r_G = v_G/w_G$ of G is larger than $r_O = v_O/w_O$. Hence

$$v_G = r_G \cdot w_G \geq \tfrac{r_G \cdot W}{2} \geq \tfrac{r_O \cdot W}{2} \geq \tfrac{r_O \cdot w_O}{2} = \tfrac{v_O}{2}.$$

(e) Let ALG be a deterministic online algorithm. We define ALG to be strictly c-competitive if $\text{ALG} \geq c \cdot \text{OPT}$ for a $c > 1$. Clearly, ALG can not be c-competitive for a $c > 1$. Let $c \leq 1$. Assume the first number is 1. If ALG selects it and the second number is $\frac{1}{c} + 1$, we have $\text{ALG} = 1 < c\left(\frac{1}{c} + 1\right) \leq c \cdot \text{OPT}$. If ALG rejects it and all remaining numbers are $c/2$, we have $\text{ALG} = c/2 < c = c \cdot \text{OPT}$.

Task 3: Correct Bracket Expressions *(18 Points)*

A string over the symbols $\{(,)\}$ is called a *correct* bracket expression, if it contains the same number of opening and closing brackets and *every* prefix of that string has at least as many opening brackets as it has closing ones (or, analogously, every suffix has at least as many closing brackets as opening ones).

Let B be a string of length $2n$ that consists of exactly n opening '(' brackets and n closing ')' brackets but in arbitrary order. We want to compute the *minimum number of swaps* of pairs of symbols in B in order to obtain a correct bracket expression. For instance, for $B =$ ")())()((" the answer is 1, because we can turn B into a correct expression by swapping the first and the last bracket.

(a) Give an algorithm with running time $\mathcal{O}(n)$ that computes the minimum number of swaps to make B correct. *(8 Points)*

(b) Argue that your algorithm is correct. *(10 Points)*

Sample Solution

(a) We use a greedy approach. Let i be the smallest value, such that for the prefix of B up to symbol i, the number of opening brackets is one smaller than the number of closing brackets. Analogously, let j be the largest value such that for the suffix of B starting from symbol j, the number of closing brackets is one smaller than the number of opening brackets.

Note that at position i there must be closing bracket and at position j there must be an opening bracket, as otherwise the prefix and suffix could be made shorter. We swap the symbols on position i and j, increase the counter by one and repeat, until no such prefix and suffix is found. By continuing the search from the current positions of i, j and stopping when $i = j$, we need to sweep B just once, which takes $\mathcal{O}(n)$ time. An algorithm description in this spirit is sufficient for full points.

The following pseudocode gives more detail:

Algorithm 2 swappings($B = b_1, \ldots, b_{2n}$)

```
i ← 1, j ← 2n
p, s ← 0                  ▷ to track difference in opening closing brackets in prefix and suffix
c ← 0                                                              ▷ number of swaps
while i < j do
    while p ≥ 0 do                                       ▷ while prefix has surplus '('
        i ← i + 1
        if b_i = '(' then p ← p + 1
        else p ← p − 1
    while s ≥ 0 do                                       ▷ while suffix has surplus ')'
        j ← j + 1
        if b_j = ')' then s ← s + 1
        else s ← s − 1
    c ← c + 1, p ← p + 1, s ← s + 1    ▷ pretend that a swap (i, j) took place and adjust p, s
return c
```

(b) In each iteration of the outer loop, the swaps that we have made (or pretended to make) take care that for indices i, j all prefixes up to i have at least as many opening brackets as closing ones, and all suffixes from j on have at least as many closing brackets as opening ones. The algorithm terminates when $i \geq j$, so there is an index k such that the above is true for $i = j = k$ meaning we have the correctness condition for suffixes and prefixes up to parameter k. As the number of opening and closing brackets in B is the same, this means that after doing the swaps (which we only pretend in the code above to do) our bracket expression is correct after the algorithm

terminates. Thus the number we return is an upper bound for the number of swaps that is required to make B correct. We still have to prove that it is not possible to be better than that. We will consider our solution and an optimal one and make an exchange argument.

Let S be the series of indices $(i_1, j_1), \dots, (i_c, j_c)$ that our algorithm swaps (or swaps that are counted) sorted by the first coordinate. Assume there is a different series of swaps $S^* = \{(i^*_1, j^*_1), \dots, (i^*_c, j^*_c)\}$ (sorted by the first coordinate), with $|S^*| \leq |S|$ that makes B correct as well. Let $(i, j) \in S$ be the first swap where the series S and S^* differ. Note that before (i, j) is applied the bracket expression can not be correct yet as can be seen from our algorithm, thus S^* can not be finished yet and there must be a next swap $(i^*, j^*) \in S^*$ with $i \neq i^*$ or $j \neq j^*$. By our greedy condition, the prefixes shorter than i and suffixes shorter than $n - j + 1$ do not violate the required condition. The swap (i, j) only increases the according difference in opening and closing brackets for all prefixes of length at least i and the all suffixes of length at least $n - j + 1$, which are the ones that need to be "fixed". It is therefore possible to replace (i^*, j^*) in S^* with (i, j) and S^* still remains a valid series of swaps which makes B correct. Doing this iteratively we can transform the swaps of S^* into those of S, thus $|S^*| \geq |S|$.

Task 4: Maximum k-cut *(15 Points)*

Let $G = (V, E)$ be an undirected graph. The goal in the maximum k-cut problem is to partition V into $V_1, \ldots, V_k$, such that the number of cut-edges, i.e., edges $\{u, v\}$ with $u \in V_i, v \in V_j, i \neq j$, is maximized.

(a) Give an *efficient* deterministic algorithm with an approximation ratio $1 - \frac{1}{k}$. *(6 Points)*

(b) Prove the approximation ratio of your algorithm. *(9 Points)*

Sample Solution

(a) Initially, $V_i = \emptyset$ for $i \in \{1, \ldots, k\}$. Let $V = \{v_1, \ldots, v_n\}$. For $\ell = 1..n$, we add v_ℓ to the set V_i such that it has a minimum number of neighbors in the same set V_i (break ties arbitrarily).

(b) In each step we maintain the condition that the number of edges between nodes in different sets V_i, V_j is at least $1 - \frac{1}{k}$ of the total number of edges m_ℓ in the graph G_ℓ induced by the nodes $v_1, \ldots, v_\ell$ that have been added to some V_i so far.

Initially this is trivially true since all sets V_i are empty. Presume the claim is true for $G_{\ell-1}$, i.e., we have $(1 - \frac{1}{k})m_{\ell-1}$ nodes in the k-cut. Let d be the degree of v_ℓ. Thus, by adding v_ℓ, there are d new edges appearing in G_ℓ.

Note that, since there are k sets there must be a set V_i in which v_ℓ has at most $\frac{d}{k}$ neighbors, as otherwise the degree of v_ℓ would be larger than d. Thus, of the newly incoming d edges at least $d - \frac{d}{k}$ are in the k-cut. Thus the total number of edges in the k-cut of G_ℓ after adding v_ℓ is at least $(1 - \frac{1}{k})(m_{\ell-1} + d) = (1 - \frac{1}{k})m_\ell$.

The claim follows inductively for $\ell = n$, i.e., the final k-cut contains a $(1 - \frac{1}{k})$ portion of all edges in G. The approximation ratio immediately follows as the best solution can clearly not have more than $|E|$ edges.

Task 5: Randomized Algorithms *(16 Points)*

(a) Assume we are given a randomized approximation algorithm $\mathcal{A}$ for the minimum vertex cover problem. Concretely, $\mathcal{A}$ is a *Monte Carlo* algorithm with (deterministic) runtime $T(n)$ and parameter functions $f(n)$ and $p(n)$ with the following property: Given a graph G from a family of graphs with n nodes, $\mathcal{A}$ outputs a vertex cover S of G such that with probability at least $p(n)$, we have $|S| \leq f(n)$.

Describe a *Las Vegas* algorithm $\mathcal{B}$ for minimum vertex cover approximation that given a n-node graph G from that family, always outputs a vertex cover S of G with $|S| \leq f(n)$, and finishes in time $\mathcal{O}(T(n) + f(n))$ with probability at least $p(n)$. Prove the randomized guarantee for the running time. *(6 Points)*

(b) Given an array A of length n containing bits $\{0, 1\}$. We want to test if all entries in A are 0 or if it is "sufficiently far away" from an an array containing only 0.

Concretely, we want an algorithm that returns "almost-zero" if all entries of A are 0. We want to return "not-zero" if *more than* $\sqrt{n}$ (for simplicity you can assume that n is a square number) entries in A containing 1. If the number of 1's in A is in the interval $[1, \sqrt{n}]$, then we may return either answer.

Give a sub-linear (i.e., $o(n)$ running time) randomized algorithm that accomplishes this with probability of success at least $1 - \frac{1}{n}$. Argue the correctness and prove the success probability. *(10 Points)*

Sample Solution

(a) We repeat $\mathcal{A}$ until it returns a set S with $|S| \leq f(n)$. Running $\mathcal{A}$ and checking whether $|S| \leq f(n)$ takes $\mathcal{O}(T(n) + f(n))$. With probability at least $p(n)$, one run is sufficient.

(b) Our algorithm picks k indices $i_1, \ldots, i_k \in \{1, \ldots, n\}$ independently, uniformly at random. It returns "not-zero" if there is an i_j with $A[i_j] = 1$, else it returns "almost-zero".

Our algorithm clearly always succeeds if A contains either only 0's or only 1's. If the number of 1's is in the range $[2, \sqrt{n}]$ we are allowed to return any result, so the algorithm can not fail either.

The only case where our algorithm *can* fail is when we return "almost zero" even though there are *more than* $\sqrt{n}$ one-entries in A. This is the case if we never pick an index i_j with $A[i_j] = 1$. To make the probability for that sufficiently low we choose $k \geq \sqrt{n} \ln n$.

$$\mathbb{P}(\text{failure}) = \left(1 - \tfrac{1}{\sqrt{n}}\right)^k \leq e^{-\frac{k}{\sqrt{n}}} \leq e^{-\ln n} = \frac{1}{n}$$

The runtime is $\mathcal{O}(k) = \mathcal{O}\left(\sqrt{n} \ln n\right) \subseteq o(n)$.