Here are some extra practice problems taken from past semesters. You are welcomed to check your solutions with me or discuss them in the telegram group chat.

1. **Greedy Algorithms** (AY 24/25 Sem 2). You are arranging a summer camp for n students. Student i stays during the period $[a_i, b_i]$, where a_i and b_i are positive integers such that $a_i \leq b_i$. Assume that students who arrive earlier also leave no later: $a_1 \leq a_2 \leq \ldots \leq a_n$ and $b_1 \leq b_2 \leq \ldots \leq b_n$. You would like to hold a number of parties in such a way that every student can attend at least one party. (Each party occurs at an integer time t. Student i can attend any party that occurs at a time t such that $a_i \leq t \leq b_i$.) Since parties are expensive, you would like to minimize the number of parties held. Design an analyze an algorithm to compute this number, and state the running time of your algorithm in terms of n using the O-notation.

   Optimal substructure property: If b_1 is contained in an optimal solution S, then $S = \{b_1\} \cup S'$, where S' is an optimal solution for all intervals that do not contain b_1.
   Proof. Note that S' must satisfy all intervals that do not contain b_1. If it is not an optimal solution for these intervals, say S'' is a better solution, then $S'' \cup \{b1\}$ would be a better solution than S for all intervals, a contradiction.

   Greedy choice property: There is an optimal solution such that one party occurs at time b_1.
   Proof. Consider any optimal solution, which must contain a time in $[a_1, b_1]$. If this time is b_1, we are done. Else, we can replace it with b_1; since $b_1 \leq b_i$ for every other i, any student who is satisfied with the previous time is also satisfied with b_1, so the new solution is also optimal.

   The two claims together directly yield a greedy algorithm. First, we include b_1 and remove all intervals that contain b_1; this can be done by going through $a_2, a_3, \ldots$ until we find $a_i > b_1$ (which also means that $a_{i+1}, a_{i+2}, \ldots > b_1$). Then we proceed with the remaining intervals in a similar fashion. Since the intervals are already sorted and we process each of them only once, the algorithm runs in time $O(n)$.

2. **Greedy Algorithms** (AY 23/24 Sem 2). You are given n events where each takes one unit of time. Event i will provide a profit of g_i dollars ($g_i > 0$) if started at or before time t_i where t_i is an arbitrary real number. (Note: If an event is not started by t_i then there is no benefit in scheduling it at all. All events can start as early as time 0.) Also only one event can be scheduled at any particular time interval. Give as efficient algorithm as you can, to find a schedule that maximizes the total profit.

   Sketch. First, we sort the jobs according to $\lfloor t_i \rfloor$ (sorted from largest to smallest). Let time t be the current time being considered (where initially $t = \lfloor t_n \rfloor$). All jobs i where $\lfloor t_i \rfloor = t$ are inserted into a priority queue with the profit g_i used as the key. An extractMax is performed to select the job to run at time t. Then t is decremented and the process is continued. Clearly the time complexity is $O(n \log n)$. The sort takes $O(n \log n)$ and there are at most n insert and extractMax operations performed on the priority queue, each which takes $O(\log n)$ time.

   Greedy choice property: Suppose that event 1 is the event with the latest deadline (and if there are multiple such events, choose the one with the largest profit). There exists an optimal solution that schedules event 1 at time $\lfloor t_i \rfloor$.
   Proof. Consider an optimal solution S in which $x + 1$ events are scheduled at times $0, 1, \ldots, x$. Let event k be the last job run in S. The greedy schedule will run event 1 last (at time $\lfloor t_1 \rfloor$). From the greedy choice property we know that $\lfloor t_1 \rfloor \geq \lfloor t_k \rfloor$.

   - Case 1: $\lfloor t_1 \rfloor = \lfloor t_k \rfloor$: By our greedy choice, we know that $g_1 \geq g_k$. If event 1 is not in S then we can just replace event k by event 1. The resulting solution S' is at least as good as S since $g_1 \geq g_k$. The other possibility is that event 1 is in S at an earlier time. Since $\lfloor t_1 \rfloor = \lfloor t_k \rfloor$, we can switch the times in which they run to create a schedule S' which has the same profit as S and is hence optimal.
   - Case 2: $\lfloor t_1 \rfloor > \lfloor t_k \rfloor$: In this case, S does not run any event at time $\lfloor t_1 \rfloor$ since job k was its last job. If event 1 is not in S, then we could add it to S contradicting the optimality of S. If event 1 is in S we can run it instead at time $\lfloor t_1 \rfloor$ creating a schedule S' that makes the greedy choice and has the same profit as S and is hence also optimal.

   Optimal substructure: Let P be the original problem of scheduling events $1, \ldots, n$ with an

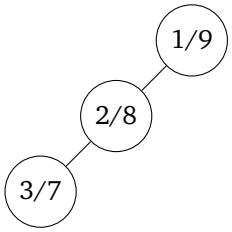
optimal solution S. Given that event 1 is scheduled first we are left with the sub problem P' of scheduling events $2, \ldots, n$. Let S' be an optimal solution to P'. Clearly profit(S) = profit$(S')+g_1$ and hence an optimal solution for P includes an optimal solution to P'.

3. **Greedy Algorithms** (AY 23/24 Sem 2). Given a set of keys $1, 2, \cdots, n$, where the key i has weight w_i. The weight of the key reflects how often the key is accessed, and thus heavy keys should be higher in the tree. The Optimal Binary Search Tree problem is to construct a binary-search tree for these keys, in such a way that

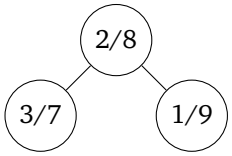
$$wac(T) = \sum_{i=1}^{n} w_i d_i$$

is minimized, where d_i is the depth of key i in the tree (note: here we assume the root has a depth equal to one). This sum is called the *weighted access cost (wac)*. Consider the greedy heuristic for Optimal Binary Search Tree: for keys $1, 2, \cdots, n$, choose as root the node having the maximum weight. Then repeat this for both the resulting left and right subtrees.
Prove or disprove that the greedy heuristic is optimal.

Solution. The greedy heuristic is not optimal. Apply this heuristic to keys 1, 2, 3 with respective weights 9, 8, 7. It results in the following tree with wac $9 \cdot 1 + 8 \cdot 2 + 7 \cdot 3 = 46$:



However, the following tree gives a smaller wac of $8 \cdot 1 + 9 \cdot 2 + 7 \cdot 2 = 40$:



4. **Amortized Analysis** (AY 23/24 Sem 2). You are asked to maintain a list L while supporting the following two operations:

   - Insert(x, L): Insert an integer x in the list L (cost is 1).
   - ReplaceSum(L): Compute the sum of all the integers in L. Then remove all these integers, and finally just store the computed sum in L (cost is length of the list L).

   Show that the amortized cost of both Insert and ReplaceSum operation is $O(1)$. *(As an exercise, you should try to prove it twice, once using accounting method and another time using potential method.)*

Sketch. For accounting method, charge \$2 for each Insert and \$2 for each ReplaceSum.
For potential method, define $\phi(t)$ = number of elements in L.

5. **Amortized Analysis** (AY 23/24 Sem 2). In the Set Union problem we have n elements, that each are initially in n singleton sets, and we want to support the following operations:

   - Union(A, B): Merge the two sets A and B into one new set $C = A \cup B$ destroying the old sets.
   - SameSet(x, y): Return true, if x and y are in the same set, and false otherwise.

   We implement it in the following way. Initially, give each set a distinct color. When merging two sets, recolor the smaller (in size) one with the color of the larger one (break ties arbitrarily). Note, to recolor a set you have to recolor all the elements in that set. To answer SameSet queries, check if the two elements have the same color. (Assume that you can check the color an element in O(1) time, and to recolor an element you also need O(1) time. Further assume that you can know the size of a set in O(1) time.)
   Use Aggregate method to show that the amortized cost is $O(\log n)$ for Union. That means, show that any sequence of m union operations takes $O(m \log n)$ time. (Note, we start with n singleton sets.)

Solution. Observe that each element can be recolored at most $\log n$ times. This is because since we are recoloring smaller set, an element is recolored k times means it was part of smaller set k times, and each time the size doubles. More specifically, if we consider all these k unions where that element is part of the smaller set, and let $s_1, s_2, \ldots, s_k$ be the sizes of those smaller sets. Clearly $s_i \geq 2s_{i-1}$, and so $k \leq \log n$.
Also notice that any sequence of m union operations can involve at most $2m$ many elements. Thus total number of recoloring is bounded by $O(m \log n)$.

6. **Amortized Analysis** (AY 23/24 Sem 2). Suppose Alice insists Bob to maintain a dynamic table (that supports both insertion and deletion) in such a way its size must always be a Fibonacci number. She insists on the following variant of the rebuilding strategy. Let F_k denote the k-th Fibonacci number. Suppose the current table size is F_k. After an insertion, if the number of items in the table is F_{k-1}, allocate a new table of size F_{k+1}, move everything into the new table, and then free the old table. After a deletion, if the number of items in the table is F_{k-3}, we allocate a new hash table of size F_{k-1}, move everything into the new table, and then free the old table.
Use either Potential method or Accounting method to show that for any sequence of insertions and deletions, the amortized cost per operation is still $O(1)$.

Solution. Suppose the current table size is F_k, then there must be at most F_{k-1} elements present in the table. Consider the following charging scheme for insertion: Charge \$4 for each insertion, use \$1 for insertion and remaining \$3 put in the bank. Between the creation of table of size F_{k+1} and F_{k+2} there must be at least $F_{k+1} - F_k = F_{k-1}$ elements being inserted. So the bank balance at the time of creating table of size F_{k+2} must be

$$3F_{k-1} > F_{k-3} + 2F_{k-2} = F_{k-1} + F_{k-2} = F_k.$$

So we can move all the F_k elements to the new table using the bank balance (free of cost). Hence the amortized cost of insertion is $O(1)$.

For the deletion consider the following charging scheme: Charge \$3 for each deletion, use \$1 for deletion and remaining \$2 put in the bank. Between the creation of table of size F_{k-1} and F_{k-2} there must be at least $F_{k-3} - F_{k-4} = F_{k-5}$ elements being deleted. So the bank balance at the time of creating table of size F_{k-2} must be $2F_{k-5} > F_{k-6} + F_{k-5} = F_{k-4}$. So we can move all the F_{k-4} elements to the new table using the bank balance (free of cost). Hence the amortized cost of deletion is $O(1)$.

Note, at any point the bank balance is some constant times the number of elements present in the table, and hence is non-negative.

7. **Reductions** (AY 23/24 Sem 2). We have seen the (undirected) Hamiltonian cycle problem in tutorial. In a similar way, we can define the Directed Hamiltonian Cycle problem as follows: Given a directed graph whether there exists a directed cycle that visits each vertex exactly once. There is a slightly intricate poly-time reduction from the 3-SAT problem to the directed Hamiltonian cycle problem (which we will not see in this course, but you may assume that the directed Hamiltonian cycle is NP-complete).
Now we ask you to give a poly-time reduction from the directed Hamiltonian Cycle problem to the (undirected) Hamiltonian Cycle problem. (Note, we have already seen in lecture that the Hamiltonian Cycle problem is is in NP. So we get that the (undirected) Hamiltonian Cycle problem is NP-complete.) [**Hint**: Represent each node in the directed version using 3 nodes in the undirected version.]

Solution. Quick intuition: We need a way to inform the undirected Hamiltonian cycle problem of the direction that is present in the directed version. This will be done by representing each node by 3 nodes. Let's dive right into it.

To show DHC $\leq_p$ UHC, let's start with the polytime transformation.

Given a graph $G = (V, E)$ for the DHC problem, let's create a $G' = (V', E')$ where G' is an undirected graph and G has a Hamiltonian cycle iff G' has a Hamiltonian cycle. For every vertex $v \in G$, it will be represented in G' as 3 vertices, v', v'', v''', with undirected edges (v', v'') and (v'', v'''). For every directed edge (u, v) in G, it will be represented by the undirected edge (u''', v') in G'. This procedure is clearly a polynomial transformation. Nice! 1/3 done.

Let's move on to the next part, if there is a Hamiltonian cycle in G there is a Hamiltonian cycle also exists in G'. Suppose G has the HC: $v_1, v_2, ..., v_n$, we can quickly see that the a Hamiltonian cycle also exists in G', $v_1', v_1'', v_1''', v_2', v_2'', v_2''', ..., v_n', v_n'', v_n'''$. Yay 2/3 of the way there.

For the final part we need to show that if there is a Hamiltonian cycle in G', there is a Hamiltonian cycle in G. Suppose that G' has a HC. Since each of the vertices v'' has degree 2, they must be preceded by v' and succeeded by v''' in this cycle (or the other way around). Therefore, if the vertices v_i' appear in the cycle in the order $v_1', a_2', ..., v_n'$, then it the cycle must look like $v_1', v_1'', v_1''', v_2', v_2'', v_2''', ..., v_n', v_n'', v_n'''$ in G' and thus $v_1, v_2, ..., v_n$ in G. Okay some of you guys might be thinking, what if there exists a v_k' in the cycle that is traversed in the order $v_i''', ..., v_k', v_j'''$? Wouldn't this break our invariant on direction? Good question! Let's say our cycle is $v_1', v_1'', v_1''', v_2', v_2'', v_2''', ..., v_i''', ..., v_k', ..., v_j''', ..., v_n'', v_n'''$. This means that vertices that represent v_{i-1} in G' must also be visited in the order v_{i-1}''', v_{i-1}' because there can only be edges between u''' and v' in G'. So if we extend this, we'll get the same cycle but in the opposite order! NICE! I hope you're convinced now.

8. **NP-Completeness** (AY 23/24 Sem 2). Consider the Max-Clique problem: Given an undirected graph $G = (V, E)$ and an integer k decide whether there exists a clique of size at least k in G (i.e., as a subgraph of G). Show that Max-Clique problem is NP-complete. [**Hint**: Try a reduction from the Maximum Independent Set problem.]

   Sketch. Given a graph G and a positive integer k, we should be able to verify the certificate in polynomial time. The certificate is a subset V' of the vertices, which comprises the vertices belonging to the clique. We can validate this solution by checking that each pair of vertices belonging to the solution are adjacent, by simply verifying that they share an edge with each other. This can be done in polynomial time. Hence, Max-Clique is in NP.

   We next show that Maximum Independent Set $\leq_p$ Max-Clique. Given a graph G, we compute its complement graph $\overline{G}$. $\overline{G}$ has a clique of size $\geq k$ if and only if G has a maximum independent set of size $\geq k$. (You need to prove this in detail.) Therefore, Max-Clique problem is NP-hard.

9. **NP-Completeness** (AY 23/24 Sem 2). Consider the MAX-2-SAT problem: Given a 2-CNF formula ϕ with m clauses and an integer $1 \leq k \leq m$, decide whether there is an assignment to the variables that satisfies at least k clauses of ϕ. Show that MAX-2-SAT is NP-complete. [**Hint**: Try a reduction from 3-SAT.]

   Solution. First we note that if a truth setting of the variable makes at least k clauses true, then by substituting these truth values we can efficiently check the truth or falsity clause by clause, keeping a counter to hold the number of clauses found to be true so far. Hence MAX-2-SAT is in NP.

   The following is a reduction from 3SAT to MAX-2-SAT. That is, given an instance of 3SAT we construct an instance of MAX-2-SAT so that a satisfying truth assignment of 3SAT can be extended to a satisfying truth assignment of MAX-2-SAT. Note that a satisfying truth assignment is one that makes at least k clauses true in the MAX-2-SAT instance.

   Let S be the instance of 3SAT where the clauses are $C_1, ..., C_m$ where $C_i = \{x_i, y_i, z_i\}$, where each x_i, y_i, z_i represents either a variable or its negation and $1 \leq i \leq m$. From S we build an instance S' of MAX-2-SAT as follows. Each clause C_i in S corresponds to a clause group C_i' in S' where: $C_i' = \{(w_i), (x_i), (y_i), (z_i), (\overline{x_i} \vee \overline{y_i}), (\overline{y_i} \vee \overline{z_i}), (\overline{x_i} \vee \overline{z_i}), (\overline{w_i} \vee x_i), (\overline{w_i} \vee y_i), (\overline{w_i} \vee z_i)\}$ where w_i is a new variable and $1 \leq i \leq m$. We set $k = 7m$.

   Each clause in S' as constructed above has at most two literals. It can be seen that the clauses in S' can be efficiently generated from the clauses in S in polynomial time. We now argue that a satisfying truth assignment of S exists if and only if it can be extended to a satisfying truth assignment for S' appropriately.

   Assume that S is satisfiable. Then in a typical clause $C_i = \{x_i, y_i, z_i\}$ either one or two or all three variables are true.

   - Let $x_i = T, y_i = F, z_i = F$. With this truth assignment in S', if $w_i = T$, six clauses of C_i' become true; if $w_i = F$, seven clauses of C_i' become true.
   - Let $x_i = T, y_i = T, z_i = F$. This truth assignment in S' together with $w_i = T$ or $w_i = F$, makes seven clauses of C_i' true.

- Let $x_i = T, y_i = T, z_i = T$. With this truth assignment in S', if $w_i = T$, seven clauses of C'_i become true; if $w_i = F$, six clauses of C'_i become true.

In summary, a satisfying truth assignment of S can be extended to a satisfying truth assignment of S' where exactly seven clauses in each clause group get satisfied. Moreover no setting of w_i causes more than seven of the ten clauses to be true in each clause group in S'.

Now assume that S is not satisfiable. Then in at least one clause $C_i = \{x_i, y_i, z_i\}$ in S, we have $x_i = F, y_i = F, z_i = F$. With this truth assignment in S', if $w_i = T$, four clauses of C'_i become true. That is, if S is not satisfiable, no truth setting can make at least seven clauses true in each clause group C'_i. Since 3SAT is NP-complete, we conclude that MAX-2-SAT is also NP-complete.