Algorithms Exam
TIN093/DIT093/DIT602

Course: Algorithms

Course code: TIN 093 (CTH), DIT 093 and DIT 602 (GU)

Date, time: 15th March 2023, 8:30–12:30

Place: Johanneberg

Responsible teacher: Peter Damaschke, Tel. 5405, email `ptr@chalmers.se`

Examiner: Peter Damaschke

Exam aids: dictionary,
printouts of the Lecture Notes (which may contain own annotations),
one additional handwritten A4 paper (both sides).

Time for questions: around 9:30 and around 11:00.

Solutions: will be published after the exam.

Results: will appear in ladok.

Point limits: 28 for 3, 38 for 4, 48 for 5; PhD students: 38. Maximum: 60.

Inspection of grading (exam review): to be announced.

Instructions and Advice:

- First read through all problems, such that you know what is unclear to you and what to ask the responsible teacher.
- Write solutions in English.
- Start every new problem on a new sheet of paper.
- Write your exam number on every sheet.
- Write legible. Unreadable solutions will not get points.
- Answer precisely and to the point, without digressions.
  Unnecessary additional writing does not only cost time.
  It may also obscure the actual solutions.
- But motivate all claims and answers.
- Strictly avoid code for describing a complex algorithm.
  Instead *explain* in your words how the algorithm works.
- If you cannot manage a problem completely, still provide your approach or partial solution to earn some points.
- Facts from the course material can be assumed to be known.
  You don't have to repeat their proofs.

Remark: The number of points is not always "proportional" to the length or difficulty of a solution, but it may also be influenced by the importance of the topics and skills.

Good luck!

Problem 1 (15 points)

Imagine that you have bought a big stock of food. Every item has a best-before date, and you can consume only a fixed number of items per day. Your problem is now to plan the consumption such that every item will be consumed before its expiration date and hence nothing is wasted.

We are aware that this setting is not exactly realistic, as it ignores many real-world factors (which items go well together, spontaneous appetite, non-strict expiration dates, extended tenability after cooking, etc.). Anyway, let us model the basic version as a formal problem:

Time is divided into discrete slots $0, 1, 2, 3, \ldots$. We are given a constant positive integer c, and n items indexed by $1, \ldots, n$, each with a positive integer t_i which denotes the last slot in which item i may be used. Assign the items to the time slots, such that every time slot gets at most c items assigned, and every item i is assigned before its expiration time t_i (where "before" includes the slot t_i).

1.1. Propose a greedy algorithm for this problem, and prove by an exchange argument that it is correct. That is, you must prove that your algorithm always finds some solution if there exists some. (12 points)

1.2. State and explain a time bound for your algorithm, as a function of n, in O-notation. Clarify what you count as elementary operations. (3 points)

Remark: The problem is not formulated as an optimization problem. But nothing prevents us from using comparisons and minimization in 1.1, if it helps.

Problem 2 (9 points)

The solution to an instance of an optimization problem is not always uniquely determined, rather, there may exist many optimal solutions.

Now think of any problem that can be solved by dynamic programming. Suppose that, in addition to computing some optimal solution, we also want to know whether this optimal solution is uniquely determined. For clarity: If there exist several optimal solutions, it is not required to output them all. Still we want only *one* of them, and the additional information whether there *exist* other optimal solutions or not.

How would you modify an already existing dynamic programming algorithm, in order to output this extra information? How much additional time do you need? In particular, does this raise the O-complexity of the algorithm?

You may discuss this question in generality – the principle is the same in every dynamic programming algorithm. But if you feel more comfortable with a specific problem, you may explain your answers for one problem of your choice from the course. (This will not count negatively.)

Problem 3 (12 points)

A colleague of yours gets the idea to solve the Knapsack problem by divide-and-conquer and argues as follows: "For simplicity assume that the number n of items is a power of 2, otherwise create some additional dummy items of zero value and weight, which do not affect the solution. Divide the set of items in two subsets of $n/2$ items and solve these two instances recursively. The overall optimal solution is simply the union of these two optimal solutions. The time should satisfy the recurrence $T(n) = 2T(n/2) + p(n)$, with some polynomial p that needs to be figured out, and then $T(n)$ should be polynomial as well, according to the master theorem."

If you think that this works, complete the details that your colleague has left out and derive a time bound.

If you think that this cannot work, explain as specifically as possible what your colleague has missed in the argumentation above.

Of course, only one of these two options can be correct.

Problem 4 (12 points)

We call a subset $X \subset V$ of nodes in an undirected graph $G = (V, E)$ a *homogeneous set* if X forms either a clique or an independent set. Consider the following problem:

Given an undirected graph G and an integer $k > 0$, decide whether G contains a homogeneous set of k nodes.

Show that this problem is NP-complete. Since a right idea is crucial here, we propose already some reduction below in 4.2:

4.1. Argue that our problem belongs to NP. (3 points)

4.2. Let G be an arbitrary undirected graph with n nodes, and let H be the graph obtained from G by adding n isolated nodes (i.e., fresh nodes that are not incident to any edges). Prove the following: G has an independent set with k nodes if and only if H has a homogeneous set with $n + k$ nodes.
(6 points)

4.3. Finally conclude from 4.1 and 4.2 that our problem is NP-complete. Do not forget to consider the time of the reduction. (3 points)

Problem 5 (12 points)

Given an undirected graph with an even number of nodes, we wish to paint its nodes red and blue such that: every node gets exactly one of the colors, adjacent nodes get different colors, and exactly half of the nodes become red and blue, respectively. Describe an algorithm that produces such a coloring (or reports that no such coloring exists for the given graph), in polynomial time. Note that the graph is not assumed to be connected. You need not develop an entire algorithm from scratch, rather, you may combine known suitable algorithms to a new one.

Solutions (attached after the exam)

1.1. Sort the items by increasing t_k, and then simply assign the first c items to the first slot, the next c items to the next slot, and so on. In words: Use the items as early as possible, prioritized by their expiration times. For the correctness proof, consider any valid solution. Denote by s_k the time when item k is used. Assume that $s_j < s_i$ for some $i < j$. Then there also exist two items with this property which are neighbors in the "schedule". We can exchange items i and j. Since i is now used earlier, we cannot exceed its expiration time t_i. Moreover, since $s_j < s_i \leq t_i \leq t_j$ before the exchange, item j (which is now assigned the old s_i) is still used before t_j. Hence the solution remains valid, and we have strictly decreased the number of inversions in the sequence of the times s_k. By repeating this exchange operation as long as possible, we finally arrive at the greedy solution.
(12 points)

1.2. Considering arithmetic operations and comparisons of integers as elementary, we can state: Sorting needs $O(n \log n)$ time, and assigning the times to the slots obviously needs only $O(n)$ time, which is dominated by $O(n \log n)$. (3 points)

2. One may compute rhe optimal values of sub-instances and memoize them in an array as usual. It suffices to modify the backtracing procedure: When an optimal value is achieved by several cases, we know immediately that the overall optimal solution is not unique, since every backtracing "path" produces a solution. If the optimal case is always unique during the entire backtracing procedure, we know that the solution is also unique. The time in O-notation does not increase, as we only have to check uniqueness of the optimal case in every step. This only adds constant time to every calculation that the algorithm executes anyway. (9 points)

3. Knapsack is NP-complete, which most likely rules out any polynomial-time algorithm, so we should suspect that something is wrong. Now specifically: Let I be the set of items, let I_1 and I_2 denote the two instances after dividing (hence $I = I_1 \cup I_2$), and let $S \subset I$ denote an optimal solution. The colleague forgot that an instance of Knapsack also needs to specify a capacity. In order to split the given instance, we must also decide on a partitioning of the capacity W into two capacitites: $W = W_1 + W_2$. But we do not know in advance the total weights of $S \cap I_1$ and $S \cap I_2$. We may have to

try many partitionings of W and finally take the best solution. Therefore the number of recursive calls is not just 2, but it may even depend on W. (12 points)

4.1. When a subset C of nodes is provided, we can verify, in polynomial time, that C has k nodes, and that either no or all possible edges between these nodes exist. (3 points)

4.2. If G has an independent set with k nodes, then they form, together with the n new isolated nodes, an independent set with $n+k$ nodes in H. Conversely, assume that H has a homogeneous set C with $n+k$ nodes. The case of a clique is impossible. Thus C is an independent set. At most n nodes of C are new isolated nodes, hence at least k nodes of C appear in G and form an independent set there. (6 points)

4.3. Membership in NP was shown in 4.1, Independent Set is known to be NP-complete, and the reduction in 4.2 needs polynomial time, since only n isolated nodes are created and the sum $k+n$ is computed. Hence all conditions for NP-completeness are fulfilled. (3 points)

5. First compute the connected components and a 2-coloring of each one. If some is not 2-colorable, no solution exists, and we can stop. Otherwise, the 2-coloring of every connected component is uniquely determined, subject to swapping the two colors. Now we may either reduce the remaining recoloring problem to Subset Sum, or solve it directly by dynamic programming. Here we describe the latter option. Sort the connected components arbirarily, and let x_i and y_i be the number of red and blue nodes, respectively, in the i-th component. Define a function P by $P(k,r) = 1$ if the first k components can be recolored such that they have together exactly r red nodes, and $P(k,r) = 0$ else. Then we have $P(k,r) = P(k-1, r-x_k) \vee P(k-1, r-y_k)$. Let c be the number of connected components and n the total number of nodes. If $P(c, n/2) = 1$, we reconstruct a coloring by backtracing, and no solution exists if $P(c, n/2) = 0$. All subroutines run in $O(n^2)$ time. The dynamic programming part needs only $O(cn)$ time, where $c \leq n$. (12 points)