# Algorithms Exam TIN093/DIT602

**Course:** Algorithms

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

**Date, time:** 27th October 2018, 14:00–18:00

**Building:** L

**Responsible teacher:** Peter Damaschke, Tel. 5405

**Examiner:** Peter Damaschke

**Exam aids:** one A4 paper (both sides), dictionary,
printouts of the Lecture Notes and assignments
(possibly with own annotations).

**Time for questions:** around 15:00 and around 16:30.

**Solutions:** will appear on the course homepage.

**Results:** will appear in ladok.

**Point limits:** CTH: 28 for 3, 38 for 4, 48 for 5; GU: 28 for G, 48 for VG;
PhD students: 38. Maximum: 60.

**Inspection of grading (exam review):**
Time will be announced on the course homepage.

**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 may obscure the actual solutions.
- Motivate all claims and answers.
- Strictly avoid code for describing a complex algorithm.
  Instead *explain* how the algorithm works.
- If you cannot manage a problem completely, still provide your approach or partial solution to earn some points.
- Facts that are known from the course material can be used.
  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 (5 points)

We are given an undirected graph $G = (V, E)$, where every node has a positive weight, and we are given a integer $k$. The problem is to find a connected subgraph $H$ of $G$ with exactly $k$ nodes and with maximum total weight of the nodes in $H$. (A subgraph consists of any subset of nodes of $G$ and of the edges between them. Do not confuse this with the notion of a connected component.)

An obvious greedy algorithm is: Initially, let $H$ be empty. First put some node with largest weight in $H$. In every further step, add to $H$ some node that is adjacent to some of the nodes that are already in $H$ (such that $H$ always remains connected). Repeat this step until $H$ has $k$ nodes.

Does this algorithm guarantee the optimal solution (and if so: why?), or can you present a counterexample?

## Problem 2 (10 points)

Suppose that $B$ boxes are arranged in a line, and these boxes are indexed accordingly with positive integers $1, 2, 3, \ldots, B$. We have $n$ objects in the boxes with indices $x_1 < \ldots < x_n$. For some reason we are requested to store the objects, without changing their order, in such a way that no two consecutive boxes are used. (In other words, the difference between the indices of any two boxes containing objects has to be at least 2.) Moreover, objects can be moved in one direction only, say, only to boxes with larger indices. We want to move the objects as little as possible. To be precise, let us denote the indices of the boxes in the solution by $y_1 < \ldots < y_n$. The problem is to minimize the cost $\sum_{i=1}^{n} |y_i - x_i|$ under the constraints $y_{i+1} - y_i \geq 2$ and $y_i \geq x_i$ for all $i$. You may assume that $B$ is large enough, such that there exists a solution.

For example, if $n = 6$ objects are in the boxes $11, 12, 14, 16, 17, 22$, we may move them to the boxes $11, 13, 15, 17, 19, 22$, and the cost of this solution would be $0 + 1 + 1 + 1 + 2 + 0 = 5$.

There is a pretty obvious greedy algorithm that may solve this problem in general. We describe it in code, because it has only a few lines:

for $i := 2$ to $n$ do
begin
if $x_i < x_{i-1} + 2$ then $x_i := x_{i-1} + 2$;
$y_i := x_i$;
end

2.1. How much time does this algorithm take? (Consider arithmetic operations with integers as elementary operations.) Briefly motivate your answer. (3 points)

2.2. Prove that, in fact, this greedy algorithm always yields a solution with minimum cost. – Hint: Compare the greedy solution to a (hypothetical) better solution and apply a suitable exchange argument or derive a contradiction. It may be a good idea to look at the leftmost box where the two solutions differ. (7 points)

## Problem 3 (10 points)

Suppose that we have the same situation as in Problem 2, with one important change: It is allowed to move objects in both directions. Formally: Given $n$ integers $x_1 < \ldots < x_n$, the problem is to find integers $y_1 < \ldots < y_n$ so as to minimize the cost $\sum_{i=1}^{n} |y_i - x_i|$ under the constraints $y_{i+1} - y_i \geq 2$ for all $i$.

In the above example, the 6 objects in the boxes $11, 12, 14, 16, 17, 22$ may now be moved to the boxes $10, 12, 14, 16, 18, 22$, and the cost of this solution would be only $1 + 0 + 0 + 0 + 1 + 0 = 2$.

So this additional degree of freedom can lower the costs but it also makes it more difficult to compute an optimal solution. Greedy approaches seem to fail, and the next natural step is to try dynamic programming. (We do not claim that this yields already the fastest possible algorithm, but at least it works.)

Specifically, we define the following function: Let $OPT(j, b)$ be the smallest possible cost of a solution $y_1 < \ldots < y_j$ for the first $j$ objects that uses only the first $b$ boxes (that is, $y_j \leq b$). We may formally define $OPT(j, b) := \infty$ if no solution exists at all.

3.1. Derive a formula that allows to compute all values $OPT(j, b)$. Explain why your formula is correct. – Hint: Certainly your formula must use the given positions $x_j$ somehow, otherwise it cannot work. (6 points)

3.2. Remember that $B$ denotes the total number of boxes. Give (and motivate) a time bound of the dynamic programming algorithm that is based on your formula from 3.1. It should be polynomial in $n$ and $B$. But you are not expected to describe the whole algorithm, with initializations, backtracing, and so on. (4 points)

## Problem 4 (10 points)

The currency of some country exists as coins of values $c_1, \ldots, c_n$, where the $c_i$ are integers (positive, of course!) with $c_1 < \ldots < c_n$. That is, we assune that the values are already sorted in increasing order.

We want to pay a certain amount $m$ of money, and figure out whether the exact amount $m$ can be paid with $k = 4$ coins. Note that some of these coins may have equal values.

The problem with $k = 2$ coins can be solved in $O(n)$ time. *You can use this result here without proof.* (It was the subject of an assignment, but it was presented there as a different story: renting rooms in a warehouse.)

The problem with $k = 4$ coins can be naively solved in $O(n^4)$ time: Just compute all $O(n^4)$ possible sums of 4 values and check whether some of them equals $m$.

4.1. Give an algorithm for $k = 4$ that needs only $O(n^2 \log n)$ time. – Hint: Divide the problem in two simpler problems and combine their solutions in a suitable way. (However this is not divide-and-conquer, as no recursion will be needed.) (8 points)

4.2. Why can't we simply use the dynamic programming algorithm for Subset Sum, in order to achieve a time bound as in 4.1? (2 points)

## Problem 5 (7 points)

A construction company gets the job to erect a number of buildings along a street of length $s$. There exist only $n$ prefabricated buildings, each with a given length $l_i$ and a given height $h_i$, for $i = 1, \ldots, n$. Their widths (in the direction orthogonal to the street) are all the same, hence the space (living space, office space, or whatever) in the buildings is proportional to both $l_i$ and $h_i$. The company can choose among these $n$ available prototypes, and each prototype can be used at most once. The sum of lengths of the buildings must not exceed the length of the street. The goal is to maximize the total space in the buildings.

Is the problem of an optimal choice of buildings solvable in polynomial time (in $n$) or is it NP-complete? Give either a polynomial-time algorithm or a reduction from a known NP-complete problem. Of course, only one of these options can be correct. In any case: Describe your algorithm or reduction precisely and completely, not only in vague terms, and explain its correctness.

## Problem 6 (8 points)

In one of the assignments we defined the Half-Half Subset Sum problem: We are given $n$ integers $w_1, \ldots, w_n$, and the problem asks whether some subset of them has the sum $\sum_{i=1}^{n} w_i/2$. This problem is NP-complete. *You can use this result here without proof.*

Now suppose that we have $n$ objects of sizes $w_1, \ldots, w_n$, where $n$ is an even number, and the $w_i$ are non-negative integers. We want to divide these objects among two persons, such that each person gets exactly $n/2$ objects of total size $\sum_{i=1}^{n} w_i/2$. (Note that these are two conditions: Give both persons the same number of objects AND the same total size.)

Prove that this problem is also NP-complete, by a reduction from Half-Half Subset Sum. – Hint to get started: Create a suitable number of additional "dummy" objects of size 0. Do not forget that a reduction requires an equivalence proof, besides the construction of an instance.

## Problem 7 (10 points)

Let $G = (V, E)$ be a directed acyclic graph (DAG) with $n$ nodes and $m$ edges, where every edge has some given positive weight. Furthermore, let $s, t \in V$ be two nodes. We wish to find a directed path from $s$ to $t$, such that the minimum weight of the edges on this path is maximized. (This objective is quite different from both the shortest and the longest path problem.)

Give an algorithm that solves this problem in $O(n+m)$ time. And as usual, motivate all claims.

To avoid a possible trap: The problem cannot be solved via spanning trees; this method works only for undirected graphs and it also takes more than linear time.

## Solutions (attached after the exam)

1. One counterexample is a path of 4 nodes, with weights $4, 1, 3, 3$, and $k = 2$. The greedy algorithm would start with 4, and then it is forced to add 1. But the optimal solution would be $3 + 3$. (5 points)

2.1. It takes $O(n)$ time, because it consists of only one loop, which in every step performs a constant number of additions and comparisons of numbers. (3 points)

2.2. The intuitive reason is that the greedy algoruthm never moves an object more than necessary to the right. One possible formal way to write down the proof is proposed here: Let $y_1 < \ldots < y_n$ be the greedy solution, and assume there is a better solution $z_1 < \ldots < z_n$. Since $\sum_{i=1}^{n} |z_i - x_i| < \sum_{i=1}^{n} |y_i - x_i|$, there must exist an index $j$ where $z_j < y_j$. Let $j$ be the smallest such index. This means $y_i \leq z_i$ for all $i < j$. We have $z_{j-1} + 2 \leq z_j$, hence $y_{j-1} + 2 \leq z_{j-1} + 2 \leq z_j \leq y_j - 1$. This further implies $y_{j-1} + 3 \leq y_j$. But the greedy algorithm chooses $y_j > y_{j-1} + 2$ only if the $j$-th object is not moved, that is, if $y_j = x_j$. But since $z_j < y_j$, we conclude $z_j < x_j$, thus, the assumed better solution is not valid. (7 points)

3.1. Let $y$ denote the new position of object $j$. Moving the object from box $x_j$ to box $y$ adds $|y - x_j|$ to the cost. Moreover, we can combine this move with the best solution for the first $j - 1$ objects that uses at most $y - 2$ boxes. This shows: $OPT(j, b) = \min_{y \leq b}(|y - x_j| + OPT(j - 1, y - 2))$. (6 points)

3.2. We must compute $OPT(n, B)$. To this end we must compute $nB$ values $OPT(j, b)$. Every such calculation performs $O(B)$ arithmetic operations with integers, and table look-ups of earlier values. Hence the time bound is simply $O(nB^2)$. (4 points)

4.1. First compute all possible sums of 2 values. We call them pairwise sums. This can be done naively in $O(n^2)$ time. Then sort all pariwise sums, which takes $O(n^2 \log(n^2)) = O(n^2 \log n)$ time. Clearly, every sum of 4 values is the sum of two pairwise sums. Hence we can now apply the already existing $O(n)$-time algorithm, but instead of $n$ numbers we have to deal with $O(n^2)$ numbers (the pairwise sums computed before). Therefore this part takes $O(n^2)$ time. The total time complexity is $O(n^2 \log n + n^2) = O(n^2 \log n)$. (8 points)

4.2. Possible answers: Its time bound depends on the values to be added, not only on the number $n$ of items, and it is not a polynomial in (only) $n$. We may use it to compute all sums of $k$ values, but the time would not be better than the naive $O(n^k)$. (2 points)

5. The problem is NP-complete, which can be shown by a reduction from Knapsack. (Actually, the reduction is merely a translation of one problem into another.) Consider any instance of Knapsack, with $n$ items of weights $w_i$ and values $v_i$, and with capacity $W$. We construct an instance of our "building problem" as follows. Let $s := W$ and $l_i := w_i$. We want the value $v_i$ to be proportional to $l_i h_i = w_i h_i$, therefore we set $h_i := v_i/w_i$. (Without loss of generality, our proportionality factor is 1.) Now it is obvious, for any threshold $t$, that some subset of items with total value at least $t$ fits in the knapsack, if and only if some subset of buildings has a total length not exceeding $s$ and provides space at least $t$. (7 points)

6. Given an instance $x$ of Half-Half Subset Sum with $n$ integers $w_1, \ldots w_n$, we construct an instance $y$ of the new problem, with the same integers and additionally $w_{n+1} = \ldots = w_{2n} = 0$. The number of objects is $2n$ (rather than $n$), hence each person must get $n$ objects. If $x$ has a solution, we first divide the "real" objects according to this solution. Then we divide the "dummy" objects such that both persons have $n$ objects. Hence $y$ has a solution. Conversely, if $y$ has a solution, then $x$ has a solution as well, because the "dummy" objects do not matter here. (8 points)

7. First we coonstruct a topological order in $O(n + m)$ time. We may assume that $s$ and $t$ is the first and last node, respectively, because other nodes cannot be on any path from $s$ to $t$. So let $v_1 = s$ and $v_n = t$, possibly with a reduced $n$. Next define $OPT(j)$ as the largest number $w$ such that there is a directed path from $v_1$ to $v_j$ where all edges have a weight at least $w$. Let $w(i, j)$ denote the weight of the edge $(v_i, v_j)$. Then $OPT(j) = \max\{\min\{OPT(i), w(i, j)\} |\ (v_i, v_j) \in E\}$. Explanation: Every path to $v_j$ has some last edge $(v_i, v_j)$. We can take an optimal path to $v_i$. If $OPT(i) > w(i, j)$, then the last edge lowers the smallest weight on the path to $w(i, j)$, otherwise the smallest weight remains the same. Finally we take the best possible $v_i$ by maximizing over all $v_i$. The time is $O(n + m)$ in total, because every edge is considered only once. Backtracing is done in the standard way. (10 points)