Algorithms Exam TIN093[*]/DIT602

Course: Algorithms

Course code: TIN 093, TIN 092 (CTH), DIT 602 (GU)

Date, time: 21st October 2017, 14:00–18:00

Building: SBM

Responsible teacher: Peter Damaschke, Tel. 5405

Examiner: Peter Damaschke

Exam aids: one A4 paper (both sides), dictionary,
printed Lecture Notes and slides (possibly with own annotations),
any edition of Kleinberg, Tardos: "Algorithm Design".

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.

[*] also forTIN092

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 an 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 (12 points)

Let $x_1, \ldots, x_n$ be positive real numbers which are however smaller than 1. We wish to find a pair i, j such that $x_i + x_j \leq 1$, but $x_i + x_j$ is as close as possible to 1. In other words, the difference $1 - (x_i + x_j)$ shall be minimized but still be non-negative.
An $O(n^2)$-time algorithm for this problem is pretty obvious: examine all pairs and take the best. But we can do faster:

First sort the numbers, that is, re-index them such that $x_1 \leq \ldots \leq x_n$. Initially let $i := 1$ and $j := n$. As long as $i < j$, repeat the following actions:
(1) If $x_i + x_j \leq 1$ then increment i, that is, $i := i + 1$.
(2) If $x_i + x_j > 1$ then decrement j, that is, $j := j - 1$.
Always maintain the largest sum $x_i + x_j \leq 1$ that has been found so far.

1.1. How much time does this algorithm need? (Also explain your answer.) You can consider operations with real numbers as elementary. (5 points)

1.2. Show that this algorithm returns, in fact, the largest sum $x_i + x_j \leq 1$. You need to argue why the optimal pair is not overlooked, although by far not all pairs i, j are considered. (7 points)

Remark/hint: It can be debated whether this is a typical greedy algorithm, however the argument needed in 1.2 is much reminescent of an exchange argument.

Problem 2 (12 points)

We had solved the String Editing problem, with two input strings named $A = a_1 \dots a_n$ and $B = b_1 \dots b_m$, by a dynamic programming algorithm. The core of this algorithm was the edit distance formula

$$OPT(i,j) = \min\{OPT(i-1,j)+1, OPT(i-1,j-1)+\delta_{ij}, OPT(i,j-1)+1\},$$

where $\delta_{ij} = 1$ was defined by: $\delta_{ij} = 1$ if $a_i \neq b_j$, and $\delta_{ij} = 0$ if $a_i = b_j$. Initial values were $OPT(i,0) = i$ and $OPT(0,j) = j$ for all i and j, respectively.

But now the famous microbiologist Prof. Dr. Chris-Per Kaas faces some variations of the String Editing problem:

2.1. A must be transformed into B by a minimum number of *insert* and *delete* operations, whereas characters cannot directly be replaced by other characters at unit cost.
Adapt the dynamic programming algorithm to this new problem, and explain why your modification correctly solves the problem. (2 points)

2.2. A must be transformed into B by a minimum number of *insert* and *replace* operations, whereas characters cannot be deleted. (Clearly, a solution can only exist if A is no longer than B.)
Again: Adapt the dynamic programming algorithm to this new problem, and explain why your modification correctly solves the problem. (4 points)

2.3. A must be transformed into B by *insert* operations, whereas characters can neither be deleted nor be replaced by other characters. Again, a solution does not always exist, but if one exists, the algorithm must find some. Here we do not need dynamic programming.
Give a greedy algorithm for this problem version, along with a time bound and correctness argument. (A fully worked-out correctness proof is not expected, but the key argument should be visible.) The time bound should be considerably better than the known one for the original String Editing problem. (6 points)

Problem 3 (8 points)

We are given b bins, which are denoted by indices $1, \ldots, b$. For each index i, the ith bin contains p_i "positive" and n_i "negative" objects. Furthermore, a number f is given.
We wish to split the set of bins $\{1, \ldots, b\}$ into two sets P and N such that $\sum_{i \in P} n_i \leq f$ and $\sum_{i \in N} p_i$ is minimized.

How can you compute such sets $P, N \subseteq \{1, \ldots, b\}$, and how much time does this take? Express your time bound as a function of b and f.
Hint: You should avoid developing a new algorithm from scratch. Instead, you can reformulate the problem to obtain a similar one that we have already solved.

Motivation of Problem 3
(You can skip this paragraph during the exam; it is not needed for solving the exercise, but maybe you are curious afterwards.)
The problem can appear in classification tasks in machine learning. Objects in a sample are grouped into abstract "bins", according to their properties, and every object is also labeled as either positive or negative. Assuming that the sample is representative, we want to predict the label of new, previously unseen objects, only based on their properties. The cases predicted as positive/negative are collected in P/N. A given rate of false positive predictions is allowed (the negative objects in P), and under this restriction we wish to minimize the rate of false negative predictions (the positive objects in N).

Problem 4 (4 points)

The time analysis of the known divide-and-conquer algorithm for Counting Inversions had led to the reucrrence $T(n) = 2T(n/2) + O(n)$ with solution $T(n) = O(n \log n)$. There we had counted operations with arbitrary integers as elementary operations. If we are, instead, interested in the number of operations with digits, we can see that a merging phase has to add $O(n)$ numbers of size $O(n)$, hence with $O(n \log n)$ digits, and therefore merging needs $O(n \log n)$ digit operations. Now we get the recurrence

$$T(n) = 2T(n/2) + O(n \log n).$$

Unfortunately it is not covered by the "master theorem" for recurrences, yet we can follow the same solution method. We do the first step: Repeated substitution yields $T(n)$ in explicit form:

$$T(n) = \sum_{i=0}^{\log_2 n} 2^i (n/2^i) \log(n/2^i).$$

(We may assume that n is a power of 2, and that the constant in the $O(n \log n)$ term is 1, since all this does not affect the O-result.)

Show that this sum is bounded by $O(n(\log n)^2)$. The calculation is short and straightforward, but please be precise.

Problem 5 (10 points)

Reminder: A clique in a graph is a subset of nodes such that all pairs of nodes in this subset are joined by edges. ("All possible edges exist.")
We define the problem Double Clique as follows. We are given an undirected graph $G = (V, E)$ and an integer k. The problem is to decide whether G contains two disjoint cliques, each with k nodes.
More formally, the problem asks, for the given graph G: Do there exist two node sets $V_1 \subset V$ and $V_2 \subset V$ such that $|V_1| = |V_2| = k$, $V_1 \cap V_2 = \emptyset$, and each of V_1 and V_2 is the node set of a clique in G?

Prove that Double Clique is NP-complete. In more detail:

5.1. Why is Double Clique in NP? (2 points)

5.2. Describe a reduction from a suitable "start problem" to Double Clique. Make sure that it needs only polynomial time. (4 points)

5.3. Prove that your reduction is correct, that is: Show that any given instance of your start problem is equivalent to the instance of Double Clique that your reduction constructs. (4 points)

Remark: Do not think complicated in 5.2 – an idea of a possible reduction should be rather obvious.

Problem 6 (14 points)

The following problem is of great interest for scheduling of unit-time jobs under precedence constraints.
A k-coloring of a directed graph $G = (V, E)$ is a function c that assigns to every node v an integer $c(v) \in \{1, \ldots, k\}$ (a "color") such that every directed edge $(u, v) \in E$ satisfies $c(u) < c(v)$. The problem is to construct a k-coloring with minimum k. Obviously, a coloring can exist only if the graph is a DAG.

For k-coloring of undirected graphs, the condition was only $c(u) \neq c(v)$ on every undirected edge. While the usual, undirected coloring problem is known to be NP-complete, amazingly the directed coloring problem on DAGs can be solved in polynomial time, as you are supposed to show now. We split the task into three small exercises that you can solve independently one-by-one.

Given a DAG $G = (V, E)$, we first create an additional node s and all possible directed edges (s, v), $v \in V$. We define $c(s) := 0$. For any two nodes, let $L(u, v)$ denote the length of a longest (not shortest!) directed path from u to v. (The length of a path is simply the number of edges.)

6.1. Prove that the function defined by $c(v) := L(s, v)$ for all v is a coloring of $G = (V, E)$. That is, you must show $c(u) < c(v)$ for every directed edge $(u, v) \in E$. (5 points)

6.2. We claim that every possible coloring c' of G must have the property $c'(v) \geq L(s, v)$ for all nodes v. Explain why. (4 points)

6.3. Finally, based on 5.1 and 5.2, propose an algorithm that solves the k-coloring problem on DAGs. Explain why it is correct, and give a time bound. (5 points)

Solutions (attached after the exam)

1.1. Sorting costs $O(n \log n)$ time. Then, every iteration either increases i or decreases j, which can happen at most n times. Every iteration costs $O(1)$ time, since a constant number of additions and comparisons are done. Thus, the loop costs $O(n)$ time. In total we need $O(n \log n)$ time. (5 points)

1.2. Consider the first step where $i = 1$ and $j = n$.
(1) If $x_1 + x_n \leq 1$ then $x_1 + x_n$ is a valid solution. But there is no need to consider other pairs with x_1. If $x_1 + x_j \leq 1$ for any j, then $x_j \leq x_n$ implies $x_1 + x_j \leq x_1 + x_n \leq 1$. That is, we can exchange x_j by x_n and get a better solution. Therefore it is safe to ignore x_1 henceforth. We can set $i := 2$ and consider the instance $x_2, \ldots, x_n$ only.
(2) If $x_1 + x_n > 1$ then it is safe to ignore x_n henceforth: Since $x_1 \leq x_j$ for all j, we have $x_j + x_n > 1 \geq x_1 + x_n > 1$. Hence all other pairs with x_n are invalid solutions, too. Therefore we can set $j := n - 1$ and consider the instance $x_1, \ldots, x_{n-1}$ only.
The same arguments as above apply to the instance $x_i, \ldots, x_j$ for general i and j. (7 points)

2.1. We keep the formula for $OPT(i, j)$ but we make replace operations too expensive to be applied, simply by defining $\delta_{ij} = \infty$ if $a_i \neq b_j$. Alternatively we can define $\delta_{ij} = 2$ if $a_i \neq b_j$, since a replace operation is equivalent to one insert and one delete operation. (2 points)

2.2. The term $OPT(i-1, j) + 1$ accounts for the case that we transform $a_1 \ldots a_{i-1}$ into $b_1 \ldots b_j$ and delete a_i at unit cost. By omitting this term we disable this option. Hence

$$OPT(i, j) = \min\{OPT(i-1, j-1) + \delta_{ij}, OPT(i, j-1) + 1\}$$

is a suitable dynamic programming formula for this problem version. We must also change the initial values, since no solution exists if A is longer than B. For instance, we may set $OPT(i, 0) := \infty$ for all $i > 0$. (4 points)

2.3. Scan the string A from left to right and assign every symbol in A to the earliest possible symbol in B. The time is $O(n + m)$, because we need to traverse both A and B only once.
The correctness argument is easier to write down by using a more formal description of this greedy algorithm: We construct a function g inductively.

Let $g(0) := 0$, and let $g(i+1)$ be the smallest $j > g(i)$ such that $a_i = b_j$. We align every a_i to $b_{g(i)}$. A solution exists if and only if $g(n)$ still exists. To show correctness, we consider any different solution to the instance, described by another function h. Let i be the smallest index such that $h(i+1) \neq g(i+1)$. Since $g(i+1)$ is the smallest $j > g(i) = h(i)$ such that $a_i = b_j$, we have $h(i+1) > j = g(i+1)$. Thus we can re-define $h(i+1) := j$ and get another valid solution where also $h(i+1) = g(i+1)$. By this exchange argument we can, step by step, turn h into g. (6 points)

3. Minimizing $\sum_{i\in N} p_i$ is equivalent to maximizing $\sum_{i\in P} p_i$, because $\sum_i p_i$ is fixed by the given instance, and $\sum_{i\in N} p_i + \sum_{i\in P} p_i = \sum_i p_i$. Now it should be evident that the given problem is just the Knapsack problem with the following input parameters: capacity f, sizes n_i, and values p_i. (Consider P as the knpasack to be filled.) As we know, Knapsack can be solved in $O(bf)$ time by dynamic programming. (8 points)

4. The only idea needed here is to simplify the logarithmic term.
$T(n) = \sum_{i=0}^{\log_2 n} 2^i(n/2^i)\log(n/2^i) < \sum_{i=0}^{\log_2 n} 2^i(n/2^i)\log n = \sum_{i=0}^{\log_2 n} n\log n = O(n(\log n)^2)$. (4 points)

5.1. We can verify in polynomial time that two given subsets of nodes are cliques, that they are disjoint, and that they have k nodes each. (2 points)

5.2. Naturally we give a reduction from Clique. Let H be any graph, and k an integer. We construct a graph G, simply by taking two copies of H on two disjoint node sets. Obviously, copying can be done in polynomial time. We keep the given k. (4 points)

5.3. If H has a clique of k nodes, then, clearly, G has two disjoint cliques of k nodes, in the two copies of H. Conversely, suppose that G has two disjoint cliques of k nodes. Consider any one of them. This clique is entirely in one copy of H, because no edges exist between the two copies. Therefore H has a clique of k nodes. (4 points)

6.1. Consider a longest path Q from s to u. By definition, its length is $c(u)$. Appending the edge (u, v) yields a path from u to v. The node v is not already on Q, since this would create a directed cycle. Hence we have found a path from s to v with $c(u) + 1$ edges. Thus $c(v) \geq c(u) + 1$. (5 points)

6.2. This follows instantly from the condition $c'(u) < c'(v)$ for every directed edge: Along a directed path from s to any node, the value of c' inreases by at least 1 on every edge, therefore the color must be at least the path length. In particular, this holds also for a longest path. (4 points)

6.3. Compute the lengths $L(s,v)$ of longest paths from s to all other nodes v. As we know, this can be done in $O(|E|)$ time. Due to 6.1 and 6.2, $L(s,v)$ equals the smallest possible color of v. Hence, if all colors (i.e., lengths) are bounded by k, we have found a k-coloring, and otherwise we know that no k-coloring can exist. (5 points)

Addendum to solutions (attached later)

An alternative way to write the proof of 1.2:
Let $x_a + x_b$ be an optimal solution. We can miss it in only two ways:
(1) We reach $i < a$ and $j = b$, but next we decrement j.
(2) We reach $i = a$ and $j > b$, but next we increment i.
Both cases lead to contradictions:
(1) Here $x_i + x_j \leq x_a + x_b \leq 1$, and in this case i is incremented instead.
(2) Here $x_a + x_b \leq x_i + x_j$. If $x_b = x_j$, we have just found another optimal solution, which is fine. If $x_b < x_j$ then, since $x_a + x_b \leq 1$ was optimal, we have $x_i + x_j > 1$, and in this case j is decremented instead.