Algorithms Exam TIN093/DIT600

Course: Algorithms

Course code: TIN 093 (CTH), DIT 600 (GU)

Date, time: 24th October 2015, 14:00–18:00

Building: M

Responsible teacher: Peter Damaschke, Tel. 5405.

Examiner: Peter Damaschke.

Exam aids: a handwritten A4 paper (both sides), dictionary, the printed Lecture Notes; 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): times will be announced on the course homepage.

Instructions and Advice:

- Numbers in parentheses are the points.
- 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.

Good luck!

Problem 1: Digit Operations (8)

For two integers with n and m digits, respectively, in the decimal system, let us assume that multiplication, as well as division with remainder, requires $O(nm)$ time, although theoretically faster algorithms are possible.
Given an integer p with n digits, we want to test whether p is a prime number. A rather simple algorithm proceeds as follows: Compute p/i for all integers i with $2 \leq i \leq \lfloor\sqrt{p}\rfloor$. If some of the p/i is integer then p is not prime, otherwise p is prime. (We remark that this is by far not the smartest algorithm for the problem.)

1.1. Why is this algorithm correct? That is, why is it enough to test factors i up to $\lfloor\sqrt{p}\rfloor$ only? (2)

1.2. How much time does this algorithm need? Express the time in two ways: once as a function of n, and once as a function of p. Give your derivations of the time bounds, not only your final answers. (3)

1.3. How much time do we save compared to the most naive algorithm that tests all factors $i = 2, \ldots, p$? Give a clear quantitative statement. (2)

1.4. Is the time of the proposed algorithm polynomial in the input length? Why, or why not? (1)

Problem 2: Both Fast and Correct (14)

The security of some technical facility shall be checked and approved by external experts. For the sake of reliability, two experts shall give their opinions independently. An expert can come and work at one of m days. A number of k experts are available for this task. Each expert i provides a price list: If expert i works on day j, then (s)he will charge us with cost c_{ij}. The problem is to select two different experts and two different days for them, such that the sum of costs is minimized.
More formally: We are given a matrix with k rows and m columns, and entries c_{ij} in the ith row and jth column. We want to find two entries with minimum sum that are neither in the same row nor in the same column. That is: minimize $c_{ij} + c_{i'j'}$ under the conditions $i \neq i'$ and $j \neq j'$.
We assume $k \leq m$, the other case is symmetric. For simplicity we also assume that all numbers c_{ij} are distinct.
An obvious greedy algorithm would fail, and naive exhaustive search that tries all valid pairs and takes the cheapest would need $O(k^2m^2)$ time. Instead we develop an $O(km)$ time algorithm, in a few small steps:

2.1. In each row, we call the two smallest entries "candidates". Prove that an optimal solution can only consist of candidates. (4)

2.2. Give a procedure that determines all candidates in all rows in $O(km)$ time altogether. (3)

2.3. Finally, we try all pairs of candidates exhaustively and take a pair that satisfies the condition ($i \neq i'$ and $j \neq j'$) and has minimum cost. Argue why this correctly solves the problem, and why the total time is $O(km)$ as claimed. (4)

2.4. We also claim that $O(km)$ is the optimal worst-case time bound for this problem. Give a clear argument why this cannot be improved. (3)

Problem 3: Scheduling a Fixed Number of Jobs (14)

Prof. Justin Time wants to plan some project that requires k short steps of work. For technological or logistic reasons, these steps can be done only at specific moments in time, and after each step, the next step can be done only after at least L time units. Furthermore, the possible times have different costs. The goal is to minimize the total costs.
More formally: We are given n real numbers $s(1) < \ldots < s(n)$ (possible times), another n real numbers $c(1), \ldots, c(n)$ (costs), an integer $k < n$, and a constant L. Choose k indices $i_1 < \ldots < i_k$ such that $s(i_{j+1}) - s(i_j) \geq L$ holds for all $j = 1, \ldots, k-1$, and the following sum is minimized:

$$\sum_{j=1}^{k} c(i_j)$$

As an **example** let $L = 3$, $k = 4$, and times and costs are given by:

i	1	2	3	4	5	6	7	8	9	10
s(i)	3	4	5	6	8	10	12	15	17	20
c(i)	1	2	1	2	3	2	3	2	1	3

Then, for example, the indices $1, 4, 6, 9$ form a valid solution, since all chosen times $3, 6, 10, 17$ have gaps of at least 3 time units in between. The total cost is $1 + 2 + 2 + 1 = 6$.
Develop a dynamic programming algorithm for this problem. We propose already an "OPT function": Define $OPT(j, m)$ to be the minimum total cost of j steps that are all done before time $s(m)$.

3.1. How can we actually compute the function $OPT(j, m)$? That is, give a recursive formula, and argue why it is correct. (4)

3.2. Analyze the time of your algorithm. (5)

3.3. Finally we consider the special case when all costs are equal, say, $c(i) = 1$ for all i. Then the only problem is: "Does the given instance allow a solution with k steps at all?" Give a greedy algorithm that solves this problem, and argue why it is correct. – A complete correctness proof is not expected, but the key argument should be there. (5)

Problem 4: Optimal Time Bound by "Divide" (8)

The following search problem can be tackled by some rudimentary form of divide-and-conquer, with only two recursion levels.
A certain industrial product will be broken if it is exposed to a mechanical shock, with a force larger than some unknown value f. If the force is at most f, then the object remains intact. We refer to these two cases as "bad" and "good". We want to figure out the exact value of f by testing a number of objects. Suppose that f is capable of integer values from 1 to n, that is: $f \in \{1, \ldots, n\}$. Since we do not want to destroy too many tested objects, we allow only two bad tests.

4.1. Give a search algorithm that determines f by using $O(\sqrt{n})$ tests, at most two of which are bad. (Maybe the idea is obvious, nevertheless a hint: Divide the range of f into roughly $\sqrt{n}$ intervals of $\sqrt{n}$ values ...) Describe the details and show that the algorithm has the claimed properties. (5)

4.2. Give an argument why any search algorithm needs at least $\sqrt{n}$ tests in the worst case, if only two bad tests are permitted. (3)

Problem 5: Traversal – With Pressure (6)

Mr. Hector Pascal is sales representative of a company that offers pressure washers for cleaning corridors in large industrial buildings. In order to demonstrate the product he has to walk through all corridors, thereby applying his pressure washer, and return to the start point. A requirement is that he must traverse every corridor at least twice, namely, at least once in each direction.
The system of corridors can be modelled as a connected undirected graph. *Note that nothing else is assumed; the graph can be arbitrary.* Thus, the task is to start in some node, traverse every edge at leat once in each direction, and return to the start node. For clarity: Every edge (u, v) must be traversed from u to v at some time, and also from v to u at some time.

5. Show that it is always possible to walk through the graph, traversing every edge *exactly* once in each direction. (Hence, in a graph with m edges, the length of the walk is only $2m$.) Propose an algorithm that computes such an optimal tour. Make sure that you explain why your algorithm fulfills the given requirement. (6)

Problem 6: Simple Reductions Between Graph Problems (10)

We formulate a graph problem named Cycle Deletion:
Given is an undirected, connected graph $G = (V, E)$ where every edge has some positive cost. The goal is to destroy all cycles cheaply, that is, remove a subset $F \subset E$ of edges with minimum total cost, such that the remaining graph $(V, E \setminus F)$ has no cycles.
A motivation is that cycles are undesirable in certain networks. But here we cannot go into possible applications. We will figure out the complexity of Cycle Deletion. First some preparation:

6.1. Instead of a minimum spanning tree in a graph we might be interested in a maximum spanning tree, that is, a spanning tree with maximum total edge cost. Show that Minimum Spanning Tree is polynomial-time reducible to Maximum Spanning Tree, and vice versa. (3)

6.2. Show that Maximum Spanning Tree is polynomial-time reducible to Cycle Deletion, and vice versa. (4)

6.3. Polynomial-time reductions can be used for two purposes: to get new polynomial-time algorithms or to prove NP-completeness. How is the situation in our case? More precisely: You know the complexity of Minimum Spanning Tree. What do 6.1 and 6.2 imply for the complexity of Cycle Deletion? Explain. (Note that you can still answer even if you could not manage 6.1 and 6.2.) (3)

Solutions (attached after the exam)

1.1. Let $p = rs$ (product of integers). If both r and s were greater than $\sqrt{p}$ then $rs > p$. Hence it suffices to detect a factor up to $\sqrt{p}$. (2 points)

1.2. Every division costs at most $O(n^2)$ time, where $n = O(\log p)$. Since we do $\sqrt{p}$ of them, the total time is $O(\sqrt{p}(\log p)^2)$. Conversely, $p = O(10^n)$. Thus the time bound also reads as $O(10^{n/2}n^2)$. (3 points)

1.3. By the same reasoning as before, the time is $O(p(\log p)^2)$, or $O(10^n n^2)$. We save a factor $O(\sqrt{p}) = O(10^{n/2})$. (2 point)

1.4. No, because the input length is the number n of digits, and our bounds are exponential in n. (1 point)

2.1. Consider any solution, with two entries c_{ij} and $c_{i'j'}$. Assume that c_{ij} is not among the "candidates", that is, two entries in row i are smaller. At least one of them is in some column $j'' \neq j'$. Since $c_{ij''} + c_{i'j'} < c_{ij} + c_{i'j'}$, we got a valid solution better than optimal, a contradiction. (4 points)

2.2. Loop through each row and maintain the two smallest numbers seen so far. This costs $O(m)$ time per row, hence $O(km)$ time in total. (3 points)

2.3. Correctness follows from 2.1, since only candidates can be in an optimal solution. We have only $2k$ candidates, thus we check $O(k^2)$ pairs. Since $k \leq m$, this bound is within $O(km)$. (4 points)

2.4. The input size is km. We must read the entire input, otherwise we can miss a small entry and thus the correct solution. (3 points)
More elaboration on the last step: Consider instances where all entries are strictly between 1 and 2, except one 0 entry at an unknown position. Then the optimal pair has cost $< 2 + 0 = 2$, and any pair without the 0 has cost $> 1 + 1 = 2$. Thus we can miss the optimum if we do not read all entries.

3.1. For doing j steps until $s(m)$ we have two options: Either do j steps until $s(m-1)$, or do the jth step exactly at time $s(m)$, which costs $c(m)$, and do $j-1$ steps until time $s(m) - L$. This yields:

$$OPT(j, m) = \min\{OPT(j, m-1), c(m) + OPT(j-1, m')\}$$

where m' is the largest index with $s(m') \leq s(m) - L$. (4 points)

3.2. The auxiliary values m' for each m can be computed in $O(n)$ time by scanning the array of the $s(m)$-values from left to right. In the recursion we have to compute $OPT(j, m)$ for all $j \leq k$ and $m \leq n$. Each recursion step needs $O(1)$ time, hence the total time is simply $O(kn)$. (5 points)

3.3. Go from $i = 1$ to n and always take the earliest possible time (respecting the gap L) for the next step of work. Correctness is seen by an exchange argument: Suppose that $s(a)$ is a possible next step, but a solution chooses some later $s(b)$ instead. Then, replacing $s(b)$ with $s(a)$ gives another valid solution, so we could have chosen $s(a)$. (5 points)

4.1. First we test only multiples of $\sqrt{n}$, always rounded to the next integer. There exist $O(\sqrt{n})$ of them. We test them strictly in ascending order. As soon as we get the first bad outcome, we know a range of size $\sqrt{n}$ that contains the unknown value. Now we test *all* values in this range, again in ascending order. As soon as we get the second bad outcome, we have found the correct value. We did $O(\sqrt{n})$ tests in total. (5 points)

4.2. Let $f_1 < f_2 < f_3 < \ldots$ be the values tested as long as all outcomes are good. This sequence can have at most $\sqrt{n}$ members (or the claim is already proved). Thus it has a jump of size at least $\sqrt{n}$: there exists i with $f_i - f_{i-1} \geq \sqrt{n}$. If f_i happens to be a bad test, we must test all (at least $\sqrt{n}$) values between f_{i-1} and f_i in ascending order. This is necessary, because only one further bad test is allowed (see the exercise from week 4). (3 points)

5. Two solution options are given below. (6 points)

Using DFS: We do DFS and traverse the edges according to the following rules. Whenver DFS(u) calls DFS(v), we traverse the edge (u, v) in this direction, and traverse (v, u) when DFS(v) is finished (and we are back to DFS(u)). Whenever a neighbor is already marked, we have found a back edge, and we traverse it immediately forth and back. Since every edge belongs to one of these two cases, every edge is traversed exactly once in each direction.

Using induction on the number of edges: Assume that such a walk W of length $2(m-1)$ is possible in every connected graph G with $m-1$ edges. We append a new edge (u, v) to G; one of u, v may be a new node. As soon as W reaches u (this will happpen since the graph is connected), we go from u to v and back, and continue with W.

6.1. An instance of Minimum Spanning Tree is a graph G with edge costs, and a threshold k for the cost of a solution. Let G have n nodes and the largest edge cost h. We construct an instance $f(G)$ of Maximum Spanning Tree: The graph is the same, but every cost c is replaced with $h - c$. Every spanning tree has $n-1$ edges, thus: G has a spanning tree of cost at most k if and only if $f(G)$ has a spanning tree of cost at least $(n-1)h - k$. Function f is computable in polynomial time. The reverse reduction works in the same way. (3 points)

6.2. Removal of a set $F \subset E$ destroys all cycles if and only if $(V, E \setminus F)$ is a spanning tree. This observation yields the following reduction. An instance of Maximum Spanning Tree is a graph G with edge costs. Let G have n nodes and sum of edge costs s. We construct an instance $f(G)$ of Cycle Deletion: Graph and edge costs remain the same. The threshold is chosen as $s - k$ because: G has a spanning tree of cost at least k if and only if $f(G)$ has a cycle-destroying edge set F with cost at most $s - k$. The calculation needs only polynomial time. The reverse reduction works in the same way. (4 points)

6.3. Cycle Deletion is polynomial-time reducible to Maximum Spanning Tree (by 6.2) which is polynomial-time reducible to Minimum Spanning Tree (by 6.1). The latter problem is polynomial-time solvable. Since polynomial-time reducibility is transitive, this yields a polynomial-time algorithm for Cycle Deletion. (3 points)

Remark: Points are not always proportional to the length or difficulty of a solution but also reflect the importance of a topic or skill.