Algorithms Exam
TIN093/DIT093/DIT602

Course: Algorithms

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

Date, time: 13th March 2024, 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 (14 points)

Let C be a collection of strings with symbols from some fixed finite alphabet. The same strings may appear many times in C. Therefore we want to produce a "compressed" representation of C that contains only the distinct strings, and the multiplicity of every string, i.e., the number of occurrences in C, if it is larger than 1. Let n be the total length of all strings in C.

For instance, (exam, pass, exam, pass, pass, good, happy, celebrate, happy) with $n = 43$ shall be shortened to (2 exam, 3 pass, good, celebrate, 2 happy).

Of course, this requires comparisons of symbols and counting, but if these operations are poorly organized and redundant, one can easily use more computation time than necessary. In fact, this basic problem can be solved in $O(n)$ time. We sketch the first "half" of such an algorithm:
Create a directed tree, whose edges are labeled by symbols. Every node has a counter, which is initially 0 (when the node is created). The initial tree consists of a root node only. Take the strings from C one by one, and proceed as follows for every such string $a_1a_2a_3 \dots a_m$: Start at the root and follow the directed path of edges labeled a_1, a_2, a_3, etc. If an edge is missing, say after some a_k, then create a path of new edges for the rest of the string, labeled by $a_{k+1}, \dots, a_m$, and attach it to the tree. When the string ends, increment the counter of the node where you currently are.

1.1. Explain why this tree construction needs only $O(n)$ time. Make use of the fact that the alphabet is fixed. (4 points)

1.2. How do you get, from this tree, the desired list of all distinct strings in C and their multiplicities, in $O(n)$ time? That is, explain the procedure and motivate its correctness and the time bound. (7 points)

1.3. Is there an algorithm that solves this problem even faster than in $O(n)$ time in the worst case? If you think so, give an idea how this can work.
If you do not think so, give a clear reason why it is impossible. (3 points)

Problem 2 (16 points)

This problem deals with compression of strings as well, but in a different setting.
A substring of a string $a_1 \dots a_n$ is any string of consecutive symbols $a_i \dots a_j$, that is, a segment of $a_1 \dots a_n$ that starts at some position i and ends at some position j.
We are given a string A of n symbols, $A = a_1 \dots a_n$, and a fixed finite set S of substrings that are known to appear frequently. We consider a simple compression method that works as follows: Look for pairwise disjoint substrings of A that are equal to some strings in S and replace each of them with some special symbol. (That means, one special symbol is reserved for every string in S.) Every replacement of a substring of length ℓ shortens A by $\ell - 1$. We also say that $\ell - 1$ symbols are "saved" by that.
The problem is now to select pairwise disjoint substrings of $a_1 \dots a_n$ (appearing in S) whose replacement saves, in total, as many symbols as possible. We define $OPT(j)$ as the maximum number of symbols that could be saved if the string $a_1 \dots a_j$ were given.

2.1. How would you compute $OPT(j)$ ($j = 1, \dots, n$) efficiently? Explain. (6 points)

2.2. How much time does a dynamic programming algorithm based on 2.1 need? Provide enough details to motivate your time bound, as a function of n. Remember that S is fixed, hence its size may be considered to be a constant. (4 points)

2.3. We may instead reduce our problem to Weighted Interval Scheduling and solve it with the known algorithm. Explain the details and state and motivate a time bound for the overall procedure, including the reduction. (6 points)

Problem 3 (10 points)

Imagine the following scenario: We wish to solve some complicated maximization problem. The possible values of solutions are all integers k in the range $1, \ldots, n$. An algorithm is available, which can test for every given k whether a solution of value *exactly* k exists. This algorithm needs $O(t(n))$ time, for some known function t. It remains to find the largest such k.
Of course, we can run the algorithm for every k and then take the largest positive answer. This would need $O(t(n) \cdot n)$ time. But it would be very nice to apply binary search to find the largest k already in $O(t(n) \cdot \log n)$ time. The question arises whether this is doable.
Below we state five different conditions that describe possible additional knowledge about the problem. Some definitions first: The term *majority* means "more than half". We call a problem *convex* if it has the following property: Whenever $i < k < j$, and a problem instance has two solutions of values i and j, respectively, then it also has some solution of value k.

3.1. We know nothing special about the problem.
3.2. We know that solutions exist for the majority of values k.
3.3. We know that the problem is convex.
3.4. We know that a solution exists for $k = 1$, and the problem is convex.
3.5. We know that solutions exist for the majority of values k, and the problem is convex.

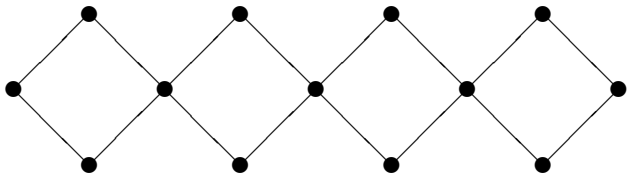
In each of the five cases above, answer the following questions: Can we solve our problem in $O(t(n) \cdot \log n)$ time? Briefly explain how you accomplish this, or why you think that the mentioned condition is not sufficient. (2 points for each case)

Problem 4 (12 points)

Recall that the Interval Partitioning problem asks to partition a given set of intervals into k subsets, each of which contains only pairwise disjoint intervals. Next, the Coloring problem asks to paint the nodes of a given graph with k colors, such that any two nodes joined by an edge receive different colors. Note that k is not constant, but some value of k is given as a part of any problem instance. Suppose that the widely accepted conjecture $\mathcal{P} \neq \mathcal{NP}$ is true.

4.1. Give a polynomial-time reduction from Interval Partitioning to Coloring. That is, describe the reduction, motivate the polynomial time, and show the required equivalence of instances. (8 points)

4.2. Can you also, conversely, reduce Coloring to Interval Partitioning in polynomial time? Motivate your positive or negative answer by a strict argument, without hand waving. (4 points)



Problem 5 (8 points)

In this exercise we consider undirected graphs $G = (V, E)$. Remember a few definitions: A subgraph of G is a graph $H = (W, F)$ with $W \subseteq V$ and $F \subseteq E$. A connected subgraph of G is a subgraph which is also connected. A connected component of G is a connected subgraph that cannot be extended (by further nodes) to a larger connected subgraph.
In most problems we are interested in *some* valid solution. Now let us consider some problems where we want *all* valid solutions to a given instance.

5.1. Can we always output all shortest paths between all pairs of nodes in G is polynomial time? (3 points)

5.2. Can we always output all connected subgraphs of G is polynomial time? (2 points)

5.3. Can we always output all connected components of G is polynomial time? (1 point)

5.4. Can we always output all cliques in G is polynomial time? (2 points)

Motivate every answer: Either give a polynomial-time algorithm, or refer to a known one, or give a strict argument for impossibility.

Hint: The depicted graph might help you find the correct answers to some of these questions.

Solutions (attached after the exam)

1.1. For each of the n symbols in all strings, $O(1)$ operations are done: following the next edge in the tree or creating a new edge, and adding 1 to a counter when the symbol was the end of the string. The correct next edge to be followed can be found by $O(1)$ comparisons of symbols, since the alphabet is fixed, that is, it has constant size. (4 points)

1.2. List all tree nodes systematically, e.g., by BFS. From every node v with a counter value $c > 0$, follow the path back to the root, and concatenate the labels of the traversed edges in reverse order. This yields the corresponding string and its multiplicity c. This is correct since the counter was incremented each time when a copy of the same string ended in v, and there is a one-to-one correspondence between strings and end nodes. Since each of the distinct strings is followed backwards only once, the time does not exceed $O(n)$. (7 points)

1.3. This is impossible. Every algorithm for this problem must at least read the input: If we skip any symbol, we cannot know the entire string, hence the correct output cannnot be guaranteed. (3 points)

2.1. We may formally set $OPT(0) = 0$. To compute $OPT(j)$, we distinguish some cases. If no substring from S ends at a_j, then $OPT(j) = OPT(j-1)$. If some substrings from S end at a_j, we may or may not select one of them for replacement. In more detail, if $a_i \dots a_j$ is in S and we replace it with its special symbol, then we save $OPT(i-1) + j - i$ symbols. Hence $OPT(j)$ is the maximum of $OPT(j-1)$ and of all expressions $OPT(i-1) + j - i$ that satisfy the mentioned condition. (6 points)

2.2. For every position j we must check which strings from S end there, as substrings of the given string. Since S contains a constant number of strings of some constant maximum length, this takes $O(1)$ time. By the same argument, the number of operations needed to calculate every $OPT(j)$ is $O(1)$, hence the overall time is $O(n)$. Backtracing (to recover an actual solution) does not exceed this time bound, as usual. (4 points)

2.3. For all strings in S we determine their occurrences in A. For every such occurrence we create a corresponding interval and assign to it a weight equal to its length minus 1. For the reason mentioned in 2.2, all this is doable in $O(n)$ time, and $O(n)$ intervals are constructed. Hence the algorithm for

Weighted Interval Scheduling runs in $O(n)$ time as well. – This is essentially the same algorithm as in 2.2, just described differently. (6 points)

3.1. Impossible. Since arbitrary combinations of Yes and No can appear, we must test all n values of k in the worst case. (2 points)

3.2. This does not help. Solutions may exist for all k in the range $1, \ldots, n/2$, but still we have to find the largest solution among the remaining values $k > n/2$, thus we are back to case 3.1, with ca. $n/2$ values. (2 points)

3.3. This does not help either. A solution might exist for only one value k, and in the worst case we find it only be testing all values. (2 points)

3.4. The condition implies that solutions exist for all $k \leq m$ and do not exist for all $k > m$, where m is some unknown cut-off value. We can determine m by binary search: If some k has a solution (has no solution) then $m \geq k$ ($m < k$) can be concluded. (2 points)

3.5. The condition implies that a solution exists for $k = \lfloor n/2 \rfloor$. Now we can continue as in 3.4. The only difference is that the left end of the range to be searched is $k = \lfloor n/2 \rfloor$ rather than $k = 1$. (2 points)

4.1. Represent every interval by a node, and for every pair of intersecting intervals, create an edge joining the corresponding nodes. Obviosuly, this can be done in polynomial time. If a partitioning into k subsets (as specified) exists, this yields a partitioning of the nodes into k subsets each of which is an independent set. Hence we can paint all nodes of such a subset with the same color, such that k colors are sufficient. Conversely, if a k-coloring exists, we can partition the node set into k subsets being independent. This partitions the set of intervals into k subsets whose intervals are pairwise disjoint. (8 points)

4.2. The Coloring problem is NP-complete (even for every fixed $k > 2$, and "even more so" when k is part of the instance). A polynomial-time reduction to Interval Partitioning would imply NP-completeness of the latter problem. But this problem is also known to be in P, a contradiction. Hence the answer is negative. (4 points)

5.1. No. Consider graphs as in the hint, and shortest paths between the leftmost and rightmost node. In every other step we have the choice between two nodes, and every combination yields some shortest path. Hence already the number of shortest paths is exponential. (3 points)

5.2. No. This follows instantly from 5.1, since every path is also a connected subgraph. (2 points)

5.3. Yes, even in linear time; this was shown in the course. (1 point)

5.4. No. If G itself is a clique, then every subset of nodes induces a clique, hence the number of cliques is exponential. (2 points)

An alternative argument is: The Clique problem is NP-complete. If we could list all cliques in polynomial time, we could also select one clique of the desired size in the end. However, this argument relies on the conjecture that P is not equal to NP.