CSCE629 Analysis of Algorithms

Homework 1

Texas A&M U, Fall 2019 *08/30/19*

Student: Venkata Sameer Kumar Betana Bhotla *Due: 09/06/19*

UIN: 728009992

Instructions. Typeset your submission by LaTeX, and submit in PDF format. Your solutions will be graded on *correctness* and *clarity*. You should only submit work that you believe to be correct, and you will get significantly more partial credit if you clearly identify the gap(s) in your solution. You may opt for the "Ill take 15%" option (details in Syllabus). A random subset of the problems will be graded.

You may collaborate with others on this problem set. However, you must **write up your own solutions** and **list your collaborators and any external sources** for each problem.

1. (Piazza) **Attention: this problem is due Friday, August 30, 11:59pm CDT.**
   (a) (2 points) Enroll on Piazza `https://piazza.com/tamu/spring2019/csce440640/`.
   (b) (3 points) Post a note on Piazza describing: 1) a few words about yourself; 2) your strengths in CS (e.g., programming, algorithm, ...); 3) what you hope to get out of this course; and 4) anything else you feel like sharing. See instructions on how to post a note `https://support.piazza.com/customer/en/portal/articles/1564004-post-a-note`.

   The purpose is to help me know you all, and also get you known to your fellow students.

Solution:

(a) I have enrolled in the Piazza for the CSCE 629 Analysis of Algorithms Fall 2019 course.

(b) I have also posted an introductory note about myself on the Piazza group. My note can be found on Piazza by going to the 70th post. It says "*Introduction: Sameer Kumar Hello everyone, I am Sameer Kumar. I am a first year PhD student from Mechanical Engineering department. This is my first semester. My research interests are reinforcement learning, robotics and learning & control theory. From this course I am expecting to learn how to design algorithms and how to analyze an algorithm. I hope that this will help me in my research area. Also I have a general curiosity in logic and mathematics (as a subject). I am a Numberphile! I hope everyone has a great semester*".

2. (10 points) (Order of growth rate) Take the following list of functions and arrange them in ascending order of growth rate. That is, if function $g(n)$ immediately follows function $f(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$.

- $f_1(n) = n^{2.5}$
- $f_2(n) = \sqrt{2n}$
- $f_3(n) = n + 10$
- $f_4(n) = 10n$
- $f_5(n) = 100^n$
- $f_6(n) = n^2 \log n$

Solution: Based on intuition and functional analysis from Calculus we can write a general order of time complexities which is universally true. The order of time complexities goes like this ($c > 0$) *(source)*:

Name	Runtime
Constant	$O(1)$
Log-Logarithmic	$O(\log\log(n))$
Logarithmic	$O(\log(n))$
Polylogarithmic	$O(\log(n)^2)$
Fractional power	$O(n^{1/c})$
Linear	$O(n)$
Quasilinear	$O(n\log(n))$, $O(\log(n!))$
Polynomial	$O(n^c)$
Quasi-polynomial	$O(n^{\log\log(n)})$, $O(n^{\log(n)})$
Sub-exponential	$O(2^{n^{1/c}})$
Exponential	$O(2^{n^c})$
Factorial	$O(n!)$
Upper Bound of Factorial	$O(n^n)$
Double Exponential	$O(2^{2^n})$

Using above table we can immediately arrange the functions given in the question. After arranging above functions in ascending order we get $f_2(n) = \sqrt{2n} < f_3(n) = n + 10 < f_4(n) = 10n < f_6(n) = n^2 \log n < f_1(n) = n^{2.5} < f_5(n) = 100^n$. The order in which these functions are arranges such that if function $g(n)$ immediately follows function $f(n)$, then $f(n)$ is $O(g(n))$.

We can check if each pair satisfies the above condition. We know that $O(g(n))$ is defined as $O(g(n))$ = $\{f(n)$: if there exist a positive constant c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$. For any pair in ascending order if we can prove

that there exists a positive constant c and n_0 then $f(n)$ is $O(g(n))$. Lets check for all pairs in ascending order (we don't need to check for all possible pairs because we know that Transitivity property holds true for O-functions *(source)*):

- $f_2(n) = O(f_3(n))$ i.e., $\sqrt{2n} = O(n+10)$. For some c and n_0 we have $\sqrt{2n} \leq c(n+10), n \geq n_0$. If we chose $c = 1$ and $n_0 = 2$ then we can see that the inequality holds and hence $f_2(n) = O(f_3(n))$ is true.
- $f_3(n) = O(f_4(n))$ i.e., $n + 10 = O(10n)$. For some c and n_0 we have $n + 10 \leq c(10n), n \geq n_0$. If we chose $c = 1$ and $n_0 = 3$ then we can see that the inequality holds and hence $f_3(n) = O(f_4(n))$ is true.
- $f_4(n) = O(f_6(n))$ i.e., $10n = O(n^2 \log n)$. For some c and n_0 we have $10n \leq c(n^2 \log n), n \geq n_0$. If we chose $c = 2$ and $n_0 = 10$ then we can see that the inequality holds and hence $f_4(n) = O(f_6(n))$ is true.
- $f_6(n) = O(f_1(n))$ i.e., $n^2 \log n = O(n^{2.5})$. For some c and n_0 we have $n^2 \log n \leq c(n^{2.5}), n \geq n_0$. If we remove the common term n^2 from both side then we will be left with $\log n$ on LHS and n on RHS and we already know that $\log n$ is grows slower than n; hence we can chose any value for the constants. If we chose $c = 1$ and $n_0 = 100$ then we can see that the inequality holds and hence $f_6(n) = O(f_1(n))$ is true.
- $f_1(n) = O(f_5(n))$ i.e., $n^{2.5} = O(100^n)$. For some c and n_0 we have $n^{2.5} \leq c(100^n), n \geq n_0$. If we chose $c = 1$ and $n_0 = 10$ then we can see that the inequality holds and hence $f_1(n) = O(f_5(n))$ is true.

3. (10 points) (Order of growth rate) Take the following list of functions and arrange them in ascending order of growth rate.

- $g_1(n) = 2^{\sqrt{n}}$
- $g_2(n) = 2^n$
- $g_3(n) = n(\log n)^3$
- $g_4(n) = n^{4/3}$
- $g_5(n) = n^{\log n}$
- $g_6(n) = 2^{2^n}$
- $g_7(n) = 2^{n^2}$
- $g_8(n) = n!$

Solution: Using table presented in the previous solution we can immediately arrange the functions given in the question. After arranging above functions in

ascending order we get $g_3(n) = n(\log n)^3 < g_4(n) = n^{4/3} < g_5(n) = n^{\log n} < g_1(n) = 2^{\sqrt{n}} < g_2(n) = 2^n < g_7(n) = 2^{n^2} < g_8(n) = n! < g_6(n) = 2^{2^n}$. The order in which these functions are arranges such that if function $g(n)$ immediately follows function $f(n)$, then $f(n)$ is $O(g(n))$.

We can also use properties of exponent to make a immediate conclusion about the functions which are powers of 2. We know that if $a, b > 0, \eta > 1$ and $a < b$ then $\eta^a < \eta^b$. Using this we can write $\sqrt{n} < n < n^2 < 2^n$ hence $2^{\sqrt{n}} < 2^n < 2^{n^2} < 2^{2^n}$. To figure out the rest of the inequalities we can follow the same procedure used in above solution.

- $g_3(n) = O(g_4(n))$ i.e., $n(\log n)^3 = O(n^{4/3})$. For some c and n_0 we have $n(\log n)^3 \leq c(n^{4/3}), n \geq n_0$. If we remove the common term n from both side then we will be left with $(\log n)^3$ on LHS and $n^{1/3}$ on RHS and we already know that $\log n$ is grows slower than n; hence we can chose any value for the constants. If we chose $c = 1$ and $n_0 = 10$ then we can see that the inequality holds and hence $g_3(n) = O(g_4(n))$ is true.
- $g_4(n) = O(g_5(n))$ i.e., $n^{4/3} = O(n^{\log n})$. For some c and n_0 we have $n^{4/3} \leq c(n^{\log n}), n \geq n_0$. If we chose $c = 1$ and $n_0 = 10^2$ then we can see that the inequality holds and hence $g_4(n) = O(g_5(n))$ is true.
- $g_5(n) = O(g_1(n))$ i.e., $n^{\log n} = O(2^{\sqrt{n}})$. For some c and n_0 we have $n^{\log n} \leq c(2^{\sqrt{n}}), n \geq n_0$. To compare this we can write $n^{\log n}$ as $2^{(\log n)^2}$ and $2^{\sqrt{n}}$ as $2^{\sqrt{n}\log 2}$. Since the base of the powers are same hence we can directly compare the powers; if we compare the powers then $(\log n)^2 < \sqrt{n}\log 2 \implies 2^{(\log n)^2} < 2^{\sqrt{n}\log 2} \implies n^{\log n} < 2^{\sqrt{n}}$ If we chose $c = 1$ and $n_0 = 10^2$ then we can see that the inequality holds and hence $g_5(n) = O(g_1(n))$ is true.
- $g_8(n) = O(g_6(n))$ i.e., $n! = O(2^{2^n})$. For some c and n_0 we have $n! \leq c(2^{2^n}), n \geq n_0$. We can take logarithm for both terms. By taking log we get $\sum_{i=1}^{n} \log i$ and $2^n \log 2$. We know that $\sum_{i=1}^{n} \log i < n \log n$ and $n \log n < 2^n$. Hence $\sum_{i=1}^{n} \log i < 2^n \log 2$ and if we log of two terms less than inequality then terms inside the log follow the same sign. Which give us $n! \leq (2^{2^n})$. If we chose $c = 1$ and $n_0 = 10$ then we can see that the inequality holds and hence $g_8(n) = O(g_6(n))$ is true.

Using all arguments presented above we can arrange everything in ascending order as presented above.

4. (15 points) (Understanding big-O notation) Assume you have functions f and g such that $f(n)$ is $O(g(n))$. For each of the following statements, decide whether you think it is true or false and give a proof or counterexample.

(a) $\log_2 f(n)$ is $O(\log_2 g(n))$.

(b) $2^{f(n)}$ is $O(2^{g(n)})$.

(c) $f(n)^2$ is $O(g(n)^2)$.

Solution: For this problem I am assuming that function $f(n)$ and $g(n)$ are greater than 1 and monotonically increasing functions.

(a) **TRUE** Given that we have functions $f(n)$ and $g(n)$ such that $f(n)$ is $O(g(n))$. Which means for some $c > 0$ and $n \geq n_0$ we have

$$\begin{aligned} f(n) &\leq cg(n) \\ \log_2 f(n) &\leq \log_2 c + \log_2 g(n) \; (\because if a < b \implies \log a < \log b) \\ \log_2 f(n) &\leq c_1 \log_2 g(n). \end{aligned}$$

Since c and n_0 are constants, there must be a constant c_1 such that $c_1 \geq (\log c / \log g(n_0)) + 1$. Which means we can find some $c_1 > 0$ and $n \geq n_0$ such that we have $0 \leq \log_2 f(n) \leq c_1 \log_2 g(n)$. Therefore, if $f(n)$ is $O(g(n))$ we have $\log_2 f(n) = O(\log_2 g(n))$.

(b) **FALSE** We can produce a counterexample to disprove the provided statement. For example take $f(n) = 2n$ and $g(n) = n$. We know that $2n = O(n)$. But $2^{2n} \neq O(2^n)$. Assume if $2^{2n} = O(2^n)$ then for some $c > 0$ and $n \geq n_0$ we have

$$\begin{aligned} 2^{2n} &\leq c2^n \\ 2^n 2^n &\leq c2^n \\ 2^n &\leq c. \end{aligned}$$

But above inequality doesn't hold because c is a constant and for some big values of n, n will be greater than c. Hence given statement is false. If $f(n)$ is $O(g(n))$ then we can't say $2^{f(n)}$ is $O(2^{g(n)})$.

(c) **TRUE** Given that we have functions $f(n)$ and $g(n)$ such that $f(n)$ is $O(g(n))$. Which means for some $c > 0$ and $n \geq n_0$ we have

$$\begin{aligned} f(n) &\leq cg(n) \\ f(n)^2 &\leq c^2 g(n)^2 \; (\because if a < b \implies a^2 < b^2) \\ f(n)^2 &\leq c_1 g(n)^2. \end{aligned}$$

Which means we can find some $c_1 = c^2 > 0$ and $n \geq n_0$ such that we have $0 \leq f(n)^2 \leq c_1 g(n)^2$. Therefore, if $f(n)$ is $O(g(n))$ we have $f(n)^2 = O(g(n)^2)$.

5. (Basic proof techniques) Read the chapter on `Proof by Induction` by Erickson (`http:`

//jeffe.cs.illinois.edu/teaching/algorithms/notes/98-induction.pdf), and do the following.

(a) (10 points) Prove that every integer (positive, negative, or zero) can be written in the form $\sum_k \pm 3^k$, where the exponents k are distinct non-negative integers. For example:

$$42 = 3^4 - 3^3 - 3^2 - 3^1$$
$$25 = 3^3 - 3^1 + 3^0$$
$$17 = 3^3 - 3^2 - 3^0$$

(b) (10 points) A binary tree is *full* if every node has either two children (an internal node) or no children (a leaf). Give two proofs (proof by induction and proof by contradiction) of the following statement: in any full binary tree, the number of leaves is exactly one more than the number of internal nodes.

Solution:

(a) **Proof by principal of mathematical induction**:
First lets prove the theorem for non-negative integers. Proving for non-positive integers would be the same, we have to just switch the sign.

Let n be an arbitrary non-negative integer. Assume that any non-negative integer less than n can be written as the sum of distinct powers of 3. There are two cases to consider. Either $n = 0$ or $n \geq 1$.

- The base case $n = 0$ is trivial; the elements of the empty set are distinct and sum to zero.
- Suppose $n \geq 1$ and let $m = \lfloor n/3 \rfloor$. Because $0 \leq m < n$, the inductive hypothesis implies m can be written as the sum of distinct powers of 3. Thus, $3m$ can also be written as the sum of distinct powers of 3, each of which is greater than 3^0. If n is a multiple of 3 then $n = 3m$ and we are done. Otherwise, $n = 3m \pm 3^0$ or $n = 3m \pm (3^1 - 3^0)$, which are again the sum of distinct powers of 3.

In either case we conclude that n can be written as the sum of distinct powers of 3. Similarly it can be shown that any non-positive integers can be written as sum of distinct powers of 3. Hence, all integers (positive, negative, zero) can be represented as sum of distinct powers of 3. □

(b) **Proof by principal of mathematical induction**:
Let n be an arbitrary non-negative integer. Assume that for any *full* binary tree T with internal nodes $I < n$ has no. of leaves L exactly one more than

the number of internal nodes I. There are two cases to consider. Either $n = 0$ or $n \geq 1$.

- For the base case let $n = 0$. In this case tree must only contain of root node, having no children because binary must be full. Which means we have one leaf nodes i.e., $L = 1$ and internal nodes $n = 0$ (based on our assumption). Hence, no. of leaves is exactly one more than the no. of internal nodes.
- Let $n \geq 1$. By induction hypothesis we have for any *full* binary tree T with internal nodes $I < n$ has no. of leaves L exactly one more than the number of internal nodes I.

  Let T be a *full* binary tree with n internal nodes. Then the root of T has two sub trees, namely left sub tree T_L and right sub tree T_R. Suppose T_L has I_L internal nodes and T_R has I_R internal nodes. Notice that neither T_L nor T_R can be empty because we have a *full* binary tree T. Also, every internal node of T_L and T_R are part of T. Apart from two sub trees we have only one internal node that is the root. So, using the above two arguments we can conclude that total internal nodes of T must be equal to sum of internal nodes in both sub trees and the root i.e., $n = I_L + I_R + 1$.

  From induction hypothesis we know T_L must have $I_L + 1$ leaves and T_R must have $I_R + 1$ leaves. Every leaf in both sub trees is part of the T (root is an internal node not a leaf), hence total leaves L in the entire tree T will be $I_L + I_R + 2$. Using above expression and after simplification we get $L = n + 1$, i.e., total leaves are exactly one more than total internal nodes.

In both cases, we conclude that in any full binary tree, the number of leaves is exactly one more than the number of internal nodes. □

Proof by contradiction:
Let n be an arbitrary non-negative integer. Assume that if n is the total no. of internal nodes in *full* binary tree T it doesn't satisfy the proposition given in the theorem. Also assume that for any *full* binary tree T with internal nodes $I < n$ has no. of leaves L exactly one more than the number of internal nodes I. There are two cases to consider. Either $n = 0$ or $n \geq 1$.

- For the base case let $n = 0$. In this case tree must only contain of root node, having no children because binary must be full. Which means we have one leaf nodes i.e., $L = 1$ and internal nodes $n = 0$ (based on our assumption). Hence, no. of leaves is exactly one more than the no. of internal nodes.
- Let $n \geq 1$. By induction hypothesis we have for any *full* binary tree T

with internal nodes $I < n$ has no. of leaves L exactly one more than the number of internal nodes I.

Let T be a *full* binary tree with n internal nodes. Then the root of T has two sub trees, namely left sub tree T_L and right sub tree T_R. Suppose T_L has I_L internal nodes and T_R has I_R internal nodes. Notice that neither T_L nor T_R can be empty because we have a *full* binary tree T. Also, every internal node of T_L and T_R are part of T. Apart from two sub trees we have only one internal node that is the root. So, using the above two arguments we can conclude that total internal nodes of T must be equal to sum of internal nodes in both sub trees and the root i.e., $n = I_L + I_R + 1$.

From induction hypothesis we know T_L must have $I_L + 1$ leaves and T_R must have $I_R + 1$ leaves. Because no counterexample is smaller than n. Every leaf in both sub trees is part of the T (root is an internal node not a leaf), hence total leaves L in the entire tree T will be $I_L + I_R + 2$. Using above expression and after simplification we get $L = n + 1$, i.e., total leaves are exactly one more than total internal nodes.

In both cases, we conclude that in any full binary tree, the number of leaves is exactly one more than the number of internal nodes. But this contradicts our assumption that n doesn't satisfy the given proposition stated in the theorem. We have contradicted our initial assumption, hence it has to be false. □

6. (Reduction) Given a set X of n Boolean variables $x_1, \ldots, x_n$; each can take the value 0 or 1 (equivalently, "false" or "true"). By a term over X, we mean one of the variables x_i or its negation $\bar{x}_i$. Finally, a clause is simply a disjunction of distinct terms $t_1 \vee t_2 \vee \ldots \vee t_\ell$. (Again, each $t_i \in \{x_1, x_2, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}\}$.) We say the clause has length l if it contains l terms.

   A truth assignment for X is an assignment of the value 0 or 1 to each x_i; in other words, it is a function $v : X \to \{0, 1\}$. The assignment v implicitly gives $\bar{x}_i$ the opposite truth value from x_i. An assignment satisfies a clause C if it causes C to evaluate to 1 under the rules of Boolean logic; this is equivalent to requiring that at least one of the terms in C should receive the value 1. An assignment satisfies a collection of clauses $C_1, \ldots, C_k$ if it causes all of the C_i to evaluate to 1; in other words, if it causes the conjunction $C_1 \wedge C_2 \wedge \ldots C_k$ to evaluate to 1. In this case, we will say that v is a satisfying assignment with respect to $C_1, \ldots, C_k$; and that the set of clauses $C_1, \ldots, C_k$ is *satisfiable*. For example, consider the three clauses

$$(x_1 \vee \overline{x_2}), (\overline{x_1} \vee \overline{x_3}), (x_2 \vee \overline{x_3}) .$$

   A truth assignment v that sets all variables to 1 is not a satisfying assignment, because

it does not satisfy the second of these clauses; but the truth assignment v' that sets all variables to 0 is a satisfying assignment.

We can now state the `Satisfiability` Problem, also referred to as SAT:

Given a set of clauses $C_1, \ldots, C_k$ over a set of variables $X = \{x_1, \ldots, x_2\}$, does there exist a satisfying truth assignment?

Suppose we are given an oracle $\mathcal{O}$ which can solve the `Satisfiability` problem. Namely if we feed a set of clauses to $\mathcal{O}$, it will tell us YES if there is a satisfying assignment for all clauses or NO if there exists none. Show that you can actually find a satisfying truth assignment v for $C_1, \ldots, C_k$ by asking questions to $\mathcal{O}$. Describe your procedure, and analyze how many questions you need to ask.

Solution: Problems of the above category fall into topics of computational complexity theory and computability theory. When we are trying to solve a *NP-hard* or *NP-complete* problem we are interested in two things (a) for the given problem does a solution exist (b) what is the solution/solution set. (I am not going to formally describe and write rigorous definitions for all the theory because it will take lot space)

Every NP problem can be described in two versions as follows

- Problem of deciding whether a solution exist is know as *decision problem*. In these kind of problems we try to formulate the problems in terms of decision, meaning if we pose a question we want to know if the answer is 'yes' or 'no'. A method for solving a decision problem, given in the form of an algorithm, is called a decision procedure for that problem. An example of a decision problem is deciding whether a given natural number is prime.
- Problem of finding the exact solution is know as *search problem*. Generally, we are interested in finding a solution not just knowing if one exists. Solving the search the problem is harder than the decision problem.

The Boolean satisfiability problem or SAT problem is known to be NP-complete. In the search version of the problem, we can't stop merely at saying 'yes' or 'no' to the question of whether the given formula is satisfiable; if the answer is yes we must actually find and output a satisfying assignment. In decision version of the problem 'yes' or 'no' answers to the question of whether the given formula is satisfiable is enough. In the SAT problem we are given the conjunctive normal form (CNF) which are nothing but conjunction of clauses (or a single clause). Here our decision problem is if CNF is satisfiable and search problem is find a satisfying assignment to CNF if one exist else output NONE.

Note: If we can solve the search problem then we can certainly solve the decision problem. In SAT if we are given a CNF, and if we can find a satisfying assignment or

tell that one does not exist, we can certainly say whether or not there exists a satisfying assignment. So search is harder; if we can solve it we can certainly solve decision.

We are given a oracle machine $\mathcal{O}$ which solves the decision problem and we don't need to worry how it does. We need to build a procedure to solve the "equivalent" search problem for the SAT problem. This is know as self-reduction and using the oracle we can build a procedure to solve the search problem in polynomial time. That is, search reduces to decision for SAT.

Now our problem is that we are given a CNF ϕ and we want to find a satisfying assignment to if one exist. To help us we have, oracle $\mathcal{O}$. We can use oracle $\mathcal{O}$ to test whether or not ϕ is satisfiable. Also oracle $\mathcal{O}$ can be invoked on any formula hence we will invoke it on formulas constructed out if ϕ and extract the truth assignment one bit at a time. For example consider,

$$\phi(x_1, x_2, x_3, x_4) = (x_1 \vee \neg x_2) \wedge (x_2 \vee x_3 \vee \neg x_1) \wedge (x_4 \vee \neg x_3)$$

First ask $\mathcal{O}$ if ϕ is satisfiable. If it says no, we output NONE and we are done since there is no solution. If it says yes, then we need to find values for each Boolean variable. Consider two formulas, which we denote as ϕ_0 and ϕ_1. Each of the new formulas we created are will be one less variable, formed by setting Boolean variable x_1 to one of the truth values.

- $\phi_0(x_2, x_3, x_4) = \phi(0, x_2, x_3, x_4)$. Substituting $x_1 = 0$ and simplifying the original equation will give $\neg x_2 \wedge (x_4 \vee \neg x_3)$.
- $\phi_1(x_2, x_3, x_4) = \phi(1, x_2, x_3, x_4)$. Substituting $x_1 = 1$ and simplifying the original equation will give $(x_2 \vee x_3) \wedge (x_4 \vee \neg x_3)$.

The key point is that *one of the following must be satisfiable*. Because x_1 must be either 1 or 0. If we ask $\mathcal{O}$ (since this is a decision problem) which is the correct solution then we will know what is the correct truth assignment for the Boolean variable x_1 (search problem being solved with the help of oracle). Suppose $\mathcal{O}$ says yes to $x_1 = 0$ then it is the solution; if it says no then $x_1 = 1$ is the solution. Because there must exist some satisfying assignment to ϕ with x_1 being 0 or 1. We can keep repeating this process for all Boolean variables in the ϕ CNF.

Finally we make total of $(n + 1)$ calls to the oracle, and end up with a satisfying assignment. Here n is the total no. of Boolean variables present in the problem and for each Boolean variable there is a associated call to the oracle. Finally, the additional one call comes from the initial check done on the ϕ to know if a solution exists or not in the first place. Also, note that if there is no solution to the SAT problem then we make only 1 call to the oracle. Additionally, note that we can have Boolean variable with two possible correct assignments but we are only interested in one.