Monash University
Faculty of Information Technology
2nd Semester 2023

FIT2014
Exercises 10
Complexity: P, NP, Polynomial Time Reductions

SOLUTIONS

1. We saw in Exercise Sheet 7, Question 4, that the time complexity $t(n) = 2n+1$. Since this is always $\leq 3n$ (for $n \geq 1$), we have $t(n) = O(n)$.

Note, we could use other constants here. We could have used 4 or 1729 instead of 3. For big-O notation, it doesn't matter what constant factors we use when deriving it, since the big-O notation sweeps constant factors under the carpet.

In this case, we can even use a somewhat smaller constant, if we wished. For example, we could use 2.1 instead of 3. What happens if we do? It is not true that $t(n) \leq 2.1n$; e.g., if $n = 3$ then $t(3) = 2 \times 3 + 1 = 7$ but $2.1 \times 3 = 6.3$, so $t(n) \not\leq 2.1n$. But that's ok, because $t(n) \leq 2.1n$ for all *sufficiently large* n. More specifically, if $n \geq 10$, then $2.1n = 2n + (n/10) \geq 2n + 1 = t(n)$. In fact, any constant > 2 can be made to do the job.

Having said that, when using big-O notation, there is no need to get the constant as small as possible. So it's fine to just go straight to a constant that is large enough to do the job and simple enough to keep the reasoning simple. The constant 3 served that purpose well in this case.

2. [1] (a) Each edge joins two vertices. So, since there are no isolated vertices, the number of vertices is at most twice the number of edges:

$$n \leq 2m.$$

(b) The number of clauses in φ_G is $4n + 3m$, where n and m are the numbers of vertices and edges, respectively, of G. Assume the time taken to construct each clause is at most c, where c is a constant. Then the time taken to construct G is $\leq c(4n + 3m)$. Using our upper bound from part (a),

$$\begin{aligned} \text{time complexity} &\leq c(4n + 3m) \\ &\leq c(4 \cdot 2m + 3m) \qquad \text{(by (a))} \\ &= 11c\,m \\ &= O(m), \end{aligned}$$

since $11c$ is a constant.

[1] Thanks to FIT2014 staff Anuja Dharmaratne and Carl Vu for spotting some small errors in an earlier version.

(c) There is at most one edge between any pair of vertices (since G has no multiple edges), and no other edges (since G has no loops). So

$$\begin{aligned} m &\leq \text{number of pairs of vertices of } G. \\ &= \binom{n}{2} \\ &= \frac{n(n-1)}{2} \\ &\leq \tfrac{1}{2}n^2. \end{aligned}$$

(d)

$$\begin{aligned} \text{time complexity} &\leq c(4n + 3m) \\ &\leq c(4n + 3 \cdot \tfrac{1}{2}n^2) \qquad \text{(by (c))} \\ &= 4c\,n + \tfrac{3}{2}c\,n^2 \\ &= O(n^2) \end{aligned}$$

since, in big-O notation, the faster-growing $\frac{3}{2}c\,n^2$ dominates the linear term $4cn$. The constants, too, are unimportant: the key point is that n^2 dominates n.

(e) The input file for G contains a decimal number to represent the number of vertices of G followed by the list of edges. Each edge consists of a decimal representation of its two endvertices, with two hyphens between them.

Each decimal number contains at least one digit. So the number of characters required to represent G in the specified format is $\geq 4m + 1$.

(f) We saw in (b) that the time complexity is $O(m)$, which means that there is a constant α such that, for all sufficiently large m, the time complexity is $\leq \alpha m$.

We saw in (e) that $|G| \geq 4m + 1$. For simplicity, we'll just use $|G| \geq 4m$. So $m \leq |G|/4$.

So

$$\text{time complexity } \leq \alpha m \leq \alpha|G|/4 = (\alpha/4)\,|G|.$$

So

$$\text{time complexity } = O(|G|).$$

(g) It is a polynomial-time reduction from GRAPH 3-COLOURABILITY to SATISFIABILITY.

(h)[2] The vertices are numbered $1, 2, \ldots, n$, and the number of decimal digits required to represent a number in this range is $\lfloor \log_{10} n \rfloor + 1$. This expression is a bit awkward to use algebraically. But we are heading towards a big-O expression, so we only need

[2] Thanks to FIT2014 staff member Vee Voon Yee for a correction.

upper bounds, and exact values of constant factors are unimportant. So let's be a bit loose and use the upper bound

$$\#\text{ digits to represent a vertex} \quad \leq \quad 2\log n.$$

We have one number to specify the number of vertices, then two vertex numbers and two hyphens for each edge. So

$$\text{length of } G \quad \leq \quad 2\log n + m((4\log n) + 2).$$

(i)

$$\begin{aligned}
\text{length of } G &\leq 2\log n + m((4\log n) + 2) && \text{(from (h))} \\
&\leq 2\log(2m) + m((4\log(2m)) + 2) && \text{(by (a))} \\
&= 2\log 2 + 2\log m + (2 + 4\log 2)m + 4m\log m \\
&= O(m\log m),
\end{aligned}$$

since the $m\log m$ term dominates the others.

3.

Let $\tau_t(n)$ be the times taken by your program to solve this problem, t years from now, on an input of size n.

At present, we have $\tau_0(n) = an^5$.

According to Moore's Law, processor speed doubles every two years. So, assuming all programs are run on computers whose processor speed over time follows Moore's Law exactly (which is a big simplification, especially for small t),

$$\tau_t(n) = \tau_0(n)/2^{t/2}. \tag{1}$$

Also, observe that, if the input is increased by a factor of k, then (for fixed t) the time goes up by a factor of k^5:

$$\tau_t(kn) = k^5\tau_t(n). \tag{2}$$

(a)

The condition given in this question can be written in an equation:

$$\tau_t(4n) = \tau_0(n).$$

We want to solve this for t.

Using (1), the equation becomes

$$4^5\tau_t(n) = \tau_0(n).$$

Then, using (2) gives

$$\frac{4^5\tau_0(n)}{2^{t/2}} = \tau_0(n),$$

so that

$$2^{t/2} = 4^5.$$

Solving this gives $t = 20$. So you have to wait for 20 years.

(b)

Now, $\tau_0(n) = 2^n$.

Equation (1) still holds.

Equation (2) no longer holds. Instead we have

$$\tau_t(kn) = \frac{\tau_0(kn)}{2^{t/2}} = \frac{\tau_0(n)^k}{2^{t/2}}.$$

The condition for (b) is once again

$$\tau_t(4n) = \tau_0(n).$$

Doing the appropriate substitutions gives

$$\frac{\tau_0(n)^4}{2^{t/2}} = \tau_0(n).$$

This simplifies to

$$\frac{\tau_0(n)^3}{2^{t/2}} = 1,$$

which in turn gives

$$\tau_0(n)^3 = 2^{t/2}.$$

Substituting $\tau_0(n) = 2^n$ gives $2^{3n} = 2^{t/2}$, so $t = 6n$. Using our lower bound, $n \geq 40$, we find $t \geq 240$. So your friend has to wait for at least 240 years.

This illustrates how even a large polynomial time complexity (and, in most practical contexts, running time n^5 would be considered slow) gives much more computational power, and handles increases of input size much better, than an exponential time complexity. [3]

4.

(a)

If m is the length of the input string in bases, then we are given that the running time is $5m^3$. Each base needs two bits to specify it, so $n = 2m$ and $m = n/2$. Therefore the running time is $5m^3 = 5(n/2)^3 = \frac{5}{8}n^3$.

(b)

No change is needed. Big-O running time absorbs constant factors.

[3] Thanks to Han Phan (FIT2014 tutor, 2013) for spotting a bug in an earlier version of this solution, and fixing it.

5.

An adjacency matrix is an $n \times n$ matrix of bits, so its length, as an input, is n^2 bits.

If algorithm A solves problem Π in polynomial time, then there exists k such that its time complexity is

$$\begin{aligned} &= O(\text{ (length of string representing } \textit{adjacency matrix})^k \text{)} \\ &= O((n^2)^k) \\ &= O(n^{2k}). \end{aligned}$$

We will now derive an inequality that relates the lengths of adjacency matrices and edge lists.

We use a *lower* bound for the length of the edge list.

Suppose the graph has m edges. Then the edge list consists of m pairs (i, j). For each edge, the pair (i, j) specifies its two endpoints i and j.

The total length of the edge list is clearly $\Omega(m)$. Since $m = \Omega(n)$ (since the graph has no isolated vertices)[4], we find that the length of the edge list is $\Omega(n)$, which means that

$$n = O(\text{ (length of string representing edge list) }). \tag{3}$$

So the time complexity is:

$$\begin{aligned} &= O(n^{2k}) \\ &= O(\text{ (length of edge list)}^{2k} \text{)} \\ &\quad \text{(using (3)).} \end{aligned}$$

Since the exponent $2k$ is fixed (i.e., does not depend on n), this shows that the algorithm still runs in polynomial time when the input is represented as an edge list.

A good exercise is to do the reverse problem: show that, if a problem on graphs can be solved in polynomial time when the input is given as an edge list, then it can also be solved in polynomial time when the input is instead given as an adjacency matrix.

The way input is represented does affect how quickly algorithms run in practice, so it is important. But most "reasonable" representation schemes should not affect the classification of problems as polynomial-time solvable or not.

6.

(a)

Here is a polynomial-time verifier for EDGE-COLOURING.

Input: a graph G, a positive integer k.
Certificate: a function $f : E(G) \to \{1, 2, \ldots, k\}$ that assigns a colour to each edge.

[4]Thanks to FIT2014 tutor Thomas Hendrey for a correction at this point.

For each vertex v of G:
{
    Make a list of all the colours that appear on the edges incident with v.
    If any colour appears more than once in this list, then REJECT.
    Otherwise, continue.
}
ACCEPT, since we only reach this point if every vertex has no colour repeated among the colours of its edges.

It is clear that $(G, k) \in$ EDGE-COLOURING if and only if there exists a certificate such that this algorithm accepts (since the certificate is just the edge-colouring). So this is indeed a verifier for EDGE-COLOURING.

The verifier runs in polynomial time: the outer loop is executed n times (where n is the number of vertices of G), and for each iteration, the inner loop involves a test that all list elements belong to the specified colour set and are distinct, which takes polynomial time using standard list methods (and using the fact that the number of edges incident with a vertex is $\leq n$).

So we have a polynomial-time verifier for EDGE-COLOURING.

(b) The problem can be solved in polynomial time when

- $k = 1$: in this case, G is not 1-edge-colourable unless it is a union of disjoint edges (i.e., edges that have no vertices in common);
- $k = 2$: in this case, G is not 2-edge-colourable unless it is a union of disjoint paths and even-length circuits. In the latter case, assigning one colour to any edge of one of the paths or circuits forces the colours of all other edges in that component.

7. There are problems with the following parts of the proof:

(2): In fact, not all these algorithms run in polynomial time. The conversion of a regular expression to an FA for the same language can take exponential time, because the size of the FA it creates may be exponential in the size of the regular expression. (Recall that an NFA with k states has an equivalent FA with $\leq 2^k$ states, but the number of states of the FA can be about as large as 2^k in some cases.)

So, although the algorithm given is indeed a verifier for SHORTEST REGEXP, it is not a *polynomial-time* verifier, so it does not establish that SHORTEST REGEXP $\in$ NP.

In fact, it is not known that SHORTEST REGEXP $\in$ NP. The current belief is that SHORTEST REGEXP $\notin$ NP, and that solving it requires exponential time. This is a striking contrast with the situation for Finite Automata.

(3): It is possible that the two Finite Automata are labelled differently, even if they are equivalent (i.e., their states could be numbered in a different order). So, if we just use the state numbers that come to us from the state minimisation algorithm, we may think the two FA are different even though they might be the same.

But it is not efficient to search through all possible relabellings of the states of (say) $A^{\min}$ to see if any of them give $B^{\min}$. For two Finite Automata with n states, testing all possible ways of matching up their states would involve testing $n!$ possibilities, which would take time that is exponential in the number of states in the FA (and, to make matters worse, we have already seen that the number of states in these Finite Automata could already be exponential in the size of the original regular expression!). The problem of matching up states to demonstrate equivalence is related to GRAPH ISOMORPHISM; that problem is not yet known to be in P, although there are some theoretical reasons to believe it is not NP-complete, and in practice it seems to be more tractable than NP-complete problems.

Fortunately, there are better approaches to testing equivalence of two minimised FAs.

- One approach is that outlined in Lecture 20 (slides 11–12): from $A^{\min}$ and $B^{\min}$, construct a new Finite Automaton C which accepts the symmetric difference of the languages accepted by $A^{\min}$ and $B^{\min}$, i.e.,

  $$L(C) \;=\; (L(A^{\min}) \cap \overline{L(B^{\min})}) \cup (\overline{L(A^{\min})} \cap L(B^{\min})).$$

  Here, we use the fact that, for any two FAs X, Y, we can efficiently construct FAs to recognise each of $\overline{L(X)}$, $L(X) \cup L(Y)$, and $L(X) \cap L(Y)$.

  We know that $L(A^{\min}) = L(B^{\min})$ if and only if $L(C) = \emptyset$, and this is easy to test: just use standard graph searching methods (e.g., Breadth-First Search) to determine whether there is a directed path from a Start State to a Final State in C. If there is such a path, then $L(C) \neq \emptyset$ and so $A^{\min}$ and $B^{\min}$ are not equivalent; if there is no such path, then $L(C) = \emptyset$, so $A^{\min}$ and $B^{\min}$ are equivalent.
- Another approach is to give each of the FAs a new labelling as follows.[5] For each state in the FA, determine the first word, in lexicographic order, which puts the FA in that state. This word becomes the new label of the state, and each state gets its own unique label.

  To do this, we follow transitions in alphabetical order, doing a depth-first traversal of the underlying directed graph, labelling the states as we go.

  The labelling obtained from this process is determined solely by the transitions of the FA; it is independent of whatever labelling we might have

[5] Thanks to FIT2014 tutor Thomas Hendrey for suggesting this approach.

used for the states previously. We just need to check if these new labellings of the states of each FA demonstrate the equivalence between them. We no longer need to check all possible relabellings of one FA to see if it is equivalent to the other.

(5) & (7): See (2) & (3) above.

8.

(a)

Here is a polynomial-time verifier for NEARLY SAT.

Input: a Boolean expression φ in CNF.
Certificate: a truth assignment t for φ.
Initialisation: `numberOfSatisfiedClauses` $:= 0$.
$m :=$ the total number of clauses of φ.
For each clause C of φ:
{
    If some literal in C is True under t:
        increment `numberOfSatisfiedClauses`
        /* This clause is satisfied. */
}
If `numberOfSatisfiedClauses` $\geq m - 1$ then ACCEPT.
Otherwise, REJECT.

It clearly always halts, and runs in polynomial time: the algorithm essentially looks at each literal in the expression at most once, checking the truth assignment for the corresponding variable to see if the literal is True, and seeing what effect this has on satisfaction of the clause. The size of the input is at least as large as the total number of literals, and the work done per literal is just consultation of a list of truth values and a small amount of checking.

The input expression φ belongs to NEARLY SAT if and only if there exists a truth assignment such that the number of unsatisfied clauses is ≤ 1. This is precisely the condition that there exists a certificate such that the above verifier accepts. Therefore this algorithm is indeed a verifier for NEARLY SAT.

So it is, in fact, a polynomial-time verifier for NEARLY SAT.

(b)

Input: a Boolean expression φ in CNF.
Introduce a new variable, y, that does not appear in φ.
Create two new singleton clauses, one containing just y and the other containing just $\neg y$.
Let φ' be the expression $\varphi \wedge (y) \wedge (\neg y)$.

Output: φ'.

The output expression φ' is clearly in CNF.

The function $\varphi \mapsto \varphi'$ is clearly polynomial-time computable.

If $\varphi \in$ SAT then there is a satisfying truth assignment for φ. Augment this truth assignment by assigning a truth value to y. (It does not matter whether y is True or False.) This new truth assignment satisfies all the clauses of φ and exactly one of the two singleton clauses we added. So it satisfies all but one of the clauses of φ'. So $\varphi' \in$ NEARLY SAT.

If $\varphi' \in$ NEARLY SAT, then there exists a truth assignment to φ' that satisfies all, or all but one, of the clauses of φ'. So there is at most one clause that is not satisfied. Observe that it is impossible for any truth assignment to satisfy both y and $\neg y$. So one of these clauses may be unsatisfied. It follows that all of φ must be satisfied (since at most one clause of φ' can be unsatisfied). Therefore $\varphi \in$ SAT.

Therefore the function $\varphi \mapsto \varphi'$ is a polynomial-time reduction from SAT to NEARLY SAT.

Supplementary exercises

9.

Refer to the solution of Exercise Sheet 9, Q8. There, we found mapping reductions (a) from L to L^{even}, and (b) from L^{even} to L.

Furthermore, part (d) showed that both these mapping reductions are polynomial-time computable.

Recall that if A and B are languages such that $A \leq_P B$ and $B \in$ P then $A \in$ P.

Since our two reductions are polynomial-time computable, we have $L \leq_P L^{\text{even}}$ and $L^{\text{even}} \leq_P L$. It follows that $L \in$ P if and only if $L^{\text{even}} \in$ P.

10.

(a)

Input: x

$z := 1$ // *z is smallest factor of x found so far.*

For $y := 2$ to $\lfloor \sqrt{x} \rfloor$ // *y is the number to be tested to see if it is a factor.* // *Smallest factor must be $\leq \sqrt{x}$.*

{

If y is a factor of x, then put $z := y$

}

If $z > 1$ then output z, else report that x is prime.

(b)

In the worst case, the number of loop iterations is $\sqrt{x}$. Each such iteration requires one divisibility test, a comparison, an increment, and possibly an assignment. We'll call this a constant number of steps, which is sweeping a lot under the carpet but it will do for the purposes of this exercise.

So we'll take the complexity to be $O(\sqrt{x})$. If we write n for the input size then, since x is in unary, $n = x$. So the complexity is $O(\sqrt{n}) = O(n^{1/2})$.

(c)

This complexity is polynomial, since it is $O(n^k)$ for some constant k (in this case, $k = \frac{1}{2}$).

(d)

If x is in binary, then its length as a string of bits is $\lfloor \log_2 x \rfloor + 1$. This is close to $\log_2 x$, and we'll use this approximation for now.

Writing n for the input size again, we now have $n = \log_2 x$, so $x = 2^n$. Substituting this into our complexity gives $O(x^{1/2}) = O((2^n)^{1/2}) = O(2^{n/2})$.

(e)

The complexity is now exponential in the input size. The algorithm does not run in polynomial time.

(f)

If we use base $b > 2$, then input size is $n \approx \log_b x$ and $x = b^n$. The complexity is $O(b^{n/2})$. This also is exponential time, not polynomial time.

(g)

(i)

It would be very difficult. Even something as simple as an increment would take a long time.

(ii)

In this encoding, a number is prime if and only if its encoding consists of a single 1, preceded by a (possibly empty) sequence of 0s. Its representation has the form $(0, 0, \ldots, 0, 1)$. We just have to check, for every position in the input sequence except the last, whether or not it is 0, and then we check that the last is 1. This takes constant time per position, and the number of positions is $O(n)$, where n is the input length. So the time complexity is $O(n)$, which is linear time, and certainly polynomial time.

Incidentally, the input length n is the number of prime numbers $\leq x$. This number is approximately $x/\log x$, according to the Prime Number Theorem. But we don't need to know this for this question, since we are expressing complexity in terms of the input length n, and not in terms of the number x itself.

11.

(a) We prove that P = NP[SHORT] by proving set containment in each direction.

($\subseteq$)

Every language in P has a polynomial-time decider. But this is also a polynomial-time verifier with empty certificate. So it has a polynomial-time verifier with certificate length ≤ 100. So it belongs to NP[SHORT].

Therefore P $\subseteq$ NP[SHORT].

($\supseteq$)

Let $L \in$ NP[SHORT]. By definition of NP[SHORT], it has a polynomial-time verifier V with certificate length ≤ 100. We can use this in a polynomial-time decider for L, as follows.

Input: string x, being a possible input for L.
For every string y of ≤ 100 characters
{
    Run the L-verifier, V, on input x with certificate y.
    If V accepts (x, y), then ACCEPT.
    Otherwise, continue.
}
REJECT.

This decider runs in polynomial time, since it the number of outer loop iterations is the number of strings of ≤ 100 characters, which is $1 + 2 + 2^2 + 2^3 + \cdots + 2^{99} + 2^{100}$, which is $< 2^{101}$, which is constant (although large)[6], and the time spent inside the loop is dominated by the time spent running V, which is bounded by a polynomial in the input size since V is a polynomial-time verifier.

We now show that the decider is indeed a decider for L, by showing that $x \in L$ if and only if the decider accepts x.

($\Longrightarrow$)

If $x \in L$, then there is some certificate y with $|y| \leq 100$ such that V accepts (x, y). So the decider will encounter y during its iteration over all possible certificates of length ≤ 100, and will accept in that iteration because V accepts then.

($\Longleftarrow$)

On the other hand, if the decider accepts x, then the only way that can happen is if some y causes V to accept (x, y) during a main loop iteration. That is only possible if $x \in L$, since V is a verifier for L.

[6] It is routine to extend this argument to larger alphabets.

So our decider is indeed a polynomial-time decider for L. So $L \in \mathrm{P}$.

Therefore $\mathrm{P} \supseteq \mathrm{NP}[\mathrm{SHORT}]$.

We have now shown both the required containments. So $\mathrm{P} = \mathrm{NP}[\mathrm{SHORT}]$.

(b) $f(n) = c \log n$ will do the job. We can replace 100 by $c \log_2 n$ throughout the above argument, and it will still work. The crucial observation that the number of certificates is bounded above by about $2^{c \log_2 n} = n^c$, so the number of iterations of the main loop in the decider is polynomially bounded.

Question: what happens if we use $f(n) = c(\log n)^2$? What about $f(n) = c \log n \log \log n$? What about other functions that grow more quickly than $c \log n$ (for any fixed c)?

(c) It's tempting to take some polynomial, such as $f(n) = n^2$. The class $\mathrm{NP}[n^2]$ contains a "large chunk" of NP; it includes, for example, SATISFIABILITY (for which the certificate is a list of binary values True/False for all the variables, so is in fact shorter than the input length), and 3-COLOURABILITY, and EDGE-COLOURABILITY, and HAMILTONIAN CIRCUIT, and much else. But, as far as we know, some problems in NP have verifiers whose certificates seem to need more than n^2 characters. The same issue arises if any polynomial function is used instead of n^2. It is not currently known whether there exists any constant c such that $\mathrm{NP} = \mathrm{NP}[n^c]$.

Instead, try $f(n) = 2^n$. This function grows so quickly that it eventually overtakes any polynomial, a fact we use below. We prove that $\mathrm{NP} = \mathrm{NP}[2^n]$ by proving the two required containments.

($\supseteq$)

Let $L \in \mathrm{NP}[2^n]$. A polynomial-time verifier with a certificate length bound is still a polynomial-time verifier, so L has a polynomial-time verifier, so $L \in \mathrm{NP}$.

($\subseteq$)

Let $L \in \mathrm{NP}$. Let V be a polynomial-time verifier for L. There exist constants a, k such that V only ever uses at most an^k characters in the certificate.[7] For sufficiently large n, we have $an^k < 2^n$. Let N be a constant such that $an^k < 2^n$ for all $n > N$.

We construct a new verifier for L as follows. All inputs of length $\leq N$ are treated as special cases and solved directly, by looking up a table if necessary. (This may be a large table, but it is nonetheless finite, and its size does not depend on n.) So, for these cases, no certificate is used. For inputs of length $> N$, the verifier V is used, and the length of certificate for these inputs is $< 2^n$. So, for *all* inputs, the certificate length for this new verifier is $< 2^n$. So we have shown that $L \in \mathrm{NP}[2^n]$.

[7] The polynomial time bound can be used as the bound on certificate length here, since at most one certificate character can be read in each time-step.