Tutorial work - 2-5

Algorithms And Complexity (University of Melbourne)

Department of Computing and Information Systems
COMP90038 Algorithms and Complexity Tutorial Week 2

Possible Answers and Discussion

The exercises

1. Consider the usual (unsigned) binary representation of integers. For example, `10110010` represents 178, and `000011` represents 3.

   (a) If we call the bits in an n-bit word $x_{n-1}, x_{n-2}, \ldots, x_2, x_1, x_0$ (so x_0 is the *least significant* bit), which natural number is denoted by $x_{n-1}x_{n-2}\cdots x_2x_1x_0$?

   (b) Describe, in English, an algorithm for converting from binary to decimal notation.

   (c) Write the algorithm in (pseudo-) code.

   (d) Describe, in English, how to convert the decimal representation to binary.

   Answer.

   (a) Assuming unsigned representation, n bits allows us to represent the integers from 0 to $2^n - 1$, inclusive. The bit-string $x_{n-1}x_{n-2}\cdots x_2x_1x_0$ denotes $\sum_{i=0}^{n-1} 2^i \cdot x_i$.

   (b) Here is one method, expressed in English. We build the decimal-notation number by visiting the binary digits from left to right, constructing the result in an "accumulator". Start with the accumulator being 0. As long as there is a next bit to process, double the value of the accumulator, and add the value of that next bit.

   (c) Notice how all sorts of ambiguities creep in when we use natural languages. You might easily get the impression that what was meant with the previous answer was "as long as there is a next bit, double the accumulator, and then, after all that doubling, add something." It isn't clear from the structure of the English sentence that "and add the value" is part of what should be done for each bit (and the use of a comma before "and" didn't help). In pseudo-code this should be made unambiguous:

   function BINTODEC($x_{n-1}x_{n-2}\cdots x_2x_1x_0$)
   $a \leftarrow 0$
   for $i \leftarrow 0$ to $n-1$ **do**
   $a \leftarrow 2a + x_{n-i-1}$
   return a

   (d) To convert decimal representation d to binary, the natural way is to generate the bits from right to left. To get the rightmost bit, calculate the parity of d, that is, find d mod 2. Then halve d (rounding down). Now repeat this process, to get the remaining bits. More precisely:

   function DECTOBIN(d)
   $n \leftarrow 0$
   while $d \neq 0$ **do**
   $x_n \leftarrow d \mod 2$
   $d \leftarrow \lfloor d/2 \rfloor$
   $n \leftarrow n + 1$
   return $x_{n-1}x_{n-2}\cdots x_2x_1x_0$

   This works for non-negative n.

2. Which of the following should we accept as an *algorithm* for computing the area of a triangle whose side lengths are positive numbers a, b, and c (for sides A, B, and C, respectively)?

   (a) $S = \sqrt{p(p-a)(p-b)(p-c)}$, where $p = (a+b+c)/2$

   (b) $S = \frac{1}{2}ab\sin\theta$, where θ is the angle between sides A and B

   (c) $S = \frac{1}{2}ah_A$, where h_A is the height to base A

   Answer.

   (a) This is a fine algorithm, as long as the square root operation is a primitive operation available on our computing device (or we know how to calculate square roots).

   (b) This is a problematic formulation, because, even if the sine function is considered a primitive, there is no indication of how to compute the angle θ.

   (c) Again, the formula says "the height to base A", without any indication of how to find that. So we can't really call that an algorithm.

3. In the first lecture we discussed different ways of calculating the greatest common divisor of two positive integers. A mathematician might simply write

$$gcd(x, y) = max\{z \mid \exists u, v : x = uz \wedge y = vz\}$$

   and suggest we develop a functional programming language that allows us to write this, leaving it to the language implementation to translate this definition to an efficient algorithm. Do you imagine a time when we may be able to do this? If we restrict our attention to functions like *gcd* which takes a pair of integers and returns an integer, do you think we may some day be able to automatically turn any function definition into a working algorithm?

   Answer. The point of asking this question is to call attention to that fact that there are functions that are not *computable*. For some natural and precise meaning of "algorithm" (which we shall not discuss here), a non-computable function is one that cannot be captured by an algorithm. The (computable) function *gcd* has type $(\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers, and $\mathbb{N} \times \mathbb{N}$ is the set of pairs of natural numbers. Even if we restrict our interest to functions of that type, it turns out that there are vastly more functions than there are algorithms. (For those interested in the mathematics of this, there can be no more algorithms than natural numbers—discuss. And there is no surjective function from $\mathbb{N}$ to $(\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$, let alone a bijection, by an argument known as "diagonalisation".)

   It would be nice if we could say that non-computable functions are of academic interest only, because they are all weird functions with no practical application. However, that is not the case. Many natural, and important, functions cannot be captured by an algorithm.

4. Consider the following problem: You are to design an algorithm to determine the best route for a subway passenger to take from one station to another in a city such as Kolkata or Tokyo.

   (a) Discuss ways of making the problem statement less vague. In particular, what is "best" supposed to mean?

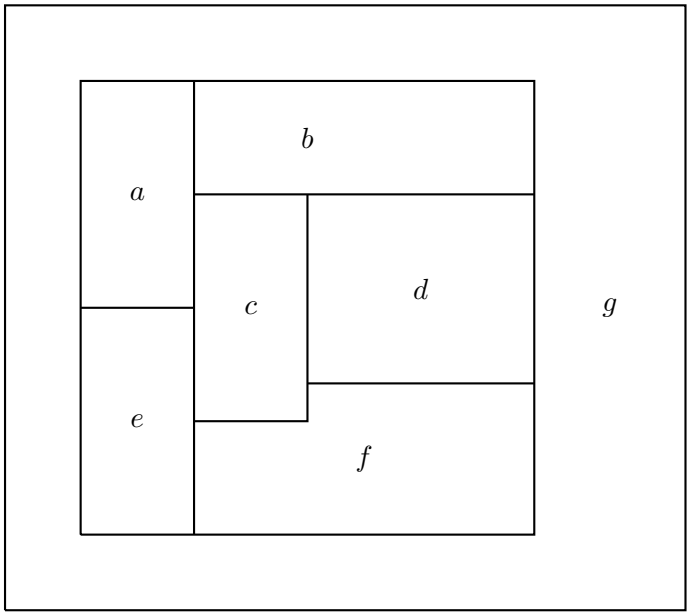
   (b) How would you model this problem by a graph?

   Answer.

   (a) In this context, "best" can mean many things. We may want to minimize the travel time, the number of train stops, the number of train changes, or some combination of these.

(b) The natural choice is to let nodes correspond to stations. Then there is an edge between two nodes iff the stations that correspond to the incident nodes are directly connected by a train line. If travel time is important (part of the definition of "best" route) then we need a weighted graph. In this case, we may also need to indicate how long it takes to change train, noting that a station may be on several lines. That information could be kept separately, or as annotations to stations, or we could do what some subway maps do: have several nodes for the same station.

5. Consider the following map:



(a) A cartographer wants to colour the map so that no two neighbouring countries have the same colour. How few colours can she get away with?

(b) Show how to reduce the problem to a graph-colouring problem.

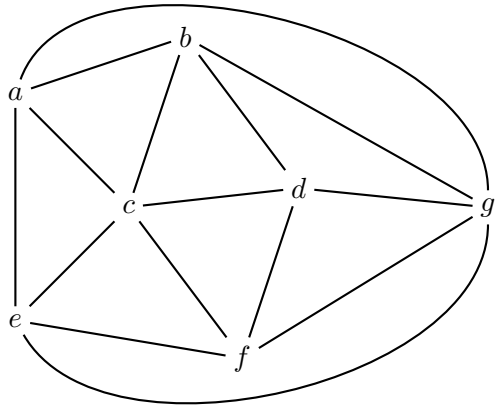
Answer.

(a) It turns out that four colours suffice for *any* planar map, no matter how complicated the map. Of course, for some maps fewer colours may be enough. The one given here does require four.

The result about the four colours has an interesting history. It has been conjectured to hold since 1852, and many incorrect proofs of the theorem have been given. Natural approaches to a proof involve extensive case analysis, too many cases to go through by hand. In 1976, Appel and Haken from the University of Illinois at Urbana-Champaign completed a proof by programming a computer to handle most of the tedious case analysis. At the time, there was much debate about the validity of such a proof: If it is too long for anybody to follow by reading, is it really a proof? Who says the program they used worked correctly—surely we also need a proof of *its* correctness. Since then, many independent computer-assisted proofs have been produced, and there is now a general consensus that the so-called four-colour theorem holds.

It is easy to determine whether a map can be coloured with one, two, or four colours (in the last case, the decision procedure can say 'yes' without even looking at its input). However, the case of three colours appears to be hard. Technically it is "NP-complete", a concept we will discuss towards the end of this course.

(b) We can generate an undirected planar graph (*planar* meaning one that has no edges crossing), by placing a node in each of the "countries" a–g and connecting two nodes iff the corresponding countries have a common border. This is a general construction; it works for all maps. Now the question "how many colours are needed for the map?" becomes "how many colours are needed to colour the nodes of the graph, so that no two neighboring nodes have the same colour?" For our example, the graph looks like this:



6. You have to search for a given number n in a *sorted* list of numbers.

   (a) How can you take advantage of knowing that the list is represented as a linked (and sorted) list?

   (b) How can you take advantage of knowing the list is represented as an array?

   Answer.

   (a) We can stop searching as soon as we find (n or) a number greater than n.

   (b) With an array we can use binary search.

Department of Computing and Information Systems
COMP90038 Algorithms and Complexity Tutorial Week 3

Sample answers

The exercises

7. One way of representing an undirected graph G is using an *adjacency matrix*. This is an $n \times n$ matrix, where n is the number of nodes in G. In this matrix, we label the rows and the columns with the names of G's nodes. For a pair (x, y) of nodes, the matrix has a 1 in the (x, y) entry if G has an edge connecting x and y; otherwise the entry has a 0. (As the graph is undirected, this is somewhat redundant: (x, y) and (y, x) will always be identical.) A *complete* graphs is one in which x and y are connected, for all x, y with $x \neq y$. We don't exclude the possibility of a *loop*, that is, an edge connecting a node to itself, although loops are rare.

   How can you tell from its matrix whether an undirected graph

   (a) is complete?

   (b) has a loop?

   (c) has an isolated node?

   Answer.

   (a) All elements (except those on the main diagonal) are 1.

   (b) Some element on the main diagonal is 1.

   (c) Some row (and hence some column) has all zeros.

8. Design an algorithm to check whether two given words are anagrams, that is, whether one can be obtained from the other by permuting its letters. For example, *garner* and *ranger* are anagrams.

   Answer. Note that we cannot just say "for each letter in the first word, check that it occurs in the second, and vice versa." (Why not?)

   A nice solution sorts the letters in each word and simply compares the two results. This solution scales well, that is, has good performance even as the words involved grow longer.

9. Here we consider a function `first_occ`, such that `first_occ`(d, s) returns the first position (in string s) which holds any symbol from the collection d (and returns -1 if there is no such position). For simplicity let us assume both arguments are given as strings. For example, `first_occ`("`aeiou`", s) means "find the first vowel in `s`." For $s =$ "`zzzzzzzzzzz`" this should return -1, and for $s =$ "`Phlogiston`", it should return 3 (assuming we count from 0).

   Assuming strings are held in arrays, design an algorithm for this and find its complexity as a function of the lengths of d and s.

   Suppose we know that the set of possible symbols that may be used in d has cardinality 256. Can you find a way of utilising this to speed up your algorithm? Hint: Find a way that requires only one scan through d and only one through s.

Answer. Let m be the length of d and let n be the length of s. The obvious algorithm is to scan through s and, for each symbol x met, go through d to see if that was a symbol we were looking for. In the worst case, when no symbols from d are in s, this means making mn comparisons. (We could also structure the search the other way round: for each element of d, find its first occurrence, and then, based on that, decide which occurrence was the earliest of them all; this also leads to mn comparisons in the worst case.)

Knowing that there are 256 possible symbols that may be used in d allows us to introduce an array `first`, indexed by the symbols. (If we are talking ASCII characters then, conveniently, the indices run from 0 to 255.) Initialise this array so all its elements are, say, -1. Now scan through s. For each symbol `x` in s, if `first[x]` is -1, replace it with `x`'s position. Note that we have not considered d at all so far; we have only recorded, in `first`, where the first occurrence of each symbol is (and left the value as -1 for each symbol that isn't in s).

Now all we need is a single scan through d, to find the value of `first[x]` for each symbol `x` in d. As we find these, we keep the smallest such value. It this value is -1, none of the symbols from d was found in s. Otherwise the value is the first occurrence of a symbol from d.

The complexity in this case is $n + m$. For most reasonable values of m and n, this is smaller than mn, so we have a better algorithm, even taking the overhead of maintaining `first` into account. Essentially we have traded in some space (the array `first`) to gain some time. This kind of *time/space trade-off* is a familiar theme in algorithm design.

10. Gaussian elimination, the classical algorithm for solving systems of n linear equations in n unknowns, requires about $\frac{1}{3}n^3$ multiplications, which is the algorithm's basic operation.

    (a) How much longer should we expect Gaussian elimination to spend on 1000 equations, compared with 500 equations?

    (b) You plan to buy a computer that is 1000 times faster than what you currently have. By what factor will the new computer increase the size of systems solvable in the same amount of time as on the old computer?

    Answer.

    (a) The answer is: 8 times longer, and it does not depend on the particular numbers of equations given in the question. We need $\frac{1}{3}(2n)^3$ operations for $2n$ equations and $\frac{1}{3}n^3$ operations for n. The ratio is $\frac{\frac{1}{3}(2n)^3}{\frac{1}{3}n^3} = \frac{8n^3}{n^3} = 8$.

    (b) If the new machine needs t_{new} units of time for a job, the old one needs $t_{old} = 1000\ t_{new}$ units. Let m be the number of equations handled by the new machine in the time the old machine handles n. The new machine handles m in time $\frac{1}{3}m^3$ and n in time $\frac{1}{3}n^3$. So we have

    $$\frac{1}{3}m^3 = 1000 \cdot \frac{1}{3}n^3$$

    Solving for m we get

    $$m = \sqrt[3]{1000\ n^3} = 10n$$

    So we can now solve systems that are 10 times larger.

11. For each of the following pairs of functions, indicate whether the components have the same rate of growth, and if not, which grows faster.

    (a) $n(n+1)$ and $2000\ n^2$
    (b) $100n^2$ and $0.01n^3$
    (c) $\log_2 n$ and $\ln n$
    (d) $\log_2^2 n$ and $\log_2 n^2$
    (e) 2^{n-1} and 2^n
    (f) $(n-1)!$ and $n!$

    Here ! is the factorial function, and $\log_2^2 n$ is the way we usually write $(\log_2 n)^2$.

Answer.

(a) Same rate of growth.

(b) $0.001n^3$ grows faster than $100n^2$. Namely, as soon as $n > 1$, we have $n^2 < n^3$. So pick the constant c to be 10,000. When $n > 1$, we have $100n^2 < 10,000 \cdot (0.001n^3)$.

(c) Same rate of growth.

(d) $\log_2 n^2 = 2 \log_2 n$, so $\log_2 n^2$ has the same rate of growth as $\log_2 n$. However, $\log_2^2 n$ grows faster; namely $\lim_{n\to\infty} \frac{\log_2 n \cdot \log_2 n}{\log_2 n} = \lim_{n\to\infty} \log_2 n = \infty$.

(e) Same rate of growth, since $2^n = 2 \cdot 2^{n-1}$.

(f) $n!$ grows faster than $(n-1)!$; namely $\lim_{n\to\infty} \frac{n!}{(n-1)!} = \lim_{n\to\infty} n = \infty$.

12. (Levitin 2.3.3.) The sample variance of n measurements $x_1, \ldots, x_n$ can be computed as either

$$\frac{\sum_{i=1}^n (x_i - \overline{x})^2}{n-1} \quad \text{where} \quad \overline{x} = \frac{\sum_{i=1}^n x_i}{n}$$

or as

$$\frac{\sum_{i=1}^n x_i^2 - (\sum_{i=1}^n x_i)^2/n}{n-1}$$

Find and compare the number of divisions, multiplications, and additions/subtractions (additions and subtractions are usually bunched together) that are required for computing the variance according to each of these formulas. For large n, which is better?

Answer. For large n, the second method is better. A bit of counting leads to this table:

	First method	Second method
Divisions	2	2
Multiplications	n	$n+1$
Additions/subtractions	$3n-1$	$2n$

13. Eight balls of equal size are on the table. Seven of them weigh the same, but one is slightly heavier. You have a balance scale that can compare weights. How can you find the heavier ball using only two weighings?

Answer. Put three balls on each weighing pan. If the six balance, use the second weighing to compare the two remaining balls. Otherwise keep just the three balls that were heavier than the three they were compared to. Put one of the three on one pan, and another on the other pan. If one is heavier, that's the heavy ball; otherwise the heavy ball is the one remaining.

14. There are 18 gloves in a drawer: 4 pairs of red gloves, 3 pairs of yellow, and 2 pairs of green. You select gloves in the dark, which means you cannot assess colour nor left/right-handedness. You can check the gloves only after the selection has been made. What is the smallest number of gloves you must pick to have at least one matching pair in the best case? In the worst case?

Answer. The best case is obviously 2 gloves. To analyse the worst case, note that we might pick as many as 4 red, 3 yellow, and 2 green gloves, and yet not have a pair, because for each colour, we happened to take gloves of the same orientation, or "handed-ness". So to be absolutely certain that we have a pair, we need to pick 4+3+2+1 = 10 gloves.

15. After washing 5 distinct pairs of socks, only 8 socks come back from the clothes line. Hence you are left with 4 complete pairs in the best case, and 3 in the worst case. What is the probability of the best case scenario?

Answer. There are $\binom{10}{2} = 45$ ways to select two socks from 10. In five of these cases the two socks have the same colour. These five cases correspond to the best-case scenario, in that we are left with four complete pairs. The probability of this happening is $\frac{5}{45} = \frac{1}{9}$.

Department of Computing and Information Systems
COMP90038 Algorithms and Complexity Tutorial Week 4

Sample answers

The exercises

18. One possible way of representing a polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

is as an array A of length $n + 1$, with $A[i]$ holding the coefficient a_i.

(a) Design a brute-force algorithm for computing the value of $p(x)$ at a given point x. Express this as a function PEVAL(A, n, x) where A is the array of coefficients, n is the degree of the polynomial, and x is the point for which we want the value of p.

(b) If your algorithm is $\Theta(n^2)$, try to find a linear algorithm.

(c) Is it possible to find an algorithm that solves the problem in sub-linear time?

Answer.

(a) Working from right-to-left, the following algorithm is the natural formulation:

```
function PEVAL(A, n, x)
    result ← 0.0
    for i ← n downto 0 do
        summand ← 1.0
        for j ← 1 to i do
            summand ← x × summand
        result ← result + a[i] × summand
    return result
```

The complexity is $\Theta(n^2)$.

(b) Working from left-to-right allows us to avoid many redundant calculations of x^i. It gives an algorithm that is both simpler and more efficient:

```
function PEVAL(A, n, x)
    result ← a[0]
    summand ← 1.0
    for i ← 1 to n do
        summand ← x × summand
        result ← result + a[i] × summand
    return result
```

(c) We cannot solve the problem in less than linear time, because we clearly need to access each of the $n + 1$ coefficients.

19. Trace the brute-force string search algorithm on the following input: The path p is '`needle`', and the text t is '`there_need_not_be_any`'. How many comparisons (successful and unsuccessful) are made?

Answer. 21 character comparisons are made.

20. Assume we have a text consisting of one million zeros. For each of these patterns, determine how many character comparisons the brute-force string matching algorithm will make:

(a) `010001` (b) `000101` (c) `011101`

Answer.

(a) 2×999995 comparisons

(b) 4×999995 comparisons

(c) 2×999995 comparisons

21. Give an example of a text of length n and a pattern, which together constitute a worst-case scenario for the brute-force string matching algorithm. How many character comparisons, as a function of n, will be made for the worst-case example.

Answer. The worst case happens when we have a text of length n consisting of the same character c repeated n times, together with a pattern of length m, consisting of $m-1$ occurrences of c, followed by a single character different from c. In this case, the outer loop is traversed $n-m+1$ times, and each time, m character comparisons are made before failure is detected. Altogether we have $(n-m+1)m = (n+1)m - m^2$ comparisons. As a function of m, this has its maximal value where $n+1-2m = 0$, that is, when the length of the pattern is about half that of the text.

22. The *assignment problem* asks how to best assign n jobs to n contractors who have put in bids for each job. An instance of this problem is an $n \times n$ *cost matrix* C, with $C[i,j]$ specifying what it will cost to have contractor i do job j. The aim is to minimise the total cost. More formally, we want to find a permutation $\langle j_1, j_2, \ldots j_n \rangle$ of $\langle 1, 2, \ldots, n \rangle$ such that $\sum_{i=1}^{n} C[i, j_i]$ is minimized.

Use brute force to solve the following instance:

	Job 1	**Job 2**	**Job 3**	**Job 4**
Contractor 1	9	2	7	8
Contractor 2	6	4	3	7
Contractor 3	5	8	1	8
Contractor 4	7	6	9	4

Answer.

Permutation	**Cost**
1,2,3,4	9+4+1+4 = 18
1,2,4,3	9+4+8+9 = 30
1,3,2,4	9+3+8+4 = 24
1,3,4,2	9+3+8+6 = 26
1,4,2,3	9+7+8+9 = 33
1,4,3,2	9+7+1+6 = 23
2,1,3,4	2+6+1+4 = 13
2,1,4,3	2+6+8+9 = 25
⋮	

and so on. The minimal cost is 13, for permutation $\langle 2, 1, 3, 4 \rangle$.

23. Give an instance of the assignment problem which does not include the smallest item $C[i,j]$ of its cost matrix.

Answer.

	Job 1	**Job 2**
Contractor 1	1	2
Contractor 2	2	4

Department of Computing and Information Systems
COMP90038 Algorithms and Complexity Tutorial Week 5

Sample answers

The exercises

24. Consider the *subset-sum problem*: Given a set S of positive integers, and a positive integer t, find a subset $S' \subseteq S$ such that $\sum S' = t$, or determine that there is no such subset. Design an exhaustive-search algorithm to solve this problem. Assuming that addition is a constant-time operation, what is the complexity of your algorithm?

    Answer:

    function SUBSETSUM(S, t)
        for each S' in POWERSET(S) **do**
            if $\sum S' = t$ **then**
                return *True*
        return *False*

    How can we systematically generate all subsets of the given set S? For each element $x \in S$, we need to generate all the subset of S that happen to contain x, as well as those that do not. This gives us a natural recursive algorithm:

    function POWERSET(S)
        if $s = \emptyset$ **then**
            return $\{\emptyset\}$
        else
            $x \leftarrow$ some element of S
            $S' \leftarrow S \setminus \{x\}$
            $P \leftarrow$ POWERSET(S')
            return $P \cup \{s \cup \{x\} \mid s \in P\}$

    There are 2^n subsets of S. $\sum S'$ calculates the sum of the elements in S', which requires $O(n)$ time. Hence the brute-force solution is $O(n \cdot 2^n)$.

25. Lecture 4's Clinton-Trump challenge is an instance of the *partition problem*: Given n positive integers, partition them into two disjoint subsets so that the sum of one subset is the same as the sum of the other, or determine that no such partition exists. Designing an exhaustive-search algorithm to solve this problem seems somewhat harder than doing the same for the subset-sum problem. Show, however, that there is a simple way of exploiting your algorithm for the subset-sum problem (that is, try to *reduce* the partition problem to the subset-sum problem).

    Answer:

    function HASPARTITION(S)
        $sum \leftarrow \sum S$
        if sum is odd **then**
            return *False*
        return SUBSETSUM($S, sum/2$)

26. Consider the *clique problem*: Given a graph G and a positive integer k, determine whether the graph contains a *clique* of size k, that is, G has a complete sub-graph with k nodes. Design an exhaustive-search algorithm to solve this problem.

Answer: Assume $G = (V, E)$ where V is the set of nodes, and E is the set of edges.

```
function HasClique((V, E), k)
    for each U ⊆ V with |U| = k do
        if IsClique(U, E) then
            return True
    return False

function IsClique(U, E)
    for each (u, u') ∈ U^2 do
        if (u, u') ∉ E then
            return False
    return True
```

There are $\binom{|V|}{k}$ ways of selecting a subset of k nodes. The function IsClique is called for each such subset, and the cost of each call is $O(k^2|E|)$.

27. Consider the special clique problem of finding triangles in a graph (noting that a triangle is a clique of size 3). Show that this problem can be solved in time $O(|V|^3|E|)$.

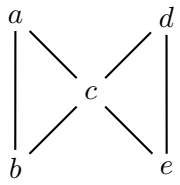
Answer: The number of subsets of size 3 is $\binom{|V|}{3} = \frac{|V|(|V|-1)(|V|-2)}{6}$. For each subset the cost is $O(|E|)$. Altogether the cost of the brute-force approach is $O(|V|^3|E|)$.

28. Draw the undirected graph whose adjacency matrix is

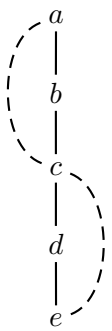
	a	b	c	d	e
a	0	1	1	0	0
b	1	0	1	0	0
c	1	1	0	1	1
d	0	0	1	0	1
e	0	0	1	1	0

Starting at node a, traverse the graph by depth-first search, resolving ties by taking nodes in alphabetical order.

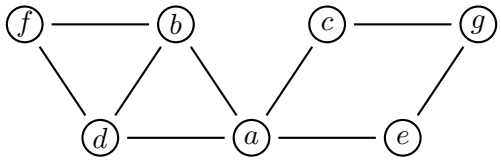
Answer: Here is the graph:



In a depth-first search, the nodes are visited in this order: a, b, c, d, e. The depth-first search tree looks as follows:



29. Consider this graph:



(a) Write down the adjacency matrix representation for this graph, as well as the adjacency list representation (assume nodes are kept in alphabetical order in the lists).

(b) Starting at node a, traverse the graph by depth-first search, resolving ties by taking nodes in alphabetical order. Along the way, construct the depth-first search tree. Give the order in which nodes are pushed onto to traversal stack, and the order in which they are popped off.

(c) Traverse the graph by breadth-first search instead. Along the way, construct the depth-first search tree.

Answer:

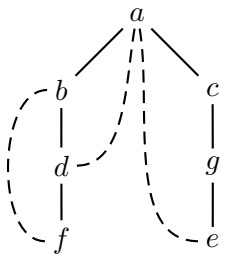
(a) Here is the adjacency matrix:

	a	b	c	d	e	f	g
a	0	1	1	1	1	0	0
b	1	0	0	1	0	1	0
c	1	0	0	0	0	0	1
d	1	1	0	0	0	1	0
e	1	0	0	0	0	0	1
f	0	1	0	1	0	0	0
g	0	0	1	0	1	0	0

The adjacency list representation:

a	$\to b \to c \to d \to e$
b	$\to a \to d \to f$
c	$\to a \to g$
d	$\to a \to b \to f$
e	$\to a \to g$
f	$\to b \to d$
g	$\to c \to e$

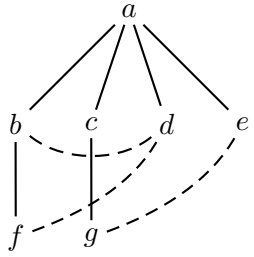
(b) In a depth-first search, the nodes are visited in this order: a, b, d, f, c, g, e.



The order of actions: push a, push b, push d, push f, pop f, pop d, pop b, push c, push g, push e, pop e, pop g, pop c, pop a. The traversal stack develops like so:

$$\begin{array}{ll} f_{4,1} & e_{7,4} \\ d_{3,2} & g_{6,5} \\ b_{2,3} & c_{5,6} \\ a_{1,7} & \end{array}$$

(c) In a breadth-first search, the nodes are visited in this order: a, b, c, d, e, f, g.



30. Explain how one can use depth-first search to identify the connected components of an undirected graph. Hint: Number the components from 1 and mark each node with its component number.

Answer: Instead of marking visited node with consecutive integers, we can mark them with a number that identifies their connected component. More specifically, replace the variable *count* with a variable *component*. In *dfs* remove the line that increments *count*. As before, initialise each node with a mark of 0 (for "unvisited"). Here is the algorithm:

```
mark each node in V with 0 (indicates not yet visited)
component ← 1
for each v in V do
    if v is marked with 0 then
        DFS(v)
        component ← component + 1

function DFS(v)
    mark v with component
    for each vertex w adjacent to v do
        if w is marked with 0 then
            DFS(w)
```

31. The function CYCLIC is intended to check whether a given undirected graph is cyclic.

```
function CYCLIC(⟨V, E⟩)
    mark each node in V with 0
    count ← 0
    for each v in V do
        if v is marked with 0 then
            cyclic ← HASCYCLES(v)
            if cyclic then
                return True
    return False

function HASCYCLES(v)
    count ← count + 1
    mark v with count
    for each edge (v, w) do                    ▷ w is v's neighbour
        if w's mark is greater than 0 then
            return True                        ▷ w has been visited before
        if HASCYCLES(w) then
            return True                        ▷ a cycle can be reached from w
    return False
```

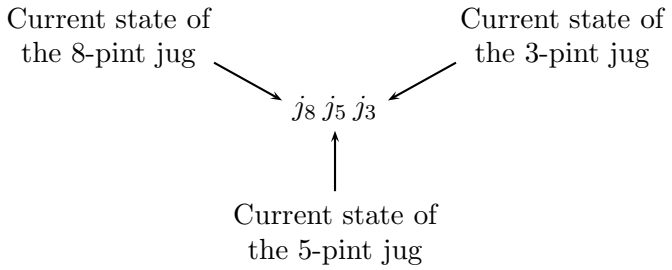
Show, through a worked example, that the algorithm is incorrect. Exercise 35 will ask you to develop a correct algorithm for this problem.

Answer: Take the undirected graph that has just two nodes, a and b, and a single edge. The edge is represented as $\{(a, b), (b, a)\}$ irrespective of whether we use an adjacency matrix or an adjacency list representation. The main functions calls HASCYCLES(a), which results in a being marked and then the recursive call HASCYCLES(b) is made. This results in b being marked, and then b's neighbours are considered. Since the neighbour a has already been marked, the algorithm returns *True*, that is, the graph is deemed cyclic, which is wrong.

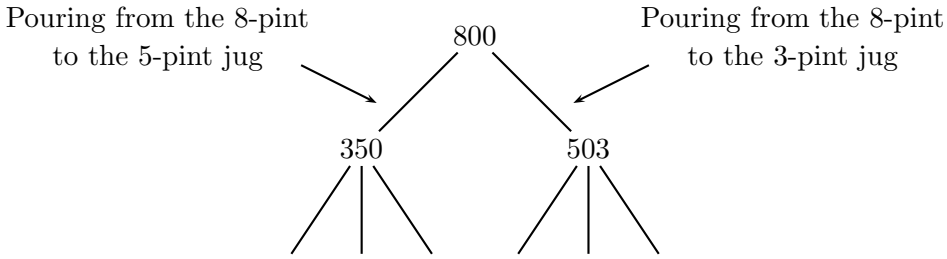
32. (Optional—state space search.) Given an 8-pint jug full of water, and two empty jugs of 5- and 3-pint capacity, get exactly 4 pints of water in one of the jugs by completely filling up and/or emptying jugs into others. Solve this problem using breadth-first search.

Answer: This is commonly known as the "three-jug problem" which can be naturally represented as a graph search/traversal problem. The problem search space is represented as a graph, which is constructed "on-the-fly" as each node is dequeued.

Each node represents a single problem "state", labelled as a string $j_8 j_5 j_3$ of digits:



The initial state of the problem is "800", which means there are 8 pints of water in the 8-pint jug, and no water in the others. A state of "353" denotes that there are 3 pints of water in the 8-pint jug, 5 pints of water in the 5-pint jug, and 3 pints of water in the 3-pint jug. For example, the top part of the search space graph is:



Note that only one action can be performed in a state transition, that is, water can be poured from only one jug between states.

Unlike other examples of graph construction seen so far in the subject, the entire graph of this search space need not be explicitly constructed. Rather, each new state-node is constructed and added to the traversal queue (from the configuration of the current state) as we go.

When each node is dequeued, it is checked to see if it contains a jug with 4 pints of water—that is our "goal" state.

Here is the "plain English" algorithm:

- Create a singleton queue with the initial state of 800.
- While the queue is not empty:
  - Dequeue the current state from the queue
  - Return the path to the current state if it contains a jug with 4 pints; halt
  - Mark the current state as visited.
  - For each state s that is possible from the current state:
    - Add s to the queue.
- Return False if the goal state was not reached.

Below we make this more precise by giving some pseudo-code. The first function finds all the possible ways we can extend a given state s.

```
function NextStates(s)
    next_states ← [ ]                                  ▷ Jugs is the set of jugs
    for j ∈ Jugs do
        room[j] ← capacity[j] − s[j]
    for (src, dest) ∈ Jugs^2 with src ≠ dest do
        pour_amount ← min(s[src], room[dest])
        if pour_amount > 0 then
            next ← a copy of s
            next[src] ← next[src] − pour_amount
            next[dest] ← next[dest] + pour_amount
            next_states ← next_states ∪ next
    return next_states
```

The next function will extract the full solution once a satisfying goal state has been found. We make use of a dictionary *prev* that maps a given state to the state from which it was reached. By keeping such a traversal record, we can obtain a path from the goal state to the initial state, thus giving us a sequence of pours to conduct in order to get our 4 pints of water.

```
function Solution(initial_state, goal_state, prev)
    path ← [ ]
    s ← goal_state
    while s ≠ initial_state do
        path ← s, path
        s ← prev[s]
    return initial_state, path
```

Here then is the breadth-first traversal. The parameters are the initial state and a predicate, *success*, which tests whether a given state is a successful goal state. (In our case, that means whether some jug in the state holds 4 pints.)

```
function BFS(initial_state, success)
    prev ← [ ]
    Inject(q, initial_state)
    while q is not empty do
        s ← Eject(q)
        if success(s) then
            return Solution(initial_state, s, prev)
        visited[s] ← True
        for next ∈ NextStates(s) do
            if visited[next] = False then
                prev[next] ← s
                Inject(q, next)
    return False
```

Here is a shortest solution path: 800, 350, 323, 620, 602, 152, 143.

33. (Optional—use of induction.) If we have a finite collection of (infinite) straight lines in the plane, those lines will split the plane into a number of (finite and/or infinite) regions. Two regions are *neighbours* iff they have an edge in common. Show that, for any number of lines, and no matter how they are placed, it is possible to colour all regions, using only two colours, so that no neighbours have the same colour.

Answer: We argue this inductively. It is clear that two colours suffice in the case of a single line. Now assume two colours suffice for n lines ($n \geq 1$). When we add line number $n+1$, we can modify the current colouring as follows. On one side of the new line, change the colour of each region. On the other side, do nothing. This gives a valid colouring. Namely, consider two arbitrary neighbouring regions. Either their common border is part of the new line, or it isn't. If it is, then the regions formed a single region before the new line was added, and since the colour of one side was reversed, the two parts now have different colours. If their border is not part of the new line, the two regions were on the same side of the new line. But then they must have different colours, as that was the case before the new line was added (they may have swapped their colours, but that's fine).