The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 2 MODULE, FULL YEAR, 2016-2017

ALGORITHMS CORRECTNESS AND EFFICIENCY

Time allowed TWO HOURS

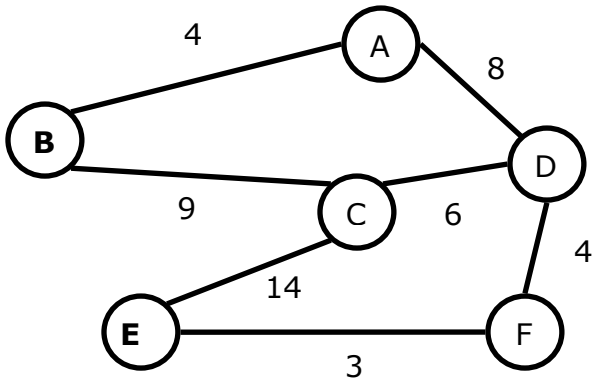
PARTIAL ANSWERS and HINTS
(NOTE THESE ARE NOT "MODEL ANSWERS"!!)

These may be updated. Please let us know if there are any problems.

1. This question concerns shortest paths in graphs. (20 marks total)

   (a) Explain, and give pseudo-code for, Dijkstra's algorithm to find the shortest path in an undirected graph when the distances for edges are given and all are non-negative. Your answer should include a brief explanation of why it always gives an optimal (shortest) path (if one exists).

   Then use Dijkstra's algorithm to find the shortest path from node **B** to node **E** in the graph below. Show your working, including the open and closed lists at each stage.



(10 marks)

ANSWER:
Code is standard from lectures, roughly

```
  PQ q
  q.add(start node)
  while ( ! q.isEmpty() ) {
    u ( c ) = pq.pop()  // remove first node with its cost
    close(u)
    if u == target node then return c.
    for each non-closed neighbour v of u
      pq.add( u with cost c + weight(uv) )  // or readjust cost if it is already there
 }
```

The method is complete and correct, assuming that nodes can be reached, because it works outwards in distance reachable. When nodes are popped from the PQ, then the distance to them is optimal, because other entries in the PQ will have larger distances to them, and because edges are non-negative none of their neighbours can get there faster.

Open:	Closed	
B(0)		
A(4), C(9)	B(0)	
C(9), D(12)	A(4)	
D(12),E(23)	C(9)	note D(15) is also achieved, but worse than D(12)
F(16),E(23)	D(12)	
E(16+3=19), ~~E(23)~~	F(16)	

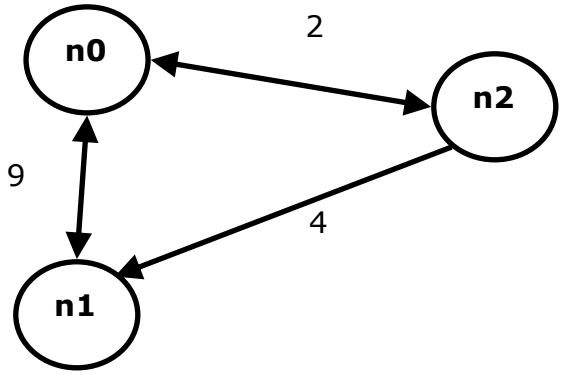
pop E reaches the goal in 19, with the path B-A-D-F-E

Note: vital not to take the first time that reach E, at E(23).

(b) Explain, and give pseudo-code for, the Floyd-Warshall algorithm to find the lengths of the shortest paths between each pair of vertices in a graph. The graph can have a mix of directed and bidirectional edges, but you can assume the distances are such that there are no negative cycles.

Your answer should include a brief explanation of why it always gives an optimal path for each pair of vertices.

Then use the algorithm on the graph below to find the matrix of all-pairs of shortest paths. Show your working.



(10 marks)

ANSWER:

HISTORIC NOTE: in the year (2016-17) the FW work assumed that self-loops are not included, and so the distances from nodes to themselves started as infinity. This convention was changed in subsequent years, including 2018-19, to be the convention that self-loops can be assumed, and so the diagonal elements would start with 0. (This is consistent with the assumption used in Dijkstra that all nodes can reach themselves in zero distance – the shortest path from a node to itself is zero.)

YOU SHOULD NOW ASSUME THAT ALL NODES HAVE A SELF-EDGE OF 0 – as in the lectures and tutorials. The computation below is using this current convention.

Algorithm is from lectures
Define a distance
d(i,j,k) = minimum from i to j using only the nodes 1…k
start from d(i, j,0) = w(i,j) the initial distances
recurrence
d(i,j,k+1) = min(d(i,j,k+1) , d(i,k,k) + d(k,j,k))

code is that

d(i, j,0) = w(i,j)
foreach k
foreach i

foreach j
d(i,j,k+1) = min(d(i,j,k+1) , d(i,k,k) + d(k,j,k))

This is correct because the recurrence relations are correct. A fundamental property of paths is that if we go from i to j via k, then we can use the optimal from i to k, and the optimal from k to j.

Using the ordering n0 n1 n2 for the rows and columns.
A value is the distance FROM the row label TO the column label.
E.g. the '4' is FROM n2 TO n1

d(i,j,0) = w(i,j) =

0	9	2
9	0	Inf
2	4	0

Now we allow paths to also to be via n0

d(i,j,1) = d(i,j,{n0}) = only the path from n1 to n2 gains; changing to min(Inf, 9+2)=11

0	9	2
9	0	11
2	4	0

d(i,j,2) = d(i,j,{n0,n1}) = nothing gains when n1 is also allowed as a via node

0	9	2
9	0	11
2	4	0

Finally, also allow n2, and so now all nodes are allowed as via node.

d(i,j,3) = d(i,j,{n0,n1,n2}) = now n0 to n1 gains, to become min(9,2+4) = 6

0	6	2
9	0	11
2	4	0

2. This question is concerned with string matching algorithms. (20 marks total)

Consider the algorithm `LastMatch` below, which returns the offset (shift) of the *last* occurrence of the pattern *P* in text *T*, or -1 if *P* does not occur in *T*:

```
LastMatch(T,P)
   for(s = T.length - P.length downto 0)
     j = 1
     while(j =< P.length and P[j] == T[s + j])
       j++
     if(j == P.length + 1)
       return s
   return -1
```

(a) State a suitable precondition for `LastMatch` in English, and a suitable postcondition in predicate calculus.

(6 marks)

ANSWER:

<precondition: *T* is an array of $n > 0$ characters

P is an array of $m > 0$ characters>

$\langle postcondition:\ s > -1 \leftrightarrow (\forall k(1 \leq k \leq m \rightarrow P[k] = T[s+k])$
$\wedge\ \forall s'\,(s < s' \leq n \rightarrow \neg\forall k\,(1 \leq k \leq m \rightarrow P[k] = T[s'+k])))\rangle$

(b) State suitable loop invariants for the **for** and **while** loops in LastMatch in predicate calculus. For the *outer* **for** loop invariant, explain how initialisation and maintenance of the invariant may be established.

(9 marks)

ANSWER:
A suitable invariant for the inner **while** loop is

$$\langle invariant : \forall k\,(1 \leq k < j \rightarrow P[k] = T[s + k])\rangle$$

that is, the characters in the subarray *P*[1 … j] match the corresponding characters in the subarray *T*[s ... s + j].

A suitable invariant for the outer **for** loop is

that is, for all shifts $s < s' \leq n$ there is no match.

To establish intialisation for the outer for loop invariant, we must show that the loop invariant holds before the first iteration of the loop, when *s* = *T*.length – P.length.

When s = *T*.length – *P*.length, there are no shifts *s′* such that $s < s' \leq n$ which can match, since the length of *T*[s′ ... *n*] < *m* (there are insufficient characters in the subarray to match the pattern). Since the right hand side of the conditional is true, the conditional is true, and the invariant is established.

To establish maintenance, we need to show that the code in the body of the outer for loop maintains the invariant of the outer loop.

Consider an iteration for some *s.* By the outer for loop invariant, for all shifts *s′* such that $s < s' \leq n$ there is no match. By the inner while loop invariant, the characters in subarray *P*[1 … j] match the corresponding characters in subarray *T*[s ... s + j], and this invariant holds when the inner while loop terminates. When the inner loop terminates, $j \leq P$.length + 1, and we evaluate the conditional `if(j == P.length + 1)`. We proceed by cases:

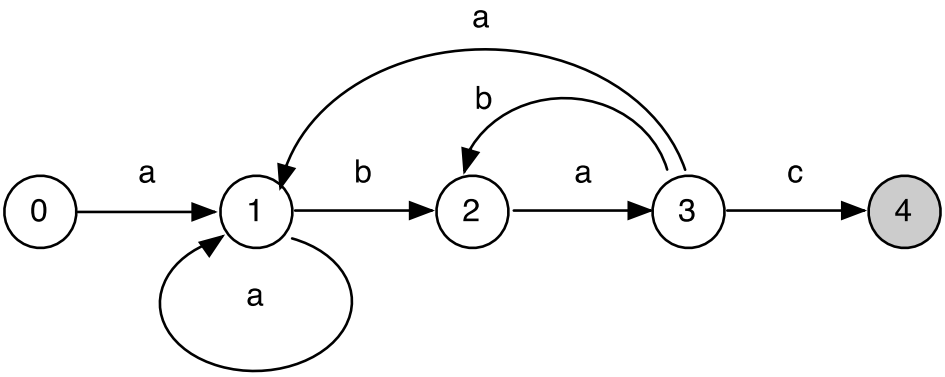
- in the case in which $j < P$.length + 1 (the conditional test is false), *P* does not match *T* at shift *s* (since not all characters in *P* were matched) so the outer for loop invariant is maintained;
- in the case in which *j* = P.length + 1 (the conditional test is true), *P* does match *T* at shift *s*, and the algorithm terminates (returning *s*), so again the outer for loop invariant is maintained.

[Note: the answers above are quite detailed – not all details are necessary to gain full marks for the question]

(c) In automata-based string matching, a string matching automaton is constructed that recognises the pattern. Draw the automaton that recognises the pattern P = abac, where the input alphabet is Σ = {a b c}. (Note: you don't need to show failing edges that return to state 0.)

(5 marks)

ANSWER:



3. This question is concerned with the 'big-Oh' family and recurrence relations. (20 marks total)

(a) Give the definitions of big-Oh, big-Omega, big-Theta, and little-oh by completing each of the following:

i) 'big-Oh': f(n) is O(g(n)) ….

ii) 'big-Omega': f(n) is Ω(g(n)) ….

iii) 'big-Theta': f(n) is Θ(g(n)) ….

iv) 'little-oh': f(n) is o(g(n)) ….

(5 marks)

ANSWER:

Standard. Expect the definitions from the lectures. That is, in abbreviated form

f(n) is O(g(n)) iff exists c, c>0 and n0 s.t forall n >= n0. f(n) <= c g(n)
f(n) is Ω (g(n)) iff exists c > 0 n0 s.t forall n >= n0. f(n) >= c g(n)
f(n) is Θ (g(n)) iff exists c' > 0, c'', n0 s.t forall n >= n0. c' g(n) <= f(n) <= c'' g(n)
f(n) is o (g(n)) iff forall c > 0 exists n0 s.t forall n >= n0. f(n) < c g(n)

This is standard Knowledge, but is asked because often students get these wrong – e.g. mixing up the ordering of quantifiers, or the inequalities.

(b) Give appropriate descriptions in terms of each of `O' (big-Oh) and `Ω' (big-Omega) of the function

$$f(n) = 2 n^2 + 5 n \log(n^4)$$

You should justify your answers, but do not need to give proofs. (5 marks)

ANSWER:

The second term is the smaller one as it is just n log n, and the log n is dominated by n.
The coefficient does not matter. Hence, it is equivalent to $f(n) = n^2$ Hence, is both O and Omega of n^2.
Note a common mistake is to think that Omega needs the smaller term.

(c) Consider the function

$$f(n) = 2 n + (n^2 \% 10)$$

where "n^2 % 10" uses the usual modulus operator, e.g. 7^2 % 10 = 9, etc.

From the definitions, prove or disprove that f(n) is O (n) (big-Oh)

(5 marks)

ANSWER:

We have (n^2 % 10) <= 10 by the definition of "%".
Hence, it is no worse than 2n + 10.
Hence, take c=3, and n0=10 (or larger).
Then we need to show

2 n + (n^2 % 10) <= 3 n for all n >= 10

(n^2 % 10) <= n for all n >= 10

which is true. Hence it is proved.

Get only partial marks if get right answer but do not do "From the definitions" and instead just drop the smaller term.

(d) Consider the recurrence relation

$$T(n) = 3\,T(n/2) \qquad \text{with} \quad T(1) = 1$$

Solve the relation exactly.
Hint:

Compute T(2), T(4), etc., and use this to find the answer for T(2^k).
Then prove your answer is correct using induction.

(5 marks)

ANSWER:

T(1) = 1
T(2) = 3
T(4) = 3*3 = 9
T(8) = 3^3 = 27
etc

Hence, can induce that T(2^k) = 3^k
Induction
Base: True at k=0. As gives T(2^0) = 3^0 so T(1) = 1.
Step: Assume true at k.
Then T(2^(k+1)) = 3 * T(2^(k+1) / 2) = 3 * T(2^k) = 3^k = 3^(k+1)
as required.

For T(n) k = log_2 n
T(n) = 3^(log_2(n)).

4. This question is concerned with MSTs in weighted undirected graphs. (20 marks total)

(a) Define a Minimum Spanning Tree for a weighted undirected graph.

(4 marks)

ANSWER:

Firstly it is a spanning tree. So it uses a subset of the edges to create a tree that connects together all the vertices. Then it is minimal in that when we consider the total weight of the edges used, then no other spanning tree has a smaller value.

Lose marks if path mentioned inappropriately, e.g. "MST is a path"

(b) Briefly describe Prim's algorithm for finding an MST.

(6 marks)

ANSWER:

Start from the MST being a single (arbitrary node).

repeat until all nodes are reached:
consider the set of all the edges from nodes in the current tree, to the other nodes, and take an arbitrary edge of the smallest weight in that set, and add it to spanning tree

(c) Give an argument that Prim's algorithm is correct, i.e. that it will always find an MST.
Hint: consider partitioning the nodes into two sets V1 and V2 and consider the edges between them.

(5 marks)

ANSWER:

Given a partition, consider the set of edges that go between V1 and V2. Suppose that an edge e has a minimum weight within that set, then that edge will be part of some MST. Proof is by contradiction, if it is not minimum then could replace it with a smaller edge from the set and so have a better spanning tree. Hence in Prims algorithm
V1 = MST so far
V2 = the other nodes
and this demonstrates it adding a minimum edge will still allow to be in an MST.

(d) Suppose that all the weights on the graph are different. I.e. that no two edges have the same weight. In this case argue that the MST is unique.
Hint: consider the partition used in part c.

(5 marks)

ANSWER:

Consider any partition V1 V2, then the minimal edge between the two partitions is unique as all the edges are different weights. But only the minimal edge is in the MST – otherwise we could reduce the weight by switching to a lower-weight edge. Hence if we consider all partitions the edge of the MST is forced. Hence there cannot be any freedom in constructing it.

Only partial marks for just pointing out that Prim's algorithm will have no arbitrary choices from a given starting node, but does not cover the issue of whether a different MST would result from a different starting node.

5. This question is concerned with sorting algorithms. (20 marks total)

(a) Explain, and give pseudo-code for, the merge-sort algorithm to sort an array A[] of integers into ascending order.

Show the working of the algorithm on the array A = [4 8 9 2 1 7 3 6]. Give the tree representing the execution of the algorithm.

State the worst case runtime of mergesort, of an array of n entries, in big-Oh notation, and give a justification of your answer.

(9 marks)

ANSWER:

The algorithm is standard

```
mergesort(A) {
    divide A into left L and right R arrays
    mergesort(L)
    mergesort(R)
    A = merge(L,R)
}
```

where the merge algorithms considers the two sorted arrays L and R, and then takes the smallest start entry of each.

The tree is:

[4 8 9 2 1 7 3 6].

[4 8 9 2 -> 2 4 8 9] [1 7 3 6 -> 1 3 6 7]

[4 8 -> 4 8] [9 2 -> 2 9] [1 7 -> 1 7] [3 6 -> 3 6]

The complexity is O(n log n).
Because at each level of the execution tree we need to do O(n) work.
The height of the tree is O(log n) because it is a well-balanced tree.

(b) Explain, and give pseudo-code for, the quicksort algorithm to sort an array A[] of integers into ascending order.

Both mergesort and quicksort are in the divide-and-conquer class of algorithms, but describe how they take a different approach to the 'divide' and to the 'conquer' steps.

(7 marks)

ANSWER:

```
qs( A ) {
    p = pivot(A)
```

```
    (L,R) = partition(A,p)
    qs(L)
    qs(R)
}
```

where the partition works on the array to ensure all elements < p are in the left side L, and all >= p are in the right side.

For the 'divide' : mergesort takes a trivial split, and quicksort has a specific partition step
For the 'conquer' : mergesort needs a non-trivial merge, but the conquer in quicksort requires no work, but is automatic.

(c) Suppose that the pivot is selected randomly from within the array. Briefly outline a proof that the resulting average case complexity is O(n log n)

(4 marks)

ANSWER:

Because 2/3 of the time it will be in the middle 1/3 of the range. So the execution tree will be of expected depth O(log n). Hence, similar argument to part (a).