Decision 1 (AQA)

1. Introductory Ideas 

Algorithm = set of instructions = flowchart = program

Should apply to general case and be as concise as possible.

- Graph
- Vertex/node (point)
- Edge/arc (line)
- Multiple arc
- Loop

Connected graph	Graph where can travel from any point to any other point somehow
Simple graph	No multiple arcs or loops
Complete graph	Simple graph where every node connected to every other node by exactly one arc
Network	Weighted graph
Degree/order of node	Number of arcs joined to node

- Complete simple graph with n nodes has $\frac{1}{2}n(n-1)$ arcs.

Graphs as matrices: adjacency matrix vs. distance/weighted matrix.

2. Minimum Spanning Trees 

Walk	Any journey around a graph/network
Trail / route	A walk which uses each arc only once (can use nodes multiple times, doesn't have to use all arcs)
Closed trail	A trail which begins and ends at same point
Path	A walk which uses each node only once (exception of first node, doesn't have to use all nodes)
Cycle	Closed path
Tree	Graph with no cycles
Connector / spanning tree	Tree connecting all nodes

- A spanning tree with n nodes will have $n-1$ arcs.

Kruskal's Algorithm (to find minimum spanning tree):
1. Choose arc of minimum weight.
2. Choose next arc of minimum weight *from any* (provided does not form cycle).
3. Repeat step 2 until all nodes connected.

Primm's Algorithm (to find minimum spanning tree):
1. Choose any node.
2. Choose arc of minimum weight *joined to connected nodes*.
3. Repeat step 2 until all nodes connected.

Primm's Algorithm with Matrices (to find minimum spanning tree):
1. Choose any node
2. Cross out row for chosen node. Circle and number column header of chosen node.
3. Circle minimum from available weights node column. Note the row header (node) for this chosen weight.
4. Repeat 2 and 3 until all nodes chosen.

3. Shortest Path

Dijkstra's Algorithm (to find minimum path between two given nodes):
1. Give start node permanent value of zero.
2. For all arcs leaving node: if '*previous temporary value + weight of new arc* < *temp min value*, write in or update, else leave existing.
3. Choose node with least minimum temporary value and make permanent
4. Repeat 2 and 3 until reach destination.

- Be able to use Dijkstra's with one way streets.

4. Route Inspection (Chinese Postman)

- Shortest trail *along all arcs* and return home.

Type of Graph	No. of Odd Nodes	Shortest Trail Distance
Eulerian / traversable	Connected graph with all nodes of even order	Total of weights
Semi-Eulerian / semi traversable	Connected graph with exactly 2 nodes of odd order	Total weight + shortest path between odd nodes
Non-Eulerian / cannot be traversed	Connected graph with >2 nodes of odd order	Total weight + min total of shortest paths between pairs of odd nodes

Each arc creates 2 degrees of order $\Rightarrow$ sum of degrees always even
$\Rightarrow$ exactly 1 odd node impossible
$\Rightarrow$ number of odd nodes always even

Where, to find additional arc weights:
1. Identify all nodes of odd order.
2. Identify all possible permutations of pairs of odd nodes
3. Find shortest paths between pairs of odd nodes and hence total shortest paths for each permutation of pairs.
4. Choose permutation with minimum total additional weight.

Numbers of possible pairings of odd nodes given by:

$$(n-1)\times(n-3)\times(n-5)\times\ldots\times 3\times 1$$

Where n = number of odd nodes.

5. The Travelling Salesman Problem

- Shortest path *visiting all nodes* and return home.

Hamiltonian Cycle	Closed cycle visiting each node *only once* (but not necessarily along each arc)
Tour	Visit every node *at least once* and return home (differs from closed path as could visit some nodes more than once)

- Graph with n vertices has $\frac{1}{2}(n-1)!$ possible tours.

Shortest route complete graphs are important here.

Nearest Neighbour Algorithm (to find a Hamiltonian Cycle):
(A heuristic algorithm: finds good quickly but not necessarily optimum solution)

1. Choose any node.
2. Choose and follow arc with minimum weight, which connects current node to another unvisited node.
3. If all nodes chosen, travel to starting node, else repeat step 2.

- Nearest Neighbour with matrices is similar to Prim's + return home.
- Questions usually involve attempting to minimize tour by trying different starting nodes. (In case of a Hamiltonian Graph starting point will not matter but for non-Hamiltonian changing the starting point may reduce size of tour.)

Upper & Lower Bounds for traveling salesman problem:

Highest lower bound Lowest upper bound.

- Upper bound is minimum value of all tours found using nearest neighbour with different starting nodes.
- To find lower bound:

1. Separate one node and all arcs joining this node from rest of graph.
2. Find length of minimum spanning tree for remaining graph.
3. Add weights of two shortest arcs from separated node to minimum connector from step 2. Note this down.
4. Repeat steps 2 & 3 for all nodes.
5. Lower bound is given by maximum value from steps 2, 3, 4.

- Lowest bound route and figure may not be a tour but instead represents the lower bound beyond which a tour is certainly not possible. Upper and lower bounds make more sense in context of larger graphs (such that the applications and algorithms in this Decision module are intended for).

6. Matching Graphs Sainsbury's

Bipartite Graph	Nodes in two sets, each arc joins a node from set A to set B. (No arcs joining nodes in set A to another in set A).
Complete Bipartite Graph	Every node in set A joins to every node in set B.
Matching	A pairing up where each node has maximum one arc (i.e. is order 1) using only arcs of a given bipartite graph.
Maximal Matching	A matching where no more arcs can be added.
Complete Matching	All nodes in set A matched up with all nodes in set B. $\Rightarrow$ no. of nodes same in each set. A complete matching is also a maximal matching.

- Bipartite graphs can be represented as matrices, which can often be compacted matrices where row headers represent set A and column headers represent set B.

- To improve a matching means to find a new matching entirely that consists of more arcs than the previous matching. It is not just adding to an existing matching.

Maximising Matching / Matching Improvement / Alternating Path Algorithm:

1. Find a (good) existing matching
2. Search for an alternating path (method 2). If none possible, stop.
    a. Turn the graph into a directed network with $\rightarrow$ on unmatched arcs and $\leftarrow$ on matched arcs.
    b. Choose an unmatched vertex on LHS and label as permanent 0.
    c. Apply Dijkstra's algorithm until either reach unmatched RHS vertex or no further labelling possible.
    d. If unmatched RHS vertex reached, trace path back, you have an alternate path.
3. Form new matching by swapping status of edges in alternating path then go to step 2.
4. If any unmatched LHS vertices remain go to step 2, else stop, matching is maximal.

7. Linear Programming

- To find best combination of variables to optimize outcome.

Decision / Control Variables	Quantities to be decided upon.
Constraint	Limitation of the given situation
Objective Function	The equation of the variable to be maximized / minimized, e.g. 'profit'.
Feasible Region	Area of graph where possible solutions lie.
Objective Line	Line on graph of objective function.

Typical process is to:
1. State the control variables.
2. Set the constraints.
3. State the objective function.
4. Formally state the problem (using algebraic inequalities).
5. Illustrate the solution graphically.
6. Draw on the objective line.
7. Move objective line to maximum / minimum extreme position of feasible region. If objective line parallel to edge of max / min point of graph, any solutions in feasible region and on this line are solutions.

8. Sorting

Four methods of varying efficiencies.

- To determine efficiency of a method count number of comparisons and number of swaps required.

Bubble Sort
1. Pass through list swapping pairs of numbers as necessary ($1^{st}/2^{nd}$, $2^{nd}/3^{rd}$, $3^{rd}/4^{th}$…).
2. Repeat passes leaving out final number from previous pass each time.
3. End when complete pass produces no swaps or when pass is of length 1.

Shuttle Sort
1. Pass through list swapping pairs of numbers as necessary ($1^{st}/2^{nd}$, $2^{nd}/3^{rd}$, $3^{rd}/4^{th}$…). Each time a swap occurs pass backwards, up the list, until no swap necessary.
2. Restart each downwards swap where you ended previous one.
3. End when pass involving final number in list is complete.

- Shuttle Sort has fewer comparisons than Bubble Sort but same number of swaps. For both methods greatest number of possible comparisons is $\frac{1}{2}n(n-1)$.

Shell Sort

1. Divide list into $INT\left(\frac{n}{2}\right)$ sublists (i.e. $\frac{n}{2}$ ignoring remainders) where lists are composed of positions $\left[1, 1+INT\left(\frac{n}{2}\right), 1+2\left(INT\left(\frac{n}{2}\right)\right)...\right]$ and $\left[2, 2+INT\left(\frac{n}{2}\right), 2+2\left(INT\left(\frac{n}{2}\right)\right)...\right]$ etc.
2. Shuttle sort each sublist and then re-compile main list.
3. Repeat with n = answer from previous $INT\left(\frac{n}{2}\right)$ until this = 1, i.e. a single list.

Quicksort Algorithm:

1. Make first number the pivot.
2. Compare other numbers in list with the pivot. If $\leq$ pivot place in list A (on LHS of pivot), if $>$ pivot place in list B (on RHS of pivot).
3. Repeat until all sublists of length 1.

- Quicksort is most efficient of the four methods.

9. Algorithms

An algorithm should provide an unambiguous next step at each stage and lead to a solution in a finite number of steps.

An algorithm can be written in words, written in pseudo-code ("repeat until" etc), written in formal programming language or drawn as a flowchart.

- To *trace an algorithm* is to follow it through for a test case writing notes and changes of variables for each step in a *trace table*.

- The *efficiency* of an algorithm is given by the number of steps taken to find a solution.

- The *order* of an algorithm is a measure of how quickly the number of steps increases as the size of the problem increases:

Linear order*	n, $n-1$, $3n$ etc
Quadratic order*	$\frac{1}{2}n(n+1)$, n^2+2n+1 etc
Exponential order	2^n, $n!$ etc

* Linear and quadratic order are both examples of polynomial time which increase fairly slowly whereas exponential order (exponential time) increases very rapidly and are unusable except for small scale problems. It is for this reason that a general method for solving any TSP remains a mystery.