Question 1.

1. Input: weighted (non negative) undirected graph G, and vertex V part of G  
   Output: label Distance[m] where all m are part of G (distance to other vertices in G starting at source S) as a distance between V and m   
     
   Priority queue (using fib heap) Q  
     
   Algorithm:   
     
     
   for (int m = 0; m < G.V(); m++)  
    Distance[m] = Double.POSITIVE\_INFINITY;  
    Q.add(m, Distance[m]);  
     
   Distance[S] = 0 // distance from source to source is 0  
   Q.change(S, distance[S]) // update the key associated with the source  
     
   while(!Q.isEmpty()) {  
    v = Q.removeMin(); // in removeMin don’t forget to sink the heap  
    for (Edge e : G.adjacent(v)) {  
    w = e.to(); // destination of the edge  
    if(distance[v] + e.weight() < D[w]) {   
    distance[w] = distance[v] + e.weight();  
    Q.change(w, distance[w]); // swim the PQ with new distance  
    }  
    }  
   }  
   return D; // holds label of all vertices with distances to V  
     
     
   To generalise the approach to work on all the nodes, we simply put the above code in a loop that loops from int v = 0 to G.V()
2. Dijkstra’s algorithm is heavily influenced by the type of priority queue we are using.   
   In general it is O(E.Tdk + V.Tem) where Tdk is the complexity of decrease key in the priority queue, and Tem is the cost to extract minimum from the priority queue.   
     
   If we assumed the priority queue implemented an efficient fibonacci heap, the complexity of Dijkstra is O(E + V.log(V)).  
     
   Based on that Dijkstra is useful when we have sparse graphs (high number of vertices)  
     
   To run dijkstra between all nodes that is O(V.Edk + V2.Tem)
3. Distance to A from A is 0   
   First initialise the distance to all nodes to be positive infinity.  
   PQ: (A, 0), (G, infinity), (D, infinity), (B, infinity), (C, infinity), (F, infinity), (E, infinity)   
     
   Now the loop:  
   - Extract (A, 0):   
   Check and update distance array, and the distance of each vertex that are connected by edges to A:  
   PQ: (B, 10), (D, 12), (G, 18), (C, infinity), (F, infinity), (E, infinity)  
   - Extract (B, 10):  
   Check and update edges in PQ  
   PQ: (D, 12), (G, 18), (C, 22), (F, infinity), (E, infinity)  
   - Extract (D, 12)  
   Check and update PQ  
   PQ: (G, 18), (C, 22), (F, 28), (E, infinity)  
   - Extract (G, 18)  
   Check and update PQ  
   PQ: (C, 22), (F, 28), (E, infinity)  
   - Extract (C, 22)  
   Check and update PQ  
   PQ: (F, 28), (E, 28)  
   - Extract (F, 28)  
   Check and update pq   
   PQ: (E, 28)   
   - extract(E, 28)  
   Check and update distances  
     
   The shortest distance between A and E is 28
4. Picking heuristic that takes the shortest path that doesn’t exceed 12.   
   For every iteration we subtract distance we travelled and try to travel other distances with remaining fuel.  
     
   The heuristic would act like this:  
     
   Edges from A: G - 18 (refuse to take itover 12), D - 12 and B - 10  
   Takes B10, still have 2 steps in the fuel, D14 and C12 don’t permit us to use the 2. Sleep, next day we have 12 steps we can only take C. we go to C we have no more fuel we sleep. At C we have F8 and E6, we choose E6. we have reached the destination we stop.

Question 2.

1. Knapsack problem is when we have a knapsack with capacity W.  
   And have set of items, each item with value and weight. We want to fit items that give us the highest value in the knapsack without exceeding the capacity.  
   0/1 knapsack means we have to fit the item fully in, and not accept partial solutions.  
     
   In branch and bound algorithm, we would explore what potentially would give us the best upper bound and current solution whilst keeping weights used less than capacity.   
     
   Pseudocode:  
     
   Input: knapsack capacity, items value, items weight,   
   Output: vector of items to put in the knapsack.   
     
   - start by creating a solution structure, parent that has 2 childrens.   
   - start with empty solution vector.  
   - compute value and upper bound of parent.  
   - set current best to parent’s value.  
   - store parent in PQ  
   while (!PQ.isEmpty() && parent.upperBound() > currentSolution) {  
    Parent = removeMax();  
    child1.solutionVector = parent.solutionVector + ‘1’;   
    child2.solutionVector = parent.solutionVector + ‘0’;  
    Computer value and upper bound for both children.  
    For each child C   
    if(C is feasible) {  
    if(C > currentSolution) {  
    currentSolution = C.solutionVector;  
    upperBound = C.upperBound;  
    }  
    PQ.add(C);  
    }  
   }  
     
   Return currentSolution;  
     
     
   Sample output would be: 111011, which indicates to put all items except for 4th in the knapsack.
2. Instance X:  
   W = 500, n = 1000  
   DP will run in time and space complexity of O(Wn).   
   BNB will have to sort items by value to weight ratios, since they are very similar BNB will act like enumeration leading to worst case of O(2n)  
     
   Instance Y:  
   W = 100, n = 200   
   BNB would fair better since the value to weight ratio is varying a lot, so when we sort it and then do BNB ti won’t branch that often and we will reach our destination really quick.  
   DP could also be used, modern computers (anything made after 1990) are able to handle this and more, those numbers are too low to make a significant comparison between DP and BNB.

Question 3.

1. 1. Insert 2:  
      5\* 2 + 9 mod 13 = 19 mod 13 = 6   
      Insert 17:   
      5 \* 17 + 9 mod 13 = 3  
      Insert 24:  
      5 \* 24 + 9 mod 13 = 12  
      Insert 8:  
      5 \* 8 + 9 mod 13 = 10  
      Insert 11:  
      5 \* 11 + 9 mod 13 = 12, 12 is already being used, linear probe it to 0  
      Insert 4:  
      5 \* 4 + 9 mod 13 = 3, already being used by 17, linear probe it to 4

Insert 6:  
 5 \* 6 + 9 mod 13 = 0, already being used, linear probe it to 1  
 Insert 19:  
 5 \* 19 + 9 mod 13 = 0, already being used, linear probe it to 1, already being used, linear probe it to 2

|  |
| --- |
| 11 |
| 6 |
| 19 |
| 17 |
| 4 |
|  |
| 2 |
|  |
|  |
|  |
| 8 |
|  |
| 24 |

* 1. Double hashing:  
     h(i, k) = h1(k) + i \* h2(k)   
     h1(k) = 5X + 9 mod 13  
     h2(k) = 5 - x mod 3   
      **Insert 2:**   
     h(0, 2) = h1(2)   
     5\* 2 + 9 mod 13 = 19 mod 13 = 6   
     **Insert 17:**   
     h(0, 17) = h1(17)  
     5 \* 17 + 9 mod 13 = 3  
     I**nsert 24:**  
     h(0, 24) = h1(24)  
     5 \* 24 + 9 mod 13 = 12  
     **Insert 8:**  
     h(0, 8) = h1(8)  
     5 \* 8 + 9 mod 13 = 10  
     **Insert 11**:  
     h(0, 11) = h1(11)   
     5 \* 11 + 9 mod 13 = 12, collusion. I++  
     h(1, 11) = h1(11) + h2(11)  
     h2(11) = 4   
     12 + 4 = 16 % 13 = 3, collusion i ++  
     h(2, 11) = h1(11) + 2\*h2(11) = 12 + 8 = 20 % 13 = 7  
     **Insert 4:**  
     h(0, 4):  
     h1(4) = 5 \* 4 + 9 mod 13 = 3, already being used by 17, increment i  
     h(1, 4) = h1(4) + h2(4)  
     h2(4) = 5 - 4 mod 5 = 1   
     Hash = 1 + 3 = 4   
     **Insert 6:**  
     h(0, 6) = h1(6) = 0  
     **Insert 19:**h(0, 19) = h1(19)  
     5 \* 19 + 9 mod 13 = 0, already being used, increment i  
     h(1, 19) = h1(19) + h2(19)   
     h2(19) = 5 - 19 mod 5 = 1   
     0 + 1 = 1

|  |
| --- |
| 6 |
| 19 |
|  |
| 17 |
| 4 |
|  |
| 2 |
| 11 |
|  |
|  |
| 8 |
|  |
| 24 |

b)   
- attempt to hash the key(k).  
- index into the hashtable using the hash, if the item found, replace it with deactivated item.  
If the item isn’t found in the place it supposed to be: linear progress until one of those happen:  
If key is found: then you remove it and replace it with deactivated item.  
If hit an empty slot: then return NO\_SUCH\_KEY  
If hit a deactivated item, keep going.  
If came back to same hashed index; terminate and return NO\_SUCH\_KEY

Question 4

1. Depth first search is a searching traversal mechanism   
   Finite rooted binary tree:  
   Starting at the root, attempt to explore each child, as far as possible (until hit a leaf node) before back tracking and exploring other option.  
     
   Pseudo code:  
   DFS(TreeNode N, int key) {  
    If (N.key() == key)   
    Return N;  
    DFS(N.leftChild(), key);  
    DFS(N.rightChild(), Key);  
   }  
     
   For directed graph:  
   That means starting at source S, we follow each edge from S until it is fully explored before picking another edge to follow. While keeping track of what visited
2. DFS (Graph G, int V) {  
    Int counter = 0;  
    Int[] nodesNumber = new int[G.V()]();   
    Stack<Integer> s = new Stack<>();  
    s.push(V);   
    while(!s.isEmpty()) {  
    Int v = s.pop();  
    for(int w : G.adjacent(v))  
    s.push(w);  
    nodesNumber[v] = counter++;  
    }
3. Directed graph: all edges have a source and a target, unless we hit a biEdge it will always move in that direction so we won’t get stuck in a loop while we perform DFS.  
   Undirected graph: edges goes both ways, meaning that if previous algorithm performed it will be stuck in a loop going from A to B and B to A.   
   A way to solve it is to keep a marked array, which if the vertex is marked, then don’t put it in the stack.  
   DFS (Graph G, int V) {  
    Int counter = 0;  
    Boolean[] marked = new Boolean[G.V()]();  
    For (int i = 0; i < G.V(); i++) Marked[i] = false;   
    Int[] nodesNumber = new int[G.V()]();   
    Stack<Integer> s = new Stack<>();  
    s.push(V);   
    while(!s.isEmpty()) {  
    Int v = s.pop();  
    for(int w : G.adjacent(v)) {   
    if(!marked[w])  
    s.push(w);  
    }  
    nodesNumber[v] = counter++;  
    Marked[v] = true;  
    }
4. Example of one that have 2 colours:   
   Binary tree.  
   Example of one that doesn’t have 2 colours:  
   Triangle.

Algorithm:   
 DFS that requires each colour to be different from last one seen (each child has different colour from the parents)