1. Below is the Recursive Fibonacci Algorithm (RFA). An integer *n*, *n*≥1, is inputted into the algorithm for the algorithm to output the *n*th item of the Fibonacci series. Please answer the following questions.
(a) What is the 10th item of the Fibonacci series? (10%)
(b) Analyze the time complexity of RFA. (10%)

Algorithm RFA(*n*)
Input: integer *n*, *n*≥1
Output: the *n*th item of the Fibonacci series

```
1: if n=1 or n=2 then
2:    return 1
3: else
4:    a ← RFA(n-2)
5:    b ← RFA(n-1)
6:    return a+b
```

2. Assume it is a known fact that problem *X* is NP-complete.
(a) How can we prove that problem *Y* is also NP-hard by taking advantage of polynomial-time reduction and the fact? (10%)
(b) If we can prove *Y* is NP-hard and *Y* is NP, then we can prove *Y* is also NP-complete. How can we prove that *Y* is NP? (10%)

3. The procedure of solving many problems can be represented by trees. Thus, the solving of these problems becomes a tree searching problem. There are many tree searching algorithms: breadth-first search, depth-first search, hill climbing and best-first search. Below is the pseudo code of the breadth-first search algorithm. Please modify the pseudo-code to be the depth-first search algorithm (6%), the hill climbing algorithm (7%) and the best-first search algorithm (7%). You should write down the complete algorithm including the input, the output and all steps.

Algorithm: Breadth-First Search Algorithm
Input: the root node *r* of a tree to search for the goal node *g* **Output**: the path from *r* to *g* as the solution or NIL to indicate failure to find a solution
Step 1: Construct a one-element queue consisting of the root node *r* Step 2: Check if the first element in the queue is the goal node *g*. If so, return the path from *r* to *g* as the solution and stop. Step 3: Remove the first element from the queue. Add all descendants of the first element, if any, to the end of the queue one by one. Step 4: If the queue is empty, then return NIL and stop. Otherwise, go to Step 2.

國立中央大學資訊工程學系 105 學年度第二學期博士班資格考試題紙

科目： 演 算 法 (Algorithms)

4. Below is the famous Dijkstra Shortest Path Algorithm (DSPA). The input of DSPA is a weighted directed graph *G*=(*V*, *E*) and a specific source node *s*, where *V* is the node set and *E* is the edge set and every edge in *E* has a positive weight. (Note that the weight of the edge (*u*, *v*) is stored in *ew*[*u*, *v*]). For every node *u* in *V*-{*s*}, DSPA can derive the shortest path from *s* to *u*. However, DSPA does not consider node weights. If we now consider node weight and include node weights in the total weight of every path, we need to extend the original DSAP. Please extend DSAP to consider the node weights. Note that we assume the weight of a node *u* is stored in *nw*[*u*] and the weight of the destination node is not included in the total weight of the path. Note that DSPA uses indentation to represent the block structure. So please use indentation properly when writing down the extended DSPA algorithm. You should write down the complete algorithm including the input, the output and all steps. (20%)

```
Algorithm: DSPA (Dijkstra Shortest Path Algorithm)
Input: G=(V, E), ew, s //G =(V, E) is a graph with edge weights stored in ew, and s is the source
Output: SP //SP is the set of shortest paths from s to all other nodes
dist[s]←0; dist[u]←∞, for each u≠s, u∈V
insert u with key dist[u] into priority queue Q, for each u∈V
while (Q≠∅)
    u←Extract-Min(Q)
    for each v adjacent to u
        if dist[v] > dist[u]+ew[u,v] then
            dist[v]←dist[u]+ew[u,v]
            pred[v]←u   //v's predecessor in the shortest path is u
calculate the shortest path from s to u to add into set SP according to pred[u], for each u∈V, u≠s
return SP
```

5. Given a knapsack with capacity *C*, and *n* objects $o_1,\ldots,o_n$ with weights $w_1,\ldots,w_n$ and values $v_1,\ldots,v_n$, the 0/1 knapsack problem is to determine x_i (x_i = 0 or 1, $1 \le i \le n$) such that $\sum_{1\le i\le n} x_i w_i \le C$ and $V = \sum_{1\le i\le n} x_i v_i$ is maximized. Below is a dynamic programming algorithm to solve the 0/1 knapsack problem to output *V*. Please modify the algorithm to solve the subset sum problem, described as follows. Given a set *S* of *n* values $v_1, v_2, \ldots, v_n$ and a special value *C*, the subset sum problem is to determine whether or not there exists a subset *S'* of *S* such that $C = \sum_{v_i \in S'} v_i$. You should write down the complete algorithm including the input, the output and all steps. (20%)

```
Algorithm: 0/1 knapsack dynamic programming algorithm
Input: knapsack capacity C, n object weights w1,…,wn and n object values v1,…,vn
Output: the maximum value V of all objects that can be kept in the knapsack
for w ← 0 to C do
    v[0, w] ← 0
for i ← 1 to n do
    for w ← 0 to C do
        if wi ≤ w then
            v[i, w] ← max(v[i-1, w], vi + v[i-1, w-wi])
        else
            v[i, w] ← v[i-1, w]
V ← v[n, C]
return V
```