Design

Graph.c

Initialize Struct Graph.
    vertices
    undirected
    visited [vertices]
    matrix[vertices][vertices]
}

Graph_create (uint32_t vertices, undirected)
    Graph * G = calloc // Give memory
        set G->vertices = verticies
        G-> undirected = undirected
    for Loop so visited [i] = false
    Nested for loop so matrix[i][j] = 0

Graph_delete ( )
    Given in PDF

graph_vertices (Graph * G)
    just return G-> vertices

graph_add_edge (uint32_t i, j k)
    If G is Not NULL and i and j < vertices.
        If undirected = false
            G-> matrix [i][j] = k
        else
            matrix [j][i] = k

```
graph_has_edge (*G, i, j)
    If (G != NULL and i|j >=0 and <=26)
        return true
    else
        return False

graph_edge_weight (*G, i, j)

        return matrix[i][j]
    else
        return False

bool graph_visited (*G, v)
    if visited [v]
        return true
    else
        false

void graph_mark_visited (*G, v)
    if in bounds
        G->visited [v] is true

void graph_make_visited (*G, v)

        but G->visited = false
```

Stack.c

Initialize struct

// Given in PDF

Stack *stack_create(capacity)

// Given in PDF

Void Stack_delete()

// Given in PDF

bool Stack_empty(stack *s)
if top of stack = 0
true

bool Stack_full(stack *s)
if top = capacity
true

uint32_t stack_size(Stack *s)

return s -> top

bool Stack_push (Stack *s, *x)

if stack_full (s) = false

return false

Increment top by 1, after setting top = x

s->top = x

bool Stack_pop (Stack *s, *x)

if stack_empty (s)

return false

Decrement top by 1

*x = s->items[s->top] // Given

uint32t pop = 0

stack_pop (stack, &pop)

stack_peek

*x = s->items[s->top - 1]

void Stack_copy (Stack *dst, Stack *src)

Check if dst->capacity == src->capacity

// loop from 0 to src->top

copy each elements from src to dst.

// out of loop

dst->top = src->top.

Path.C

```
Path * path_create(void)
    *p = (Path *) malloc(sizeof(Path))
    p->vertices = // call stack_create(VERTICES)
    p->length = 0
    return p

void_Path_delete(Path **p)
    // stack delete (&(*p)->vertices)
    // free *p
    // *p = NULL

bool path_push_vertex(Path *p, uint32 v, Graph *G)
    if length of path = 0
        stack push v to p->vertices
        p->length = Graph edge weight (start_Vertex, v)
    else
        peek() to get vertex at top
        push v on to p->vertices
        if push = false
            return false
        p->length += graph_edge_weight( )
```

if (!p)
return NULL

```
Bool  Path_pop_Vertex(Path *p, uint32 *v, Graph *G)
        if p->length = 0
            return false // nothing to pop

        stack_pop from p->vertices to v
uint32 prev  ->  stack_peek from p->vertices on to &prev
        p->length -= graph_edge_weight( , v )
        return true.

path vertices
    return # of vertices in path
        stack_size

path_length(Path *p)
    return p->length

path_copy(dsc, src)
    stack_copy(dsc, src)
        dst->length = src->length
```

tsp.c

check if your shortest path length is 0
or curr path length of shortest

```
if curr path_vertices +1 = graph_vertices(G)
    edge_from(v, start_vertex)
        path_push_vertex(curr, START_vertex, G)
        path_copy(shortest, curr)
```

```
else
    for i in 0 -> graph_vertices(G)
        if there is from v to i & i
        has not been visited:
            push i on path
            mark i visited
            DFS with (i as v)
            mark i unvisited
            pop i from path.
```