Xi Liu, xl3504, Homework 8

Problem 1
instead of picking the interval with the earliest finish point among the remaining ones, choose the interval with the latest starting time that does not conflict with all the activities that are already selected. picking the latest starting point is a greedy choice because after the choice, there are maximum number of remaining available activities. the earliest finish point (if time proceeds in the normal direction) is equal to the latest starting point (if time begins in the reverse direction). since it is proven that taking the interval with the minimum finishing point among the remaining intervals produce the non-overlapping optimal subset of intervals, then for the same reason picking the interval with the latest starting time gives the optimal solution.

```c
typedef struct activity
{
    int start,
    finish;
}activity;

activity * activity_sel(int * start, int * finish,
int n, int * ret_sz)
{
    activity * ret = malloc(n * sizeof(activity));
    *ret = *a;
    int f_i = 0;
    for(int a_i = 1; a_i < n; ++a_i)
    {
        if(start[a_i] >= finish[f_i])
        {
            ret[(*ret_sz)++] = a[a_i];
            f_i = a_i;
        }
    }
    return ret;
}
```

Problem 2

bfs() is an algorithm that runs breadth first search on G using its adjacency matrix

```
#include <stdio.h>
#include <string.h>
#include <vector>
using namespace std;

vector<vector<int>> adj;

void bfs(int src_id)
{
    bool visited[adj.size()];
    memset(visited, false, sizeof(visited));

    vector<int> queue;
    visited[src_id] = true;
    queue.push_back(src_id);

    int cur_id;
    while(!queue.empty())
    {
        cur_id = queue.front();
        printf("%d ", cur_id);
        queue.erase(queue.begin());
        for(int i = 0; i < adj[cur_id].size(); ++i)
        {
            if(adj[cur_id][i] && !visited[i])
            {
                queue.push_back(i);
                visited[i] = true;
            }
        }
    }
}
```

need to traverse every element of the $|V| \times |V|$ matrix, so time complexity is $\Theta(|V|^2)$
only an auxiliary array of size $|V|$ is used in the bfs() algorithm, so space complexity of bfs() is $\Theta(|V|)$

Problem 3

check_connect() is an algorithm that checks whether graph g is connected. time complexity of check_connect() is $O(|V|+|E|)$ since check_connect() only calls the bfs(g, 0, &visited) function once with the source node that have a node_id of 0. since bfs() on a graph given by an array of adjacency list has a time complexity of $O(|V|+|E|)$, and the array traversal after the call to bfs() is $O(|V|)$, so the overall time complexity of check_connect() is $O(|V|+|E|)$

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <list>
using namespace std;

/* dist_src: distance from the source s to this node */
struct node
{
    int node_id;
};

/* adj: pointer to an array containing adjacency lists */
struct graph
{
    int num_node;
    list<node *> * adj;
};

node * malloc_node(int node_id)
{
    node * n = (node *)malloc(sizeof(node));
    n->node_id = node_id;
    return n;
}

graph * malloc_graph(int num_node)
{
    graph * g = (graph *)malloc(sizeof(graph));
    list<node *> * adj = new list<node *>[num_node];
```

```
    g->num_node = num_node;
    g->adj = adj;
    return g;
}

void add_edge(graph * g, int node_id, node * n)
{
    g->adj[node_id].push_back(n);
}

void bfs(graph * g, int src_id, bool ** visited)
{
    *visited = (bool *)malloc(g->num_node * sizeof(node));
    memset(*visited, false, g->num_node * sizeof(node));

    list<int> queue;
    (*visited)[src_id] = true;
    queue.push_back(src_id);

    list<node *>::iterator i;
    while(!queue.empty())
    {
        int cur_id = queue.front();
        printf("%d ", cur_id);
        queue.pop_front();
        for(i = g->adj[cur_id].begin(); i != g->adj[cur_id].end(); ++i)
        {
            if(!(*visited)[(*i)->node_id])
            {
                queue.push_back((*i)->node_id);
                (*visited)[(*i)->node_id] = true;
            }
        }
    }
}

bool check_connect(graph * g)
{
```

```
    bool * visited;
    bfs(g, 0, &visited);
    for(int i = 0; i < g->num_node; ++i)
        if(!visited[i])
            return false;
    return true;
}
```

Problem 4

check() is an $O(n)$ algorithm that checks if there is any vertex in G that has edges coming to it from all other vertices of G but no edges going out from it, call a vertex that satisfies this condition a sink. for a vertex $V[i][j]$ to be a sink, $V[i][j]$ must be 0 since there should not be cycles. within column j, only $V[i][i]$ can be 0 since the sink vertex must have all edges coming to it from all other vertices. time complexity of check() is $O(n)$ since $while(i < V.size()\ \&\&\ j < V.size())$ has at most $O(2|V|)$ number of iterations and constant time cost per iteration, and the loop $for(int\ j = 0; j < V.size(); ++j)$ in sink() has at most $O(|V|)$ number of iterations, so the overall time complexity of check() is $O(n)$

```
bool sink(int i)
{
    for(int j = 0; j < V.size(); ++j)
    {
        if(V[i][j]) /* vertex has an outgoing edge*/
            return false;
        if(!V[j][i] && j != i) /* within column j,
        only V[i][i] can be 0 */
            return false;
    }
    return true;
}

bool check(int * idx)
{
    int i = 0, j = 0;
    while(i < V.size() && j < V.size())
    {
        if(V[i][j])
            ++i;
        else
            ++j;
    }
    if(i > V.size())
        return false;
    else if(!sink(i))
```

```
        return false;
    *idx = i;
    return true;
}
```

Problem 5

(a)

the boundary of the union of P and Q is the union of the left boundary of the left convex polygon, right boundary of the right convex polygon, upper tangent that connects P and Q, and lower tangent that connects P and Q

below are the steps to merge the convex polygons P and Q, which will be called merge(P, Q)

let a be the point in P with the maximum x-coordinate, b be the point in Q with the minimum x-coordinate

let a_up be a copy of a, b_up be a copy of b, a_low be a copy of a, b_low be a copy of b

to find the upper tangent:

keep a_up unchanging for now, move the point b_up in a clockwise direction while the angle formed by points a_up b_up, and clockwise neighbor of b_up makes a counterclockwise turn (left turn); since if the angle formed by points a_up, b_up, and clockwise neighbor of b_up makes a clockwise turn (right turn), then b_up cannot go up any more

keep b_up unchanging for now, move the point a_up in a counterclockwise direction while the angle formed by points b_up a_up, and counterclockwise neighbor of a_up makes a clockwise turn (right turn); since if the angle formed by points b_up, a_up, and counterclockwise neighbor of a_up makes a counterclockwise turn (left turn), then a_up cannot go up any more

now, the segment a_up to b_up is the upper tangent

to find the lower tangent:

keep a_low unchanging for now, move the point b_low in a counterclockwise direction while the angle formed by points a_low b_low, and counterclockwise neighbor of b_low makes a clockwise turn (right turn); since if the angle formed by points a_low, b_low, and counterclockwise neighbor of b_low makes a counterclockwise turn (left turn), then b_low cannot go down any more

keep b_low unchanging for now, move the point a_low in a clockwise direction while the angle formed by points b_low a_low, and clockwise neighbor of a_low makes a counterclockwise turn (left turn); since if the angle formed by points b_low, a_low, and clockwise neighbor of a_low makes a clockwise turn (right turn), then a_low cannot go down any more

now, the segment a_low to b_low is the lower tangent

(b)
find_convex_hull() is a divide and conquer algorithm that finds the convex hull of n points. find_convex_hull() calls the merge() function defined in Problem 5 (a). find_convex_hull() has a time complexity $T(n)$ of $O(n \lg n)$ since each call to merge() takes $O(n)$ times, and there are $O(\log_2 n)$ calls to merge() since problem size is halved each time (if max depth of the recursion tree is i, then $n/2^i = 1, \quad n = 2^i, \quad \log_2 n = i$).
$T(n)$ = number of calls to merge() $\cdot$ cost per merge() $= O(\lg n) \cdot O(n) = O(n \lg n)$

```
find_convex_hull(vector<point> points)
{
    if(points.size() == 1)
        return points
    left_convex_hull = points[1, 2, ..., points.size() / 2]
    right_convex_hull = points[points.size() / 2,
    points.size() / 2 + 1, ..., points.size()]
    return merge(left_convex_hull, right_convex_hull)
}
```