Xi Liu, xl3504, Homework 7

Problem 1
(a)
make_change() is an algorithm that makes change for n cents using least number of coins. first, it uses qsort() to sort the choices[] array in increasing order, then traverse the choices[] array from back to front (from i = num_choices - 1 to i >= 0), select choices[i] each time when the remaining value is greater or equal to choices[i] since choices[i] will be the choice of the maximum coin available at this point, since maximum coins are always chosen each time, least number of coins will be used to make the change

```
#include <stdio.h>
#include <stdlib.h>

void print(int * a, int sz)
{
    for(int i = 0; i < sz; ++i)
        printf("%d ", a[i]);
    printf("\n");
}

int cmp(const void * p1, const void * p2)
{
    int * a1 = (int *)p1, * a2 = (int *)p2;
    return *a1 - *a2;
}

int choices[] = {1, 5, 10, 25};
int num_choices = sizeof(choices) / sizeof(*choices);

int make_change(int n, int ** a)
{
    *a = malloc(n * sizeof(int));
    int ret_sz = 0;
    qsort(choices, sizeof(int), num_choices, cmp);
    for(int i = num_choices - 1; i >= 0; --i)
    {
        while(n >= choices[i])
        {
            n -= choices[i];
            (*a)[ret_sz++] = choices[i];
        }
    }
    return ret_sz;
}

int main()
{
    int n = 91;
    int * a;
    int ret_sz = make_change(n, &a);
    printf("make_change(%d) = %d\n", n, ret_sz);
    printf("changes:\n");
    print(a, ret_sz);
}
```

(b)
let n' be the remaining value to be changed, n' is initially set to be n, and n' will be updated to be $n' - \mathit{choices}[i]$ if a coin with the value of $\mathit{choices}[i]$ is chosen to be in the result set of coin change

if $n' \geq 25$, then the greedy algorithm will choose a coin with value of 25. {25} is the optimal choice at this point, since if {25} is not chosen, then the second-smallest-size set of coins to produce the same value is {10, 10, 5}, but $|\{25\}| = 1 < \{10, 10, 5\} = 3$, so for this choice, the greedy algorithm appends the minimum-size set of coin change to the result set of coin change

if $25 > n' \geq 10$, then the greedy algorithm will choose a coin with value of 10. {10} is the optimal choice at this point, since if {10} is not chosen, then the second-smallest-size set of coins to produce the same value is {5, 5}, but $|\{10\}| = 1 < \{5, 5\} = 2$, so for this choice, the greedy algorithm appends the minimum-size set of coin change to the result set of coin change

if $10 > n' \geq 5$, then the greedy algorithm will choose a coin with value of 5. {5} is the optimal choice at this point, since if {5} is not chosen, then the second-smallest-size set of coins to produce the same value is {1, 1, 1, 1, 1}, but $|\{5\}| = 1 < \{1, 1, 1, 1, 1\} = 5$, so for this choice, the greedy algorithm appends the minimum-size set of coin change to the result set of coin change

(c)
if the set of coin denominations is {5, 4, 1} and $n = 8$, then since the greedy algorithm always choose the maximum possible coin available, the greedy algorithm will produce a result set of coin choices as {5, 1, 1, 1} that uses 4 coins, but the optimal solution is {4, 4} which uses 2 coins

Problem 2

(a)

```
bool can_reach(int * A, int n)
{
    int leftmost_i = n;
    for(int i = n; i >= 1; --i)
    {
        if(leftmost_i <= i)
        {
            leftmost_i = min(leftmost_i, i - A[i]);
            if(leftmost_i <= 1)
                return true;
        }
    }
    return false;
}
```

(b)

call a position i reachable if and only if the frog can reach position at index i from position at the nth index through a series of jumps

let $(i,\ j) \in \mathbb{N}^2,\ 1 \leq i \leq n,\ 1 \leq j \leq n.$ for all reachable positions i, it will make positions $i - j$ (such that $1 \leq j \leq A[i]$) to also be reachable

so, traverse every position of the array A from the last index to the first index, and update the variable leftmost_i as the index of the current leftmost or farthest (dependent on the currently visiting position) position reachable from the last index of the array A

for a currently visiting position i, if $leftmost\_i \leq i$, it means that position i is within the range of the leftmost reachable position, so the frog can start from the last index to reach position i through a series of jumps, so the program can update $leftmost\_i$ by $i - A[i]$ (if $i - A[i] < leftmost\_i$)

in the process of traversal, if $leftmost\_i \leq 1$, it means that the position at the first index is within the range of the leftmost reachable position, so the frog can start from the last index to reach the first index through a series of jumps, so the program can return true; otherwise, if the traversal finished but the if($leftmost\_i \leq 1$) test always evaluated to be false, then it means that the position at the first index is outside the range of the ultimate leftmost reachable position, so the frog cannot start from the last index to reach the first index through a series of jumps, so the program can return false

(c)
time complexity $T(n)$ of can_reach() is $\Theta(n)$, because the length of array A is n. in the worst case, can_reach() visits n position in the array, and perform constant time operations for each iteration of the for loop, so $T(n) = \Theta(n) \cdot \Theta(1) = \Theta(n)$

Problem 3

(a)

color_interval() is an algorithm that computes the number of maximum overlapping intervals, this number is equal to the minimum number of colors needed to color the set $\mathcal{I}$ so that overlapping intervals are given different colors

```
#include <stdlib.h>

int cmp(const void * p1, const void * p2)
{
    int * a1 = (int *)p1, * a2 = (int *)p2;
    return *a1 - *a2;
}

int color_interval(int * s, int * f, int n)
{
    qsort(s, n, sizeof(int), cmp);
    qsort(f, n, sizeof(int), cmp);
    int i = 1, j = 1,
    max_overlap = 0, overlap = 0;
    for(int i = 1, j = 1; i <= n && j <= n; )
    {
        if(s[i] < f[j])
        {
            ++overlap;
            if(max_overlap < overlap)
                max_overlap = overlap;
            ++i;
        }
        else
        {
            --overlap;
            ++j;
        }
    }
    return max_overlap;
}
```

(b)
$\forall (a,\ b) \in \mathbb{N}^2,\ 1 \le a \le n,\ 1 \le b \le n$, let $[s_a, f_a),\ [s_b, f_b)$ be 2 intervals in set $\mathcal{I}$ such that interval $[s_a, f_a)$ starts before interval $[s_b, f_b)$
for any pair of 2 overlapping activities, the color will be different, since the if condition within the for loop checks that $if(s[i] < f[j])$, the if test evaluates to true when $s_b < f_a$, in this case the interval $[s_a, f_a)$ overlaps with $[s_b, f_b)$, so the counter *overlap* can be incremented by 1

suppose that $s[i] < f[j]$, $s[i]$ is the smallest element or earliest time instant not yet handled, so the program need to handle $s[i]$ at the current for loop iteration. $s[i]$ has the smallest starting time not yet handled, and the smallest finishing time not yet handled is $f[j]$, and since the smallest starting time not yet handled is less than the smallest finishing time not yet handled, the interval that has the starting time of $s[i]$ must overlap with the interval that has a finishing time of $f[j]$, $s[i]$ represents that a interval just started at the time instant handled by the current for loop iteration, so it is correct to increment the counter *overlap* by 1

otherwise, if instead $f[j] \le s[i]$, $f[j]$ is the smallest element or earliest time instant not yet handled, so the program need to handle $f[j]$ at the current for loop iteration. $f[j]$ has the smallest finishing time not yet handled, and the smallest starting time not yet handled is $s[i]$, and since the smallest finishing time not yet handled is less than or equal to the smallest starting time not yet handled, the interval that has the finishing time of $f[j]$ must not overlap with the interval that has a starting time of $s[j]$, $f[j]$ represents that a interval just finished at the time instant handled by the current for loop iteration, so it is correct to decrement the counter *overlap* by 1

expected running time = time of qsort + time of for loop $= O(n \lg n) + O(n) = O(n \lg n)$

Problem 4
(a)
convex_hull() is an $O(n^2 \lg n)$ algorithm to compute the convex hull after receiving each point by running graham_scan() n times after receiving each point

```
#include <stdio.h>
#include <stdlib.h>

int block = 128;

typedef struct point
{ int x, y; } point;

point p0;

void swap(point * p, int i, int j)
{
    point t = p[i];
    p[i] = p[j];
    p[j] = t;
}

/* distance squared
= (sqrt{(x2 - x1)^2 + (y2 - y1)^2})^2 */
int dist_sq(point a, point b)
{
    return (b.x - a.x) * (b.x - a.x)
        + (b.y - a.y) * (b.y - a.y);
}

/* slope1 = (y2 - y1) / (x2 - x1)
slope2 = (y3 - y2) / (x3 - x2)
if(slope1 == slope2)
    collinear
else if(slope1 < slope2)
    slope1 - slope2 < 0
    orientation is counterclockwise
else if(slope1 > slope2)
    slope1 - slope2 > 0
    orientation is clockwise

slope1 - slope2 will have
the same sign with
(x2 - x1)(x3 - x2)(slope1 - slope2)
= (y2 - y1)(x3 - x2) - (y3 - y2)(x2 - x1)

orient() returns an integer indicating
the orientation of points (a, b, c):
    0, if a, b, and c are collinear
    a negative value if orientation is counterclockwise
    a positive value if orientation is clockwise */
int orient(point a, point b, point c)
{
    return (b.y - a.y) * (c.x - b.x)
        - (c.y - b.y) * (b.x - a.x);
}

/* compare distance to p0 */
int cmp(const void * a1, const void * a2)
{
    point * p1 = (point *)a1;
    point * p2 = (point *)a2;
    int ori;
    if(!(ori = orient(p0, *p1, *p2)))
        return dist_sq(p0, *p1) - dist_sq(p0, *p2);
    return ori;
}

point * graham_scan(point * p, int n, int * top)
{
    int y_min = (*p).y, min_i = 0;
    for(int i = 1; i < n; ++i)
    {
        int y = p[i].y;
        if((y < y_min)
        || (y == y_min && p[i].x < p[min_i].x))
        {
            y_min = p[i].y;
            min_i = i;
        }
    }
    swap(p, 0, min_i);
    p0 = *p;

    qsort(p + 1, n - 1, sizeof(point), cmp);
    int sz = 1;
    for(int i = 1; i < n; ++i)
    {
        while(i < n - 1 && !orient(p0, p[i], p[i + 1]))
            ++i;
        p[sz++] = p[i];
    }
    if(sz < 3) return 0;

    point * s = malloc(sz * sizeof(point));
    *top = 0;
    s[(*top)++] = *p;
    s[(*top)++] = p[1];
    s[(*top)++] = p[2];

    for(int i = 3; i < sz; ++i)
    {
        while(*top && orient(s[*top - 2], s[*top - 1], p[i]) >= 0)
            --(*top);
        s[(*top)++] = p[i];
    }
    return s;
}

void print_stack(point * s, int top)
{
    for(int i = top - 1; i >= 0; --i)
```

```
        printf("{%d, %d}\n", s[i].x, s[i].y);
    printf("\n");
}

point recv_point(point * input, int i)
{/* or can return a point from user input */
    return input[i];
}

void convex_hull(point * input, int n)
{
    int cur_sz = block;
    point * p = malloc(cur_sz * sizeof(point));
    for(int i = 0; i < n; ++i)
    {
        if(i >= cur_sz)
        {
            cur_sz += block;
            p = realloc(p, cur_sz * sizeof(point));
        }
        p[i] = recv_point(input, i); /* receive a point */
        int top = 0;
        point * s = graham_scan(p, i + 1, &top);
        printf("after receive point %d, convex hull is\n", i);
        print_stack(s, top);
    }
}
```

(b)
convex_hull_jarvis_march() is an $O(n^2)$ algorithm to compute the convex hull after receiving each point by running n times after receiving each point, convex_hull_jarvis_march() calls some functions defined in Problem 4 (a) convex_hull_jarvis_march()'s time complexity $T(n) = O(n^2)$ since to account for receiving only 1 point at a time, a point will be added to the array pointed to by pointer p within the while loop using the assignment "p[recv_i] = recv_point(p, recv_i);", and then using the "if(p[recv_i].x < p[left_i].x) left_i = recv_i;" statement to update the current leftmost point if the added point has a smaller x value than the x value of the current leftmost point. it is correct to do this since Jarvis's march algorithm maintains the most counterclockwise point. the modifications on Jarvis's march algorithm will not have an effect on the original asymptotic time complexity of Jarvis's march algorithm, which is $O(n^2)$

```
void print(point * s, int n)
{
    for(int i = 0; i < n; ++i)
        printf("{%d, %d}\n", s[i].x, s[i].y);
    printf("\n");
}

point * convex_hull_jarvis_march(point * p, int n, int * ret_sz)
{
    if(n < 3) return 0;
    point * ret = malloc(n * sizeof(point));
    *ret_sz = 0;

    int left_i = 0;

    int cur_p = left_i, c, recv_i = -1;
    while(1)
    {
        ++recv_i;
        p[recv_i] = recv_point(p, recv_i);
        if(p[recv_i].x < p[left_i].x)
            left_i = recv_i;
        ret[(*ret_sz)++] = p[cur_p];
        c = (cur_p + 1) % n;
        for(int i = 0; i < n; ++i)
            if(orient(p[cur_p], p[i], p[c]) < 0)
                c = i;
        cur_p = c;
        printf("after receive point {%d, %d}, convex hull is\n",
        ret[recv_i].x, ret[recv_i].y);
        print(ret, recv_i + 1);
        if(cur_p == left_i)
            break;
    }
    return ret;
}
```