CS 341 — Assignment 3

Kevin Carruthers (20463098) — section 002

Winter 2015

Question 1

Part A

My code is as follows:

```
#include <iostream>
#include <limits>
#include <vector>

#define INFINITY std::numeric_limits<int>::max()

#define RED 0
#define BLUE 1


struct Point {
    int x;
    int y;
    int color;
};

struct Dominance {
    int c;
    std::vector<Point> points;
};


Dominance dominanceCount(std::vector<Point> points) {
    std::vector<Point> sorted = std::vector<Point>();
    if (points.size() == 0) {
        sorted.push_back({INFINITY, INFINITY, BLUE});
        return {0, sorted};
```

```
    } else if (points.size() == 1) {
        sorted.push_back(points.at(0));
        sorted.push_back({INFINITY, INFINITY, BLUE});
        return {0, sorted};
    }

    std::vector<Point> leftPoints = std::vector<Point>();
    std::vector<Point> rightPoints = std::vector<Point>();
    int cutoff = (points.size() / 2) - 1;
    for (std::vector<Point>::size_type i = 0; i != points.size(); ++i) {
        if (i <= cutoff) {
            leftPoints.push_back(points.at(i));
        } else {
            rightPoints.push_back(points.at(i));
        }
    }

    Dominance left = dominanceCount(leftPoints);
    Dominance right = dominanceCount(rightPoints);

    int i = 0, j = 0;
    int n = 0;
    int c = 0;

    for (unsigned int k = 0; k < points.size(); ++k) {
        if (left.points.at(i).y <= right.points.at(j).y) {
            sorted.push_back(left.points.at(i));
            if (left.points.at(i).color == BLUE) {
                ++n;
            }
            ++i;
        } else {
            sorted.push_back(right.points.at(j));
            if (right.points.at(j).color == RED) {
                c += n;
            }
            ++j;
        }
    }
    sorted.push_back({INFINITY, INFINITY, BLUE});

    return {c + left.c + right.c, sorted};
}

int main(int argc, char *argv[]) {
    int n;
```

```
    std::vector<Point> points = std::vector<Point>();
    for (std::cin >> n; n > 0; --n) {
        Point p;

        std::cin >> p.x;
        std::cin >> p.y;
        std::cin >> p.color;

        points.push_back(p);
    }

    Dominance dominance = dominanceCount(points);

    std::cout << dominance.c << std::endl;

    return 0;
}
```

Part B

I generated my input by generating n points, for the below values of n, each with x- and y-values samples randomly from zero to $10,000$. The x-values were sorted before placing them in points. Each point was assigned a random color. The experimental results agree with the theoretical analysis: the runtime of the brute-force algorithm was always longer than the divide and conquer implementation and increased at a greater rate.

Table 1: Timing data for Dominance Counting algorithm

Num Points	DivideAndConquer	Brute Force
10	0.00240802764893	0.00443291664124
50	0.00234699249268	0.00448107719421
100	0.00280213356018	0.00535011291504
250	0.00423407554626	0.00845003128052
500	0.00554609298706	0.0161769390106
750	0.00629115104675	0.0271201133728
1000	0.0100090503693	0.0503468513489
2000	0.0147519111633	0.139923095703
5000	0.0295369625092	0.759775876999

Question 2

Part A

```
1. def placeMailboxes(houses[0..n-1], D):
```

```
       // sort list of houses
2.     sort(houses)

       // initialize our mailbox list with a mailbox at -infinity for code cleanliness, re
3.     mailboxes = [-infinity]
       // for each house
4.     for i = 0 to n - 1:
           // if the house is farther than D distance from the last mailbox
5.         if house[i] > mailboxes[-1] + D:
           // add a new mailbox D distance ahead of the house
6.             mailboxes.append(house[i] + D)

       // remove the mailbox at -infinity, return the rest
7.     return mailboxes[1:]
```

We can determine the complexity of this algorithm as

$$\begin{aligned} T(n) &= O(n \log n) + O(1) + O(n)O(1) \\ &= O(n \log n) + O(1) + O(n) \\ &= O(n \log n) \end{aligned}$$

The complexity of this algorithm is thus $O(n \log n)$.

Part B

If my algorithm is not correct, it must be the case that either: a) there exists some optimal solution with fewer mailboxes, or b) my solution does not cover a house in our list. We can determine that my algorithm covers each house by reading the psuedo-code: if a house is ever uncovered by a mailbox, we immediately add a mailbox within range of the house.

To prove this algorithm is optimal, it suffices to prove that no house is covered by two mailboxes, as if this were the case we would simply remove one of those mailboxes to have a more optimal solution. We can prove this by, again, reading the psuedo-code: if a house is ever covered by a mailbox, we do not add another mailbox within range. Since the algorithm is greedy, we never "go back" and add to the list of mailboxes at some point in the ditance; thus it must be the case case that if a mailbox is added only to the end of the list, and if this occurs only when it would not overlap with other mailboxes, we must have a optimal solution. Since this is the case (ie. we add a mailbox only on line 6 and only when it would not overlap due to line 5), we must have an optimal solution.

Since this algorithm is both optimal and does not miss any houses, it must be correct.

Part C

We call my algorithm with `houses = [1, 2, 4, -1, -2, -5, 11, 5, 8, -8, -10, 14, 12, 19, 16], D = 3`. The algorithm first sorts `houses` which changes it to `[-10, -8, -5, -2, -1, 1, 2, 4, 5, 8, 11, 12, 14, 16, 19]`. The algorithm then iterates as folows:

1. Checks if `house[0] = -10` is greater than `-infinity + D`. Since this is the case ($-10 > -\infty + 3$), it places a mailbox at $-10 + 3 = -7$.
2. Checks if `house[1] = -8` is greater than `-7 + D`. It is not, so execution continues.
3. Checks if `house[2] = -5` is greater than `-7 + D`. It is not, so execution continues.
4. Checks if `house[3] = -2` is greater than `-7 + D`. Since this is the case ($-2 > -7 + 3$), it places a mailbox at $-2 + 3 = 1$.
5. Checks if `house[4] = -1` is greater than `1 + D`. It is not, so execution continues.
6. Checks if `house[5] = 1` is greater than `1 + D`. It is not, so execution continues.
7. Checks if `house[6] = 2` is greater than `1 + D`. It is not, so execution continues.
8. Checks if `house[7] = 4` is greater than `1 + D`. It is not, so execution continues.
9. Checks if `house[8] = 5` is greater than `5 + D`. Since this is the case ($5 > 1 + 3$), it places a mailbox at $5 + 3 = 8$.
10. Checks if `house[9] = 8` is greater than `8 + D`. It is not, so execution continues.
11. Checks if `house[10] = 11` is greater than `8 + D`. It is not, so execution continues.
12. Checks if `house[11] = 12` is greater than `8 + D`. Since this is the case ($12 > 8+3$), it places a mailbox at $12 + 3 = 15$.
13. Checks if `house[12] = 14` is greater than `15 + D`. It is not, so execution continues.
14. Checks if `house[13] = 16` is greater than `15 + D`. It is not, so execution continues.
15. Checks if `house[14] = 19` is greater than `15 + D`. Since this is the case ($19 > 14 + 3$), it places a mailbox at $19 + 3 = 22$.

The function will then return a list of the following mailboxes: $[-7, 1, 8, 15, 22]$.

Question 3

Part A

We know that

$$\begin{aligned} T_i &= T_{i-1} - x_i d_i \\ &= T_{i-1} - \lfloor \frac{T_{i-1}}{d_{i-1}} \rfloor d_{i-1} \\ &\leq T_{i-1} - \left(\frac{T_{i-1}}{d_{i-1}} - 1 \right) d_{i-1} \\ &\leq T_{i-1} - T_{i-1} + d_{i-1} \\ &\leq d_{i-1} \end{aligned}$$

and since we have

$$x_i = \lfloor \left(\frac{T_i}{d_i} \right) \rfloor$$

we see that

$$\begin{aligned} x_i &= \lfloor \left(\frac{T_i}{d_i} \right) \rfloor \\ &\leq \lfloor \left(\frac{d_{i-1}}{d_i} \right) \rfloor \\ &\leq \frac{d_{i-1}}{d_i} - 1 \end{aligned}$$

which gives us

$$0 \leq x_i \leq \frac{d_{i-1}}{d_i} - 1$$

Part B

Given

$$0 \leq x_i^* \leq \frac{d_{i-1}}{d_i} - 1$$

we can say that since d_{j-1} is divisible by d_j for all j we have

$$\begin{aligned} 0 \leq x_i^* &\leq \frac{ad_i}{d_i} - 1 \\ &\leq x_i^* \leq a - 1 \\ &\leq x_i^* < a \end{aligned}$$

If x_i^* were larger than a, we could simply take a fewer coins of denomination d_{i-1} and 1 more of d_i recursivly until x_i^* is smaller than a. Since this is a clear optimization, but our solution was already optimal, it must be the case that $x_i^* < a$. Thus we see that our inequality is correct and so

$$0 \leq x_i^* \leq \frac{d_{i-1}}{d_i} - 1$$

Part C

Assume X and X^* are valid solutions and $X \neq X^*$. Since i is the highest index where x_i and x_i^* differ, it must be the case that the values are the same for all $i < j \leq n$. Thus we must have the same value for $T = \sum_{j=i+1}^{n} x_j d_j$. We also know, by parts (a) and (b), that the maximum amount x_i and x_i^* differ by must be less than d_{j-1} (since both are bounded by 0 and $d_{j-1} - 1$, the most they could differ would occur when one is 0 and the other is $d_{j-1} - 1$, ie. a difference of $d_{j-1} - 1$ which is less than d_{j-1}). Since this difference is less than d_{j-1}, it must be impossible to add some extra number of coins of value d_{j-1} to the lower value between x_i and x_i^* such that $T + x_i d_i = T + x_i^* d_i$. Since all coin denominations at indices below $j-1$ are of a greater value than d_{j-1}, this must hold true for them, as well. Thus, we cannot get the correct T value for both X and X^* which is a contradiction (ie. one of them must not be a valid solution). Thus X must equal X^*.

Question 4

Part A

We can use divide and conquer: by splitting our array into two at each level of recursion, we can have $\log n$ iterations. If, at the bottom level, we return the item if it is less than s_j and merge by simply taking the larger item returned by our recursive calls (n operations), we will have an overal complexity of $O(n \log n)$.

Part B

To solve for $P(j)$, we see that we have two options: either p_j is a part of the solution and we can recursively add the profits of all segments which do not overlap with I_j (ie. $P\big(last(j)\big)$), or p_j is not part of the solution and the solution is simply $P(j-1)$. We can determine which of these is the case by taking the max of both options. Thus we have

$$P(j) = \begin{cases} 0 & \text{if } j == 0 \\ \max\Big(p_j + P\big(last(j)\big), P\big(j-1\big)\Big) & \text{otherwise} \end{cases}$$

Part C

j	I_j	p_j	$last(j)$	$P(j)$
1	$[2,3)$	2	0	$\max(2+0,0) = 2$
2	$[2,4)$	3	0	$\max(3+0,2) = 3$
3	$[1,6)$	5	0	$\max(5+0,3) = 5$
4	$[3,9)$	4	1	$\max(4+2,5) = 6$
5	$[4,9)$	4	2	$\max(4+3,6) = 7$
6	$[6,11)$	1	3	$\max(1+5,7) = 7$
7	$[5,12)$	3	2	$\max(3+3,7) = 7$
8	$[12,13)$	1	7	$\max(1+7,7) = 8$
9	$[9,14)$	3	5	$\max(3+7,8) = 10$
10	$[11,15)$	2	6	$\max(2+7,10) = 10$
11	$[14,16)$	2	9	$\max(2+10,10) = 12$
12	$[13,18)$	4	8	$\max(4+8,12) = 12$

Thus the set of intervals which provides the best solution is (I_2, I_5, I_8, I_{12}).