Algorithms 1a

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

struct point {
  int x;
  int y;
  int color;
  // 1 indicates blue
  // 0 indicates red
};

struct dominator {
  std::vector<point> points;
  int count;
};

dominator dominanceCount(std::vector<point> points) {
  dominator dom;
  dom.count = 0;
  int numBlue = 0;
  int i = 0;
  int j = 0;

  if (points.size() == 1) {
    dom.points = points;
    return dom;
  }

  point limit;
  limit.x = std::numeric_limits<int>::max();
  limit.y = std::numeric_limits<int>::max();
  limit.color = -1;
  std::vector<point> left;
  left.insert(left.begin(), points.begin(), points.begin() + points.size() / 2);

  std::vector<point> right;
  right.insert(right.begin(), points.begin() + points.size() / 2, points.end());

  dominator domL = dominanceCount(left);
  dominator domR = dominanceCount(right);

  domL.points.push_back(limit);
```

```
  domR.points.push_back(limit);

  for (std::vector<point>::iterator it = points.begin(); it < points.end(); it++) {
    if (domL.points[i].y < domR.points[j].y) {
      if (domL.points[i].y != limit.y)
        dom.points.push_back(domL.points[i]);
      if (domL.points[i].color == 1) {
        numBlue++;
      }
      i++;
    }

    else {
      if (domR.points[j].y != limit.y)
        dom.points.push_back(domR.points[j]);
      if (domR.points[j].color == 0) {
        dom.count += numBlue;
      }
      j++;
    }
  }
  dom.count += domL.count + domR.count;
  return dom;
}

int main() {
  int n;

  std::cin >> n;

  std::vector<point> points(n);

  for (int i = 0; i < n; i++) {
    std::cin >> points[i].x;
    std::cin >> points[i].y;
    std::cin >> points[i].color;
  }

  std::cout << dominanceCount(points).count;

  return 0;
}
```

Input Generation

Input is generated using a randomizer script that takes a number of iterations, and number of desired pairs.

It then randomely selects numbers to use as distinct and and y values from a specified range.

In the first test, 50 iterations were performed with 2000 pairs and randomly selected from distinct numbers between 0 and 4000.

The results of this first test follow:

Method

Average Performance

Divide and Conquer method

0.00953658

Brute Force method

0.057242916

The test times show that on average, over the 50 iterations, the divide and conquer method performed 6 times faster than the naiive brute force method.

Repeating the test with 50 iterations and twice as many pairs pairs yields a similar pattern of results:

Method

Average Performance

Divide and Conquer method

0.017255

Brute Force method

0.220860

With increasing number of pairs, the test now runs 12.80 times faster for twice the input size.

This certainly agrees with the initial analysis that the divide and conquer method should provide a faster solution.