Donut Shop Problem

Reasoning

We know that the distance a cop must travel on a perfect grid will always be $(|x_{\text{cop}} - x_{\text{shop}}| + |y_{\text{cop}} - y_{\text{shop}}|)$ as he will have to move the difference of his position and the shop position in the x and y directions, without moving diagonally. So we have to minimize the sum of all the differences in the x and y dimensions ($\sum_{i=1}^{n} |x_{\text{shop}} - x_i| + |y_{\text{shop}} - y_i|$). We can simplify this problem by making one algorithm that finds the optimal location for the donut shop in one dimension, then use it twice, once for finding the y position and once for finding the x position (our new algorithm would find an integer to minimize $\sum_{i=1}^{n} |x_{\text{shop}} - x_i|$). One way that we could make this simpler 1 dimensional algorithm could be by finding the median of the array, the only way to do this would be using the select function we looked at in class, which despite having a worst case of $O(n^2)$, has an average case of $O(n)$.

Psudocode

```
def find_optimal_shop_position_2d (A:array):
    X:array = [x for x,y in A]
    Y:array = [y for x,y in A]
    x = find_optimal_pos_1d(X)
    y = find_optimal_pos_1d(y)
    return (x,y)
def find_optimal_pos_1d (A:array):
    select(A, A.len // 2) # Take the median element from A
def select(A:array, index:int):
    pivot_value = A[0] # I am selecting the first element instead of random
since I am assuming that the order of A is not sorted.
    below = [x for x in A if x < pivot]
    above = [x for x in A if x > pivot]
    at = [x for x in A if x == pivot]
    if index < below.length():
        return select(below, k)
    if index >= below.length() and index <= below.length() + at.length():
        return pivot_value;
    if index > below.length() + at.length():
        return select(above, index-(below.length() + at.length()))
```

Proof

We want to minimize the sum $\sum_{i=1}^{n} |x_{\text{shop}} - x_i| + |y_{\text{shop}} - y_i|$. To do this we can first split it into two sums, $\sum_{i=1}^{n} |x_{\text{shop}} - x_i|$ and $\sum_{i=1}^{n} |y_{\text{shop}} - y_i|$. If we then minimize these, then we get the optimal x and y positions for the shop. The optimal position will be the median as it minimizes the sum of absolute deviations.

Running Time

Estimate

The average time is $O(n)$ with a high probability, and a worst case of $O(n^2)$.

Reasoning

The running time estimate is as follows.

```
def find_optimal_shop_position_2d (A:array):
    X:array = [x for x,y in A] # O(n)
    Y:array = [y for x,y in A] # O(n)
    x = find_optimal_pos_1d(X)
    y = find_optimal_pos_1d(y)
    return (x,y) # O(1)

def find_optimal_pos_1d (A:array):
    select(A, A.len // 2) # T(n)

def select(A:array, index:int): # T(n)
    pivot_value = A[0] # O(1)
    below = [x for x in A if x < pivot] # O(n)
    above = [x for x in A if x > pivot] # O(n)
    at = [x for x in A if x == pivot] # O(n)
    if index < below.length():
        return select(below, k) # T()
    if index >= below.length() and index <= below.length() + at.length():
        return pivot_value;  # O(1)
    if index > below.length() + at.length():
        return select(above, index-(below.length() + at.length())) # T()
```

In terms of the worst case scenario, this algorithm will take $2T(n)$ where $T(n)$ is or $O(n^2)$ time. This is because the select method takes $O(n^2)$ time in the worst case. In the probable case, the select method takes $O(n)$ time so the algorithm runs in linear time.

Worst Case Time Reasoning ($O(n^2)$)

$$T(n) = \begin{cases} T(n-1) + dn \text{ if } n > 1 \\ c \text{ if } n = 1 \end{cases}$$

$$T(n) = T(n-1) + dn$$

$$T(n-1) = T((n-1)-1) + 2dn = T(n-2) + 2dn$$

$$T(n-1) = T((n-2)-1) + 3dn = T(n-3) + 3dn$$

$$T(n) = T(n-k) + kdn$$

$$T(n) = T(0) + ndn = dn^2 = O(n^2)$$

Probable case Case Time Reasoning ($\Theta(n)$)

If we assume that x is somewhere in the middle 50% of the array, which has a 50% chance of happening, then $T(n) = \begin{cases} T(\frac{3}{4}n) + dn \text{ if } n > 1 \\ c \text{ if } n = 1 \end{cases}$ We can use the master theorem to solve this time complexity. $f(n) = dn, a = 1, b = \frac{4}{3}$

$dn = ?(n^{\frac{\log(1)}{\log(\frac{4}{3})}})$

$dn = ?(n^0)$

$dn = \Omega(1)$ So we use case 3 of the master theorem, since $f(n) = \Omega(n^{\frac{\log a}{\log b} - \epsilon})$ ($dn = \Omega(n^{\frac{\log 1}{\log \frac{4}{3}} - \frac{1}{2}}) = \Omega(-\frac{1}{2})$ ✓) This gives us $T(n) = \Theta(f(n)) = \Theta(n)$ ✓