Problem Statement

Yuckdonald's is considering opening a series of restaurants along Quaint Valley Highway (QVH). The n possible locations are along a straight line, and the distances of these locations from the start of QVH are, in miles and in increasing order, $m_1, m_2, \cdots, m_n$. The constraints are as follows:

- At each location, Yuckdonald's may open at most one restaurant. The expected profit from opening a restaurant at location m_i is p_i, where $p_i > 0$ and $i = 1, 2, \cdots, n$.
- Any two restaurants should be at least k miles apart.

We need to compute the maximum expected total profit subject to the given constraints.

Main Idea

We can use the dynamic programming paradigm to solve this problem by breaking it down into smaller subproblems and using the solutions of the smaller subproblems to construct the solution for the larger problem.

Let us define $dp[i]$ as the maximum profit earned considering only the first i positions.

When we are at the i-th position, m_i, we have two options:

- Place a restaurant at the i-th position.
- Do not place a restaurant at the i-th position.

Case 1: Not placing a restaurant at the i-th position

If we do not place a restaurant at the i-th position, then $dp[i] = dp[i-1]$. This means that the profit remains the same as when only the first $i-1$ positions were considered, which makes sense since we are not adding any new profit by skipping this position.

Case 2: Placing a restaurant at the i-th position

If we place a restaurant at the i-th position, we need to find the last position j (where $1 \leq j \leq i-1$) where a restaurant was placed, ensuring that the distance constraint $m_i - m_j \geq k$ is satisfied. We then take the maximum profit over all such j values. The recursive relation is:

$$dp[i] = \begin{cases} dp[i-1] & \text{(if no restaurant is placed at } m_i\text{)} \\ \max_{1 \leq j < i} \left(dp[j] + p_i\right) & \text{(if a restaurant is placed at } m_i\text{)} \\ & \text{subject to } m_i - m_j \geq k \end{cases}$$

This means that $dp[i]$ will either be the maximum profit without placing a restaurant at i (i.e., $dp[i-1]$), or it will be the profit at i plus the best profit we can achieve from the previous valid location j where the restaurant can be placed.

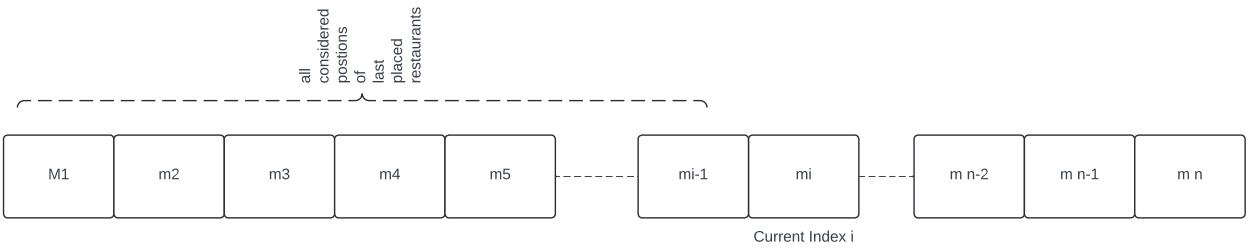

Figure 1: At index i

Pseudocode

Algorithm 1 Maximize Profit for Restaurant Placement

Initialize $dp[1 \dots n]$ to all zero // Set all values of dp array to zero
for $i = 1$ to n **do**
    for $j = 1$ to $i - 1$ **do**
        if $m_i - m_j \geq k$ **then**
            $dp[i] = \max(dp[i], dp[j] + p[i])$ // Update $dp[i]$ considering location j
        end if
    end for
    $dp[i] = \max(dp[i], dp[i-1])$ // Ensure we account for skipping location i
end for
Return $dp[n]$ // The maximum profit considering all locations

Optimized Algorithm and Pseudocode

The sliding window approach is used to optimize the problem by reducing the time complexity to $O(n)$. Instead of iterating over all previous locations to check the distance constraint for every location i, we maintain a pointer j that tracks the last valid location. The pointer j is moved forward only when the distance constraint $m_i - m_j \geq k$ is violated. This ensures that each location is processed at most once.

The steps of the sliding window approach are as follows:

- Initialize the dp array to store the maximum profit up to each location.
- For each location i, check the distance between i and the last valid location j. If the constraint is violated, increment j until the constraint $m_i - m_j \geq k$ is satisfied.
- Calculate the maximum profit at i by either placing a restaurant at i or skipping it, using the formula $dp[i] = \max(dp[i-1], dp[j] + p_i)$.
- After processing all locations, return $dp[n]$, which contains the maximum profit considering all locations.

This method is efficient because both i and j are incremented linearly, ensuring an overall time complexity of $O(n)$.

Algorithm 2 Maximize Profit for Restaurant Placement (Optimized)

```
Initialize dp[1...n] to all zero //Set all values of dp array to zero
Initialize j = 0 //Pointer to track the last valid location
dp[1] = p_1 //Base case: max profit with one restaurant
for i = 2 to n do
    while m_i − m_j < k and j <= n do
        j = j + 1 //Move pointer forward to satisfy the distance constraint
    end while
    if j ≥ 1 then
        dp[i] = max(dp[i − 1], dp[j] + p_i) //Place a restaurant at i if valid
    else
        dp[i] = dp[i − 1] //Skip placing a restaurant at i
    end if
end for
return dp[n] //The maximum profit considering all locations
```

Proof of Correctness

We are given n locations along a highway at distances $m_1, m_2, \ldots, m_n$, with profits $p_1, p_2, \ldots, p_n$, and no two restaurants can be placed less than k miles apart. We want to compute the maximum possible profit.

Dynamic Programming Definition

Define $dp[i]$ as the maximum possible profit considering the first i locations. The recurrence relation for $dp[i]$ is:

$$dp[i] = \max\left(dp[i-1], \max_{j:m_i - m_j \geq k}(dp[j] + p_i)\right)$$

Base Case:

For the first location, the maximum profit is either placing a restaurant at m_1 or not:

$$dp[1] = \max(0, p_1)$$

Inductive Hypothesis:

Assume $dp[j] = OPT(j)$ for all $j \leq i-1$, i.e., the dynamic programming solution computes the optimal profit for all locations up to $i-1$.

Inductive Step:

For location i, there are two possibilities:

- **Case 1: No restaurant at m_i:**

  If no restaurant is placed at location m_i, then the maximum profit at i is the same as the profit at $i-1$. Therefore, the recurrence is:

  $$dp[i] = dp[i-1]$$

  By the inductive hypothesis, we know that the dynamic programming solution is optimal for all previous locations, i.e., for $j < i$, we have:

  $$dp[i-1] = OPT(i-1)$$

  Thus, we can substitute into the equation for $dp[i]$:

  $$dp[i] = dp[i-1] = OPT(i-1)$$

  Since no restaurant is placed at m_i, the optimal profit for the first i locations is the same as the optimal profit for the first $i-1$ locations:

  $$OPT(i) = OPT(i-1)$$

  Therefore, it follows that:

$$dp[i] = OPT(i)$$

In this case, the dynamic programming solution correctly computes the maximum profit for the first i locations.

- **Case 2: A restaurant is placed at m_i:**

  If a restaurant is placed at location m_i, we need to find the last valid location m_j such that the distance constraint $m_i - m_j \geq k$ is satisfied. The profit at location i is the sum of the profit from placing a restaurant at m_i (i.e., p_i) and the maximum profit achievable up to the location j. This leads to the recurrence:

  $$dp[i] = \max_{j:m_i - m_j \geq k} (dp[j] + p_i)$$

  By the inductive hypothesis, the dynamic programming solution is optimal for all previous locations, i.e., for all $j < i$, we have:

  $$dp[j] = OPT(j)$$

  Substituting this into the recurrence for $dp[i]$, we get:

  $$dp[i] = \max_{j:m_i - m_j \geq k} (OPT(j) + p_i)$$

  This matches the optimal solution for the first i locations:

  $$OPT(i) = \max_{j:m_i - m_j \geq k} (OPT(j) + p_i)$$

  Therefore, the dynamic programming solution for i also computes the optimal solution in this case, and we have:

  $$dp[i] = OPT(i)$$

In both cases, whether a restaurant is placed at m_i or not, the dynamic programming solution correctly computes the maximum profit up to the i-th location. By induction, we conclude that $dp[i] = OPT(i)$ for all i, proving that the dynamic programming approach is correct and optimal.

Time Complexity

The time complexity of the sliding window approach is $O(n)$. The outer loop iterates through each location from 1 to n, processing each location exactly once. The pointer j moves forward only when the distance constraint is violated, and since both i and j move in one direction and are incremented linearly, each element is visited at most once. Therefore, the overall time complexity is $O(n)$, as no location is revisited unnecessarily.