**Big O – Baseball Itinerary Program - jerryberry**

**Data::askDijkstra() <datastructures.cpp line 708>**

* This method is an implementation of Dijkstra’s algorithm to find the shortest path between two vertices on a graph.
* This implementation runs in O(elogv)
  + e = number of edges (stadium connections)
  + v = number of vertices (stadiums)
* Pseudocode:

//v operations

For (all stadiums)

{

Push each stadium into a minimum heap

}

//e operations

While (heap is not empty)

{

//constant time operations (pre-sorted adjacency list allows for this)

Explore connections of stadium at the top of the minimum heap

//log v operations

Pop the top off the minimum heap

}

Return distances to all other stadiums from starting stadium

* Analysis:
  + This implementation first pushes every stadium into a minimum heap which runs in O(v)
  + Then it enters a loop that pops the top off the minimum after exploring all neighbors that are connected to that top which runs in O(elogv)
  + This implementation uses an adjacency list representation of the stadium graph
    - The adjacency list already has all connections pre-sorted such that the closest stadium to a stadium of interest is always at the front of the list
  + When exploring connections from a stadium at the top of the minimum heap, the adjacency list eliminates the need to iterate through all connections of that stadium to find the closest one
  + Once the minimum heap is empty, the method is completed
* **This method runs in v + elogv, but since elogv is bigger than v, the big o is O(elogv)**

**PrimeWin::on\_itinOptimizeBt\_clicked() <primewin.cpp line 1308>**

* This method reorders a tourist’s itinerary such that their trip’s total distance is minimized
* This implementation runs in O(n^2)
  + n = number of stadiums
* Pseudocode:

//n operations

For (all stadiums)

{

Paint all stadiums as unvisited

}

//worst case n operations if all stadiums are queued in the itinerary

For (number of queued stadiums in the itinerary)

{

Paint all queued stadiums as optimizable

}

//worst case n operations if all stadiums are queued in the itinerary

For (number of queued stadiums in the itinerary)

{

//elogn operations where e is the number of edges on the graph

Perform Dijkstra operation to get vector of distances to all other stadiums

//n operations

For (all stadiums)

{

Repaint stadiums to update if the optimize has visited them or not

}

Sort Dijkstra’s vector to get next closest stadium which runs in nlogn

Add the closest stadium to the optimized itinerary

}

Replace old itinerary with new optimized one

* Analysis:
  + This method paints all stadiums as unvisited first which runs in O(n)
  + It then paints the queued stadiums as stadiums that can be visited by iterating through the itinerary which runs in O(n) in the worst case if every stadium is queued in the itinerary
  + It then enters a loop that will iterate through the itinerary which runs in O(n) if every stadium is queued in the worst case
  + Within this loop, two O(nlogn) sorts are called as well as another for loop that runs in O(n) which makes the loop run in O(n^2)
* **This method runs in n + n + n (elogn + n + nlogn) which simplifies to O(n^2)**

**Data::addDist() <datastructures.cpp line 297>**

* This method modifies a distance in the adjacency matrix and in the adjacency list
* This method runs in O(nlogn)
  + n = number of stadiums
* Pseudocode:

//Modify distance in matrix which runs in O(1)

Modify distance in matrix[x][y]

Modify distance in matrix[y][x]

//Search through the list at stadium x to see if there’s a connection to stadium y

//This operation’s worst case is O(n) if the stadium is connected to every other stadium

While (iterator is not at the end of a stadium’s neighbor list)

{

Increment iterator

}

If (adding a distance)

{

Insert new distance at the end of the list which is O(1)

Sort the list based on distances which runs in O(nlogn)

}

If (deleting a distance)

{

Delete the iterator which runs in O(1)

}

* Analysis:
  + Adding to the matrix always runs in constant time since element access in a matrix is always constant time
  + The adjacency list is a vector of doubly linked lists
    - Each position in the vector represents a stadium
    - Each doubly linked list represents a stadium’s connections to other stadiums
  + Every time a new distance is added, the doubly linked list is resorted based on distance so that the smallest distance is always at the front, which is O(nlogn) in the worst case
* **This method runs in n + nlogn which simplifies to O(nlogn) since nlogn is bigger than n**

**PrimeWin::calcTrip() <primewin.cpp line 486>**

* This method calculates the total trip distance of an itinerary
* This method runs in O(nelogn)
  + n = number of stadiums
* Pseudocode:

//Loop through entire itinerary which runs in O(n) if every stadium is queued in the worst case

While (iterator is not at the end of the itinerary)

{

Perform a Dijkstra operation which is O(elogn)

//e is the number of edges

Advance iterator

Add to running total mileage

}

* Analysis:
  + The primary while loop will iterate through the entire itinerary
    - If every stadium is queued in the itinerary, then the worst case happens which is O(n)
  + Within that loop, Dijkstra’s algorithm is called which runs in O(elogn)
    - e is the number of edges in the graph
    - n is the number of stadiums
* **This method runs in n (elogn) which is already simplified, so it is O(nelogn)**