2

The login page design is very simple. Right now there's screenshots because currently the app is in two pieces - partly on my laptop and partly on Gary's laptop but Gary has the interesting bits we'll be demonstrating today. In later versions we'll probably add pictures of the city in order to get people in the mood to run. There won't be many user features because of time constraints but the main selling point will be there and that's what's important. Before we go into the demonstration let's go over the algorithm used.

3

For algorithm, we gonna use DFS(depth first search) as our basic concept. DFS is a kind of technique for traversing a graph, which when it chooses which vertex to explore next, it favors children over siblings. Children here for graph is the adjacent vertices so that DFS will go all the way to the end of one of routes, then if the vertex doesn't have any adjacent vertex or any adjacent vertex is visited, it will go back to the last vertex to go to another adjacent vertex of this vertex.

4,5

Here is one example for how DFS implements. We assume the start vertex is A, then it will go to B because B is connected to A and B is not visited. Then the only adjacent vertex of B is C, thus it go to C. A is the first vertex connecting to C, but A is visited so that D will be the next. For D, A is the only adjacent vertex and A is visited, thus it is going to go back to C to see another adjacent vertex of C. The next one is E and E is not visited, so it choose E as the next. Finally A is the only vertex connecting to E and A is visited, it will go back to C. Following this, when it go back to the start vertex which is A and all vertexes have been visited, it will stop.

6

It seems like DFS is a traversing algorithm. However, it also can be used as path finding. Since DFS always choose an adjacent vertex as the next, every time it goes back, it actually find a possible route from the start vertex to the last visited vertex. Thus we just use a stack to keep track of the path between the start vertex and the current vertex so that it will store path. In addition, we add the condition which as soon as destination vertex is encountered, it return the path as the contents of the stack. That is how we use it to find all paths between two specific vertices.

7

Here is the pseudocode of the main function of the algorithm we use for our application:

```
        DFS(depth,v,path):
    path.push(v); //add the current vertex to the path
    visited.push(v);// label the current vertex as visited
    if (depth == 0 && v is target): // if depth is 0 and the current vertex is the destination
    print path// print the path or store the path
    Return //stop clause for successful branch
    if (depth == 0): // if depth is 0, it means the current vertex is not the destination
     return   //stop clause for non successful branch return
    for each vertex u such that (v,u) is an edge: // create an iterator to go through adjacent vertices
        DFS(depth-distance(v,u),u,path) //recursively check all paths for of shorter depth. Every time it goes to the next vertex, the depth will subtract the edge of these two vertices
          path.removeLast(); // remove the last because it means it goes back
          visited.removeLast();// and remove the label
```

8

For now, we have solved the problem how to find path between two specific vertices. However, the goal of our algorithm is to find cycles with specific length. So we adjusted the DFS to make it find cycles for us. We use the above algorithm to find all routes ending in the vertices which are adjacent to the start vertex, then add the start vertex as the last vertex of the route. We successfully made the algorithm to find all cycles with specific length. After that, we add another feature to select routes of these cycles which is difficulty of routes. Difficulty means the route is longer, it is more difficult, and the difference of elevations is bigger, it is more difficult. Based on this, we store the sum of all absolute values of difference of two adjacent vertices' elevation. Thus when the route is longer, the sum will be bigger, and when the elevation of the route is keep changing more and not flat, the sum will be bigger. That is what we want. According to these adjustment, we finally get the selected routes which match the users' preference.

8.5

Now onto the demonstration of our route finding process. There are three parameters we'll have the user input: the starting location (in this case we're using the grid notation which I'll get to later on, but in the final version the user will just input cross streets), the length (this is in meters for ease of implementation), and lastly the difficulty of the route (this is the absolute change in altitude across the route divided by the overall length). For our example we'll be using starting point 12, 1000 meters, and 10 difficulty. We could make the routes longer (1km is about .6 miles so it isn't a long run, but we'd get far too many results, even with 1.5km). This gives us routes 4, 6, and 10 as potential picks for our user. Let's map out all of them.

9

Before we map them out we only need to know how to read the numbers the algorithm gives out. Here's half the points used by our database to make route (we have 25 points here for the sake of the example and to still be readable, but we likely wouldn't have gotten very far with a 1km limit). It's a simple x,y coordinate grid and here's our starting point for all of our routes.

10

This is route 4. It goes a little bit over the requested length but it's not a problem. If we limited routes to exactly the length we'd get far fewer viable routes. We're using mappings from Google maps because we plan on using the android api to display points and this would be a close second in absence of the actual map on our app. Also these numbers are the absolute difference in elevation across the points so you can get an idea of how that comes into play. Going eastward means going downhill and going westward means going uphill for this neighborhood. Make note that to build this route we used 8 points.

11

This our next suggested route, route 6. We used 11 points to make this route and it's a little bit under the 1km requirement but that's fine for us.

12

Finally our last route, number 10. This one doesn't really resemble the others as it juts out to the side here but that variation is what we want. We don't want simple routes all the time.10 points of data used to make.