**Character Mapping Challenge Overview and Description**

In this write-up, I will detail what my approach and reasoning was with the given challenge and how I arrived at the solutions that I did. I will describe several important points concerning how the solutions operate internally along with various configuration options designed to lend flexibility toward solving a broad set of problems.

Fundamentally, this challenge was a simple grid navigation exercise. The program is responsible for navigating a 6x6 grid of letters and numbers, mimicking the movements of a cursor being manipulated with navigational arrows. The grid that was to be used is shown below, and the cursor was to start at the top-left corner, on the letter “A”.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F |
| G | H | I | J | K | L |
| M | N | O | P | Q | R |
| S | T | U | V | W | X |
| Y | Z | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 | 0 |

The program was then required to output the results of the navigation and selections as part of a comma-delimited set of characters. In the instructions, the example of “IT CROWD” would result in the output of “D,R,R,#,D,D,L#,S,U,U,U,R,#,D,D,R,R,R,#,L,L,L,#,D,R,R,#,U,U,U,L,#”. In this sequence, each “#” character represents the selection of the character on which the cursor is located, “U”, “D”, “R”, “L” signify the directions of up, down, right and left, and “S” signifies a space.

**Choice of Technology**

When deciding which technology to use for this project, I initially narrowed it down to three options:

- C#  
 - C++  
 - Python

My reason for considering C# is generally pretty obvious. I work with .Net on a daily basis and am extremely familiar with all of the mechanisms that would be involved in solving a challenge like this. C# is probably the single most familiar language and environment for me at the moment due to its prominence in my work. It’s not perfect and has its limitations, but I can usually put something together pretty quickly for something like this.

My reason for considering Python is because I have been making more and more use of Python in the past several months to help solve a variety of problems that can be more tedious in other languages. I am amazed at how well it works for testing secure, RESTful APIs, along with data massaging, and other miscellaneous tasks. It, too, has its downsides. But for a task like this, Python seemed like a good fit algorithmically.

My reason for considering C++ came down to a combination of code familiarity and potential performance. I have a long background in C++ which, for some time, I have *not* been using it as extensively as I once did. Due to the nature of how things work between the stack and the heap in C++, there are particular coding patterns in C++ – such as the use of heap based objects to manage the release of stack-based resources – that you never see in languages that use managed heaps with garbage collection. In C#, a reference-based object exists in the heap, and while it provides mechanisms like the “using” keyword to control invocation of the Dispose() method on objects that implement IDisposable, it otherwise forces you maintain a certain level of disconnect regarding when an object will truly be destroyed.

I began the work in C++ and initially planned for that to be my sole submission. However, once I had figured out how I wanted to structure the code algorithmically, I quickly realized that all three languages would work well and decided to implement a solution in all three: C++, C#, and Python.

**General Description of Algorithm**

All of the elements in the character map are stored in a hash table that is keyed with the character and contains the row and column coordinates of that character for its value. A sampling of those entries is shown below. Bear in mind, that this does not represent the order in which they are actually stored in the hash. The order in which they are stored is based on the hashing algorithm. They are simply presented in this order to help understand the meaning of the values. In this example, letter “A” is in row #0 (zero-based) and column #0. Letter “P” is in row #2 and column #3.

|  |  |  |
| --- | --- | --- |
| A | -> | [ 0, 0 ] |
| B | -> | [ 0, 1 ] |
| … | … | … |
| O | -> | [ 2, 2 ] |
| P | -> | [ 2, 3 ] |
| … | … | … |
| Z | -> | [ 5, 1 ] |
| 1 | -> | [ 5, 2 ] |

For each character in a given term that is being processed, the solution does a lookup on this table using the character as its key to obtain the coordinates for the next position. It then calculates the number of up or down movements to get to the corresponding row followed by the number of left or right movements to get to the corresponding column. It generates the sequence of movements, along with the “#” for selection and advances to the next character in the term.

If a space is encountered, “S” is added to navigation sequence. If the letter in the term is not found in the character map, then “X” is added to the navigation sequence.

Because the grid is composed of units that are all connected to one another, with no obstacles, with equal transition time on all horizontal movements and equal transition time on all vertical movements, calculation of the most efficient route is very straightforward. It can be shown mathematically that a route comprised of one set of up or down movements and one set of left or right movements cannot be improved upon in terms of efficiency.

To understand, consider the example of moving from “A” to “V” in the above grid. You could do the following: D, D, D, R, R, R. You could also do a more dog-legged route of D,R,D,R,D,R. In both of these routes, you have three down movements and three right movements, yielding the same net efficiency.

**Dynamic Elements**

In all of the solutions, the following elements are all configurable through a combination of command line values and configuration files:

- Traversal technique (bounded or infinite)  
- Layout of Grid (with support for unicode character set)  
- Starting position of cursor  
- Alternate character mappings (lowercase to uppercase, accent characters, etc.)

The following sections describe each of these configuration points.

**Bounded vs Infinite Grid**

One potential optimization which could not be validated from the example of “IT CROWD” was the concept of an infinite grid versus a bounded grid. To understand what this means, consider what would happen if the cursor was moved left when it was over the “A” character in the grid above. In an infinite grid, the cursor would move to the “F” on the opposite end of the row, thus being able to move freely and infinitely in all directions with a simple wraparound to the opposite side any time it moves past a boundary. With a bounded grid, attempting to move left while on the “A” character would result in no change of position. Thus, all movement is restricted to the boundaries of the grid.

With an infinite grid, there are several opportunities that arise for improved paths to a character. Consider, for example, the path sequence from the second to the third letter in my name: DARYL. When moving from “A” to “R”, you could go down two rows and then five columns to the right, resulting in seven total moves. However, with an infinite grid, you could, instead, go down two rows and then move one column to the left, yielding only three total moves.

While the savings are not usually this dramatic (> 50%), I have found the savings in total moves from the infinite grid to be very meaningful. Using a sampling of 10,000 random words, I found that a bounded grid required about 13% more overall moves than that of an infinite grid.

**Layout of Grid**

The grid can be configured in any rectangular arrangement of letters, numbers, or other symbols. The solutions, in all languages, support unicode, albeit with a limitation of 2 byte codepoints in C++. This enables support for character maps with foreign language letters and symbols – thus enabling it to work for devices that operate in Europe, Asia and other parts of the world. It also enables it to work for a variety of grid arrangements. For example, some players will arrange the characters all on one row, as shown below.

ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789

Other players might choose to incorporate other characters, such as dashes, dollar signs, and percent signs, and thereby need a larger grid. These things, along with foreign language characters, can all be handled by the solutions presented here.

**Starting Position of Cursor**

Because of the expected variance in grid layouts and characters within the grid, the starting position of the cursor can also be expected to vary. For this example, it started over the letter “A” in the top left corner. Other devices might position it in one of the other three corners. Or, perhaps they will remember the last character used and position it there.

All of the solutions presented here have a command line argument for specifying the default starting character. If this argument is not provided, it is assumed to be the letter “A”.

**Alternate Character Mappings**

Alternate character mappings are used to define what character should be used in place of another character within an input term. The most common application in this example is casing. A lowercase “e” within a term should be treated as a capital “E” for the purpose of grid navigation.

Although mapping lowercase letters to their uppercase equivalents is the most common example, there are others. Consider, for example, the word “résumé”. In this word, the symbol “é” should also be mapped to “E”.

The same file that is used to define custom character maps also includes a section for defining alternate character mappings. These mappings enable input words with lowercase letters, along with special symbols, like “é”, “Ö”, and “ñ”, to be handled gracefully and correctly.

**Performance Characteristics**

One of the advantages of having a solution in all three languages is that it enabled me to carry out side-by-side performance evaluations of each solution using the same data set. I considered performance to be of relative importance for something like this as it was easy to imagine a solution like this in place as part of a high volume web API where poor performance would, at best, increase the demands for server hardware and related costs, and, at worst, limit the scalability of the solution altogether.

Some of the performance results were generally expected. For example, both C++ and C# fared much better than Python. Given that Python is an interpreted language, this is unsurprising. However, quite surprisingly, C# performed better than C++. I suspect that this is primarily due to improvements in the hashing algorithm of the .Net library compared with that of STL and, with a custom hasher, most of that difference could be eliminated.

|  |  |  |  |
| --- | --- | --- | --- |
| Total Seconds to Execute | | | |
|  |  |  |  |
|  | Number of Records | | |
| 100,000 | 500,000 | 5,120,000 |
| Python | 5 | 28 | 292 |
| C++ | 0 | 3 | 33 |
| C# | 0 | 1.5 | 13 |

**Other Potential Complications and Variations**

The submissions in this solution followed a relatively simple methodology of loading and processing all of the strings at once. For significantly larger data sets beyond the 5 million rows tested, this would have serious memory implications. A far more elegant solution would be to carry out “windowed” reads. This would entail opening the file, reading a section, closing the file, processing that section, then repeating the process for the next section.

If a need arose to use a genuinely non-rectangular grid, the process for finding the shortest route would become more complicated. The number of grid coordinates would remain small enough that it would not turn into a true “travelling salesman” problem, but it would still require a more sophisticated methodology, especially when an infinite grid is in use. Consider the best route from “L” to “O” in the following hypothetical (though not practical) mapping:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F |
| G | H | I | J | K | L |
| M | N | O |  |  |  |
| P | Q | R |  |  |  |
| S | T | U | V | W | X |
| Y | Z | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 | 0 |

The best route from “L” to “O” is somewhat unconventional: R, D, L

**Closing Notes**

This challenge was relatively simple to solve, but presented ample opportunity for mechanisms to accommodate variations of input. I sought to address four major points while also making the utilities user-friendly to operate.

All of the utilities will present a set of instructions concerning the command line switches if you type the executable without any arguments. Furthermore, while they support the use of a “Keys” file to specify a new mapping, such a file is not required and they can all use the default mapping that was presented at the beginning of this write-up. Lastly, the command-line argument for specifying a starting character is not case-sensitive. For example, if a user specifies a starting character of “z”, the program will respond correctly, even though the default grid only has “Z” on it. These design considerations were all factored in to make the user experience easier and more pleasant.