Row Major Indexing

Given a 2D array, a row major index for any element in the array is the index that the element would have if we flatten out the array, and concatenate the rows of the array one after the other.

Example: Consider the following 2D array:

```
  0  1  2
0  x  t  z
1  y  a  b
2  c  v  d
```

If we were to write the above 2D array as a vector in row major form, the vector would be:

```
  0  1  2  3  4  5  6  7  8
x  t  z  y  a  b  c  v  d
```

Thus, the row major index for the element ‘c’ is 6.

More generally, you can calculate the row major index of an element at position (i,j) in a 2D array as $i \times \text{Width} + j$, where Width is the number of columns in the 2D array.

In the above example, Width = number of columns in the 2D array = 3.

Therefore, the row major index for the element ‘c’, which is at position (2,0) in the 2D array is given by $2 \times 3 + 0 = 6$, which is exactly what we got before.

The knowledge of row major indices is useful while implementing the `isOnBoard()` function in PA4. The `isOnBoard()` function has the following signature:

```cpp
vector<int> isOnBoard(const string& word_to_check)
```

This function takes a string as an argument and checks whether that string can be formed by the dice on the current Boggle board. If yes, then it returns a vector of the row major indices of the dice that form `word_to_check`. For instance, in the above example, if we are looking for the word “cat”, since it can be formed by the dice on the board, the `isOnBoard()` function should return a vector containing the row major indices of the letters “c”, “a” and “t” in order. This means that `isOnBoard(“cat”)` should return the vector (6,4,1).
**Lexicon Storage and Ternary Tries**

The reference solution to PA4 uses the C++ set class to store the lexicon. However, the use of ternary tries may potentially speed up the lookup time.

In a ternary trie, each node contains:

1. a **key** digit.
2. 3 child pointers **left, middle, and right**, corresponding to keys whose digit being considered is less than, equal to, or greater than the node’s digit.
3. an **end bit**, to indicate that this node contains a key.

Example:

Let’s insert the following strings into a ternary trie in the order given below. In the below pictures, a node with a double outline indicates that the end bit for that node is true.

call, me, how, mind, not, no, money, milk, note.

Inserting “call”:
Inserting “me”.

Inserting “how”.

[Diagram of tree structures with nodes labeled 'c', 'a', 'l', 'm', 'e', 'h', 'o', 'w']
Inserting “mind”.

c
  ↘
  ↓
  a

  ↘
  ↓
  l

  ↘
  ↓
  I

  ↘
  ↓
  I

  ↘
  ↓
  w

  ↘
  ↓
  n

  ↘
  ↓
  d

  ↘
  ↓
  o

  ↘
  ↓
  e

  ↘
  ↓
  h

  ↘
  ↓
  m

  ↘
  ↓
  I

  ↘
  ↓
  c
Inserting “not”.

c
  ↓
  a

  ↓
  l

  ↓
  l

  ↓
  w

  ↓
  d

  ↓
  n

  ↓
  o

  ↓
  t

  ↓
  i

  ↓
  e

  ↓
  h

  ↓
  m

  ↓
  a

  ↓
  c
Inserting “no”.

Diagram: A tree structure with nodes labeled from top to bottom: c, a, l, h, e, m, i, o, n, t, w, d.
Inserting “money”.
Inserting “milk”.

The diagram shows a tree structure with nodes labeled with characters from the word "milk". Each node is connected to its parent node, forming a branching structure."
Inserting “note”.

If we were to look for the word “mint” in the ternary trie shown above, here are the steps that we would follow:

1. Start at the root node with digit ‘c’.
2. Since the first digit of the search key ‘m’ > ‘c’, traverse to the right child of the current node.
3. Thus, we reach the node with digit ‘m’. Since the first digit of the search key ‘m’ equals the digit of the current node, traverse down the middle child.
4. We reach the node with digit ‘e’. Also, now we are looking for a match for the second digit ‘i’ of the search key. Since the second digit of the search key ‘i’ > ‘e’, traverse to the right child of the current node.
5. We reach the node with digit ‘i’. Since the second digit of the search key ‘i’ equals the digit of the current node, traverse down the middle child.
6. We reach the node with digit ‘n’. Also, now we are looking for a match for the third digit ‘n’ of the search key. Since the third digit of the search key ‘n’ equals the digit of the current node, traverse down the middle child.
7. We reach the node with digit ‘d’. Also, now we are looking for a match for the fourth digit ‘t’ of the search key. Since the fourth digit of the search key ‘t’ > ‘d’, we try to traverse to the right child of the current node. But, since the right child of the current node is nullptr, therefore, we know that the key ‘mint’ does not exist in the lexicon.

**Boggle Board Representation**

In PA4, we need to do a search for a simple acyclic path on the Boggle board to determine whether a word can be formed on the board or not. A possible implementation for this can be done by using graphs.

The 3 different graph search algorithms discussed in class are breadth-first search, depth-first search and Dijkstra’s algorithm.

In this assignment, we need to start at a position on the Boggle board and traverse down one path to find a word completely. If the search along that path fails, then we backtrack and try a different path. This search is exactly like depth-first search. In depth-first search, you start at a node and then traverse down a single branch of the graph until you have visited as many nodes along that branch as you can. When you cannot go any further, you backtrack and resume traversal along a different branch. That’s why, it is recommended that you implement depth-first search for this assignment.