Coding Ninjas

C++ Foundation with Data Structures

Topic: Recursion

Recursion

a. What is Recursion?

In previous lectures, we used iteration to solve problems. Now, we'll learn about recursion for solving problems which contain smaller sub-problems of the same kind.
Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem. By same nature it actually means that the approach that we use to solve the original problem can be used to solve smaller problems as well.
So in other words in recursion a function calls itself to solve smaller problems. Recursion is a popular approach for solving problems because recursive solutions are generally easier to think than their iterative counterparts and the code is also shorter and easier to understand.

b. How Does Recursion Work?

We can define the steps of a recursive solution as follows:

1. **Base Case:**
   A recursive function must have a terminating condition at which the function will stop calling itself. Such a condition is known as a base case.
2. **Recursive Call:**
   The recursive function will recursively invoke itself on the smaller version of problem. We need to be careful while writing this step as it is important to correctly figure out what your smaller problem is on whose solution the original problem's solution depends.
3. **Small Calculation:**
   Generally we perform a some calculation step in each recursive call. We can perform this calculation step before or after the recursive call depending upon the nature of the problem.

It is important to note here that recursion uses stack to store the recursive calls. So, to avoid memory overflow problem, we should define a recursive solution with minimum possible number of recursive calls such that the base condition is achieved before the recursion stack starts overflowing on getting completely filled.

Now, let us look at an example to calculate factorial of a number using recursion.

Coding Ninjas

Example Code 1:

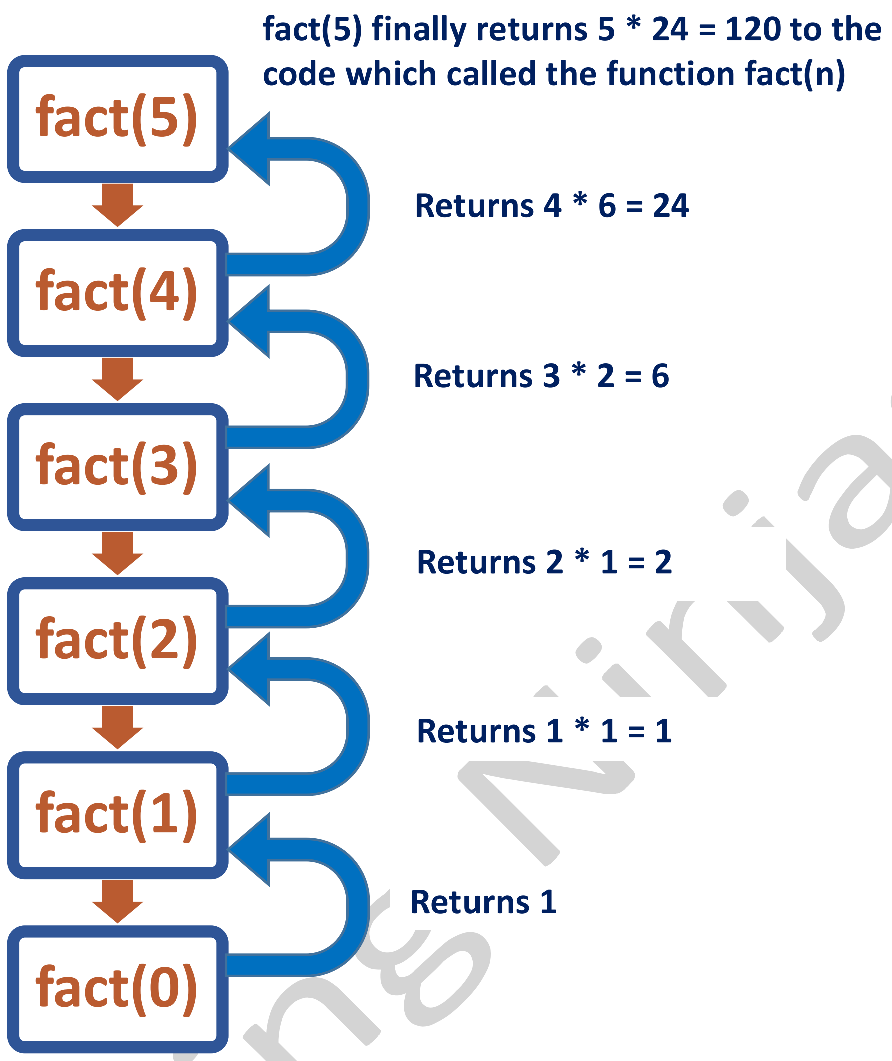
```
#include<iostream>
using namespace std;

int fact(int n)
{
   if(n==0)                          //Base Case
   {
      return 1;
   }
   return n * fact(n-1);             //Recursive call with small calculation
}

int main()
{
   int num;
   cin>>num;
   cout<<fact(num);
   return 0;
}
```

Output:

```
120            //For num=5
```

Explanation:
Here, we called factorial function recursively till number became 0. Then, the statements below the recursive call statement were executed. We can visualize the recursion tree for this function, where let n=5, as follows:

Coding Ninjas



We are calculating the factorial of n=5 here. We can infer that the function recursively calls fact(n) till n becomes 0, which is the base case here. In the base case, we returned the value 1. Then, the statements after the recursive calls were executed which returned n*fact(n-1) for each call. Finally, fact(5) returned the answer 120 to main() from where we had invoked the fact() function.

Coding Ninjas

Now, let us look at another example to find nth Fibonacci number . In Fibonacci series to calculate nth Fibonacci number we can use the formula F(n) = F(n – 1) + F(n – 2) i.e. nth Fibonacci term is equal to sum of n-1 and n-2 Fibonacci terms. So let's use this to write recursive code for nth Fibonacci number.

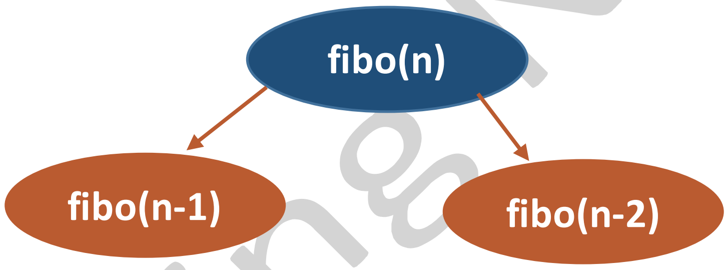
Example Code 2:

// Recursive function:

```
int fibo(int n) {
   if(n==0 || n==1) {              //Base Case
      return n;
   }
   int a = fibo(n-1);              //Recursive call
   int b = fibo(n-2);              //Recursive call
   return a+b;                     //Small Calculation and return statement
}
```

Explanation:
As we are aware of the Fibonacci Series (0, 1, 1, 2, 3, 5, 8,... and so on), let us assume that the index starts from 0, so, 5th Fibonacci number will correspond to 5; 6th Fibonacci number will correspond to 8; and so on.

Here, in recursive Fibonacci function, we have made two recursive calls which are depicted as follows:



Note: One thing that we should be clear about is that both recursive calls don't happen simultaneously. First fibo(n-1) is called, and only after we have its result and store it in "a" we move to next statement to calculate fibo(n – 2).

It is interesting to note here that the concept of recursion is based on the mathematical concept of **PMI** (Principle of Mathematical Induction). When we use PMI to prove a theorem, we have to show that the base case (usually for x=0 or x=1) is true and, the induction hypothesis for case x=k is true must imply that case x=k+1 is also true. We can now understand how the steps which we followed in recursion are based on the induction steps, as in recursion also, we have a base case while the assumption corresponds to the recursive call.

Coding Ninjas

C++ Foundation with Data Structures

Topic: Time Complexity

Coding Ninjas

Introduction

An important question while programming is: How efficient is an algorithm or piece of code?

Efficiency covers lots of resources, including:

- CPU (time) usage
- memory usage
- disk usage
- network usage

All are important but we are mostly concerned about CPU time.

Be careful to differentiate between:

1. Performance: how much time/memory/disk/etc. is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code we write.

2. Complexity: how do the resource requirements of a program or algorithm scale, i.e. what happens as the size of the problem being solved by the code gets larger. Complexity affects performance but not vice-versa.
The time required by a function/method is proportional to the number of "*basic operations*" that it performs.

Here are some examples of basic operations:

- one arithmetic operation (e.g. a+b / a*b)
- one assignment (e.g. int x = 5)
- one condition/test (e.g. x == 0)
- one input read (e.g. reading a variable from console)
- one output write (e.g. writing a variable on console)

Some functions/methods perform the same number of operations every time they are called.
For example, the size function/method of the string class always performs just one operation: return number of Items; the number of operations is independent of the size of the string. We say that functions/methods like this (that always perform a fixed number of basic operations) require constant time.

Coding Ninjas

Other functions/methods may perform different numbers of operations, depending on the value of a parameter. For example, for the array implementation of the Vector/list(Java) class(vector/list classes are implemented similar to the dynamic class we have built), the remove function/method has to move over all of the items that were to the right of the item that was removed (to fill in the gap). The number of moves depends both on the position of the removed item and the number of items in the list. We call the important factors (the parameters and/or fields whose values affect the number of operations performed) the problem size or the input size.

When we consider the complexity of a function/method, we don't really care about the exact number of operations that are performed; instead, we care about how the number of operations relates to the problem size. If the problem size doubles, does the number of operations stay the same? double? increase in some other way? For constant-time functions/methods like the size function/method, doubling the problem size does not affect the number of operations (which stays the same).

Furthermore, we are usually interested in the worst case: what is the most operations that might be performed for a given problem size. For example, as discussed above, the remove function/method has to move all of the items that come after the removed item one place to the left in the array. In the worst case, all of the items in the array must be moved. Therefore, in the worst case, the time for remove is proportional to the number of items in the list, and we say that the worst-case time for remove is linear to the number of items in the array. For a linear-time function/method, if the problem size doubles, the number of operations also doubles.

Coding Ninjas

Big-O Notation

We express complexity using big-O notation. For a problem of size N:
a constant-time function/method is "order 1": O(1)
a linear-time function/method is "order N": O(N)
a quadratic-time function/method is "order N squared": $O(N^2)$
Note that the big-O expressions do not have constants or low-order terms. This is because, when N gets large enough, constants and low-order terms don't matter (a constant-time function/method will be faster than a linear-time

function/method, which will be faster than a quadratic-time function/method). See below for an example.
Formal definition:

A function T(N) is O(F(N)) if for some constant c and for all values of N greater than some value n_0:
T(N) <= c * F(N)
The idea is that T(N) is the exact complexity of a function/method or algorithm as a function of the problem size N, and that F(N) is an upper-bound on that complexity (i.e. the actual time/space or whatever for a problem of size N will be no worse than F(N)). In practice, we want the smallest F(N) - the least upper bound on the actual complexity.
For example, consider T(N) = 3 * N^2 + 5. We can show that T(N) is O(N^2) by choosing c = 4 and n_0 = 2. This is because for all values of N greater than 2:

3 * N^2 + 5 <= 4 * N^2
T(N) is not O(N), because whatever constant c and value n0 you choose, I can always find a value of N greater than n_0 so that 3 * N^2 + 5 is greater than c * N.

How to Determine Complexities

In general, how can you determine the running time of a piece of code? The answer is that it depends on what kinds of statements are used.

1. Sequence of statements

```
statement 1;
statement 2;
  ...
statement k;
```

The total time is found by adding the times for all statements:
total time = time(statement 1) + time(statement 2) + ... + time(statement k)

If each statement is "simple" (only involves basic operations) then the time for each statement is constant and the total time is also constant: O(1). In the following examples, assume the statements are simple unless noted otherwise.

Coding Ninjas

2. if-then-else statements

```
if (condition) {
   sequence of statements 1
}
else {
   sequence of statements 2
}
```

Here, either sequence 1 will execute, or sequence 2 will execute. Therefore, the worst-case time is the slowest of the two possibilities: max(time(sequence 1), time(sequence 2)). For example, if sequence 1 is O(N) and sequence 2 is O(1) the worst-case time for the whole if-then-else statement would be O(N).

3. for loops

```
for (i = 0; i < N; i++) {
   sequence of statements
}
```

The loop executes N times, so the sequence of statements also executes N times. Since we assume the statements are O(1), the total time for the for loop is N * O(1), which is O(N) overall.

4. Nested loops

First we'll consider loops where the number of iterations of the inner loop is independent of the value of the outer loop's index. For example:

```
for (i = 0; i < N; i++) {
   for (j = 0; j < M; j++) {
      sequence of statements
   }
}
```

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the inner loop execute a total of N * M times. Thus, the complexity is O(N * M). In a common special case where the stopping condition of the inner loop is j < N instead of j < M (i.e., the inner loop also executes N times), the total complexity for the two loops is O(N^2).

Coding Ninjas

Now let's consider nested loops where the number of iterations of the inner loop depends on the value of the outer loop's index. For example:

```
for (i = 0; i < N; i++) {
   for (j = i+1; j < N; j++) {
      sequence of statements
   }
}
```

Now we can't just multiply the number of iterations of the outer loop times the number of iterations of the inner loop, because the inner loop has a different number of iterations each time. So let's think about how many iterations that inner loop has. That information is given in the following table:

Value of i	Number of iterations of inner loop
0	N
1	N-1
2	N-2
...	...
N-2	2
N-1	1

So we can see that the total number of times the sequence of statements executes is: N + N-1 + N-2 + ... + 3 + 2 + 1. the total is O(N^2).

5. Statements with function/method calls:

When a statement involves a function/method call, the complexity of the statement includes the complexity of the function/method call. Assume that you know that function/method f takes constant time, and that function/method g takes time proportional to (linear in) the value of its parameter k. Then the statements below have the time complexities indicated.

```
f(k);  // O(1)
g(k);  // O(k)
```

When a loop is involved, the same rule applies. For example:

```
for (j = 0; j < N; j++) {
        g(N);
}
```

Coding Ninjas

has complexity (N^2). The loop executes N times and each function/method call g(N) is complexity O(N).

Best-case and Average-case Complexity

Some functions/methods may require different amounts of time on different calls, even when the problem size is the same for both calls. For example, consider the add function/method that adds an item to the end of the Vector/list. In the worst case (the array is full), that function/method requires time proportional to the number of items in the Vector/list (because it has to copy all of them into the new, larger array). However, when the array is not full, add will only have to copy one value into the array, so in that case its time is independent of the length of the Vector/list; i.e. constant time.

In general, we may want to consider the best and average time requirements of a function/method as well as its worst-case time requirements. Which is considered the most important will depend on several factors. For example, if a function/method is part of a time-critical system like one that controls an airplane, the worst-case times are probably the most important (if the plane is flying towards a mountain and the controlling program can't make the next course correction until it has performed a computation, then the best-case and average-case times for that computation are not relevant -- the computation needs to be guaranteed to be fast enough to finish before the plane hits the mountain).

On the other hand, if occasionally waiting a long time for an answer is merely inconvenient (as opposed to life-threatening), it may be better to use an algorithm with a slow worst-case time and a fast average-case time, rather than one with so-so times in both the average and worst cases.

Note that calculating the average-case time for a function/method can be tricky. You need to consider all possible values for the important factors, and whether they will be distributed evenly.

When do Constants Matter?

Recall that when we use big-O notation, we drop constants and low-order terms. This is because when the problem size gets sufficiently large, those terms don't matter. However, this means that two algorithms can have the same big-O time

Coding Ninjas

complexity, even though one is always faster than the other. For example, suppose algorithm 1 requires N^2 time, and algorithm 2 requires 10 * N^2 + N time. For both algorithms, the time is O(N^2), but algorithm 1 will always be faster than algorithm 2. In this case, the constants and low-order terms do matter in terms of which algorithm is actually faster.

However, it is important to note that constants do not matter in terms of the question of how an algorithm "scales" (i.e., how does the algorithm's time change when the problem size doubles). Although an algorithm that requires N^2 time will always be faster than an algorithm that requires 10*N^2 time, for both algorithms, if the problem size doubles, the actual time will quadruple.

When two algorithms have different big-O time complexity, the constants and low-order terms only matter when the problem size is small. For example, even if there are large constants involved, a linear-time algorithm will always eventually be faster than a quadratic-time algorithm. This is illustrated in the following table, which shows the value of 100*N (a time that is linear in N) and the value of N^2/100 (a time that is quadratic in N) for some values of N. For values of N less than 104, the quadratic time is smaller than the linear time. However, for all values of N greater than 104, the linear time is smaller.

N	**100*N**	**N^2/100**
10^2	10^4	10^2
10^3	10^5	10^4
10^4	10^6	10^6
10^5	10^7	10^8
10^6	10^8	10^{10}
10^7	10^9	10^{12}

Coding Ninjas

For quick reference, you can refer this curated list of ready-to-use data structures and methods.

Vector

The most commonly used methods of vector are:

1. push_back()
   The run time of this method is O(1)
2. [] (bracket operators)
   The run time of this method is O(1)
3. size()
   The run time of this method is O(1)

```
vector<int> v;                     // v = {}
cout<< v.size() <<endl;            // outputs 0
v.push_back(20);                   // v = {20}
v.push_back(10);                   // v = {20, 10}
v.push_back(30);                   // v = {20, 10, 30}
cout << v[1] << endl;              // outputs 10 (since, vector is zero-indexed)
cout << v.size() << endl;          // outputs 3
```

Set

Set stores its elements in sorted order and doesn't contain duplicate values.
The most commonly used methods of set are:

1. insert()
   The run time of this method is O(log n)
2. find()
   The run time of this method is O(log n)
3. size()
   The run time of this method is O(1)

```
set<int> s;                        // s = {}
cout<< s.size() <<endl;            // outputs 0
s.insert(20);                      // s = {20}
s.insert(10);                      // s = {10, 20}
s.insert(10);                      // s = {10, 20}
auto it = s.find(10);              // it is an iterator that points to 10
cout << (it != s.end()? "FOUND" : "") << endl;     // outputs FOUND
cout << s.size() << endl;          // outputs 2
```

Unordered_Set

Unordered_Set is same as set and has the same most common methods. The only difference is run time complexity.
The most commonly used methods of set are:

1. insert()
   The run time of this method is O(1)
2. find()
   The run time of this method is O(1)
3. size()
   The run time of this method is O(1)

The unordered_set() achieves this complexity because it does not keep it in sorted order.

```
unordered_set<int> s;                  // s = {}
cout<< s.size() <<endl;                // outputs 0
s.insert(20);                          // s = {20}
s.insert(10);                          // s = {10, 20}
s.insert(10);                          // s = {10, 20}
auto it = s.find(10);                  // it is an iterator that points to 10
cout << (it != s.end()? "FOUND" : "") << endl;      // outputs FOUND
cout << s.size() << endl;              // outputs 2
```

Map
Map is very similar to set, but instead of storing an element or a value, it stores a key and a value. The commonly used methods are:

1. insert()
   The run time of this method is O(log n), insertion in map is done using make_pair
2. find()
   The run time of this method is O(log n), it returns pair of key and value to us.
3. size()
   The run time of this method is O(1)
4. [] bracket operators
   The run time of this method is O(log n), if the key exists, then it returns reference to the value. If the key doesn't exist, then it will do an insertion in the map

```
map<int, int> m;                                  // m = {}
cout<< m.size() << endl;                          // outputs 0
m.insert(make_pair(20, 1));                       // m = {(20, 1)}
m.insert(make_pair(10,1));                        // m = {(10, 1), (20, 1)}
m[10]++;                                          // m = {(10, 2), (20, 1)}
auto it = m.find(10);                             // it is an iterator that points to
(10, 2)
cout << (it != m.end() ? it -> second: 0)) << endl;// outputs 2
```

```
auto it2 = m.find(20);                          // it is an iterator that points to
(20, 1)
cout << (it2 != m.end() ? it2 -> first: 0)) << endl;//outputs 20
cout<< m.size() << endl;                        //outputs 2
```

Unordered_Map

Unordered_Map shares the same relationship with Map as unordered_set shared with set. So, similarly the only difference lies in the run-time complexity and it is because map keeps key-value pairs in sorted order, while the unordered_map keeps the key-value pair in any order. The commonly used methods are:

1. insert()
   The run time of this method is O(1), insertion in map is done using make_pair
2. find()
   The run time of this method is O(1), it returns pair of key and value to us.
3. size()
   The run time of this method is O(1)
4. [] bracket operators
   The run time of this method is O(1), if the key exists, then it returns reference to the value. If the key doesn't exist, then it will do an insertion in the map

```
unordered_map<int, int> m;                                  // m = {}
cout<< m.size() << endl;                        // outputs 0
m.insert(make_pair(20, 1));                     // m = {(20, 1)}
m.insert(make_pair(10,1));                      // m = {(10, 1), (20, 1)} (this
could be in any order)
m[10]++;                                        // m = {(10, 2), (20, 1)} (this
could be in any order)
auto it = m.find(10);                           // it is an iterator that points to
(10, 2)
cout << (it != m.end() ? it -> second: 0)) << endl;// outputs 2
auto it2 = m.find(20);                          // it is an iterator that points to
(20, 1)
cout << (it2 != m.end() ? it2 -> first: 0)) << endl;//outputs 20
cout<< m.size() << endl;                        //outputs 2
```

For quick reference, you can refer this curated list of ready-to-use data structures and methods.

1) **ArrayList:**

```
ArrayList<Integer> al = new ArrayList<>();
```

1. al.add(5);

   The run time of this method is O(1).
2. al.get(index);

   The run time of this method is O(1).
3. al.size();

   The run time of this method is O(1).

```
al.add(5);                        //adds 5 at the last position
al.add(10);                       //adds 10 at the last position
al.add(20);                       //adds 20 at the last position
System.out.println(al.get(0));    //Prints 5 on new line
System.out.println(al.size());    //Prints the size i.e. 3 on new line
```

2) **HashMap:**

It is Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

```
HashMap<Character, Integer> hm = new HashMap<>();
```

1. hm.put(key, value);

   Puts the value of particular key in the hashmap. Overwrite previous value if already present. Operation done in O(1) time.
2. hm.get(key);

   Returns the value of particular key in O(1) time.
3. hm.containsKey(key);

   Returns true if this map contains a mapping for the specified key. The run time is O(1).
4. hm.isEmpty();

   Returns true if this map contains no key-value mappings.

```
hm.put(‘a’, 1);                   //Puts value 1 for character ‘a’.
hm.put('b', 3);                          //Puts value 3 for character 'b'.
System.out.println(hm.get('b'));         //Prints value for 'b' i.e. 3 on new line
System.out.println(hm.containsKey('a')); //Prints true
System.out.println(hm.isEmpty());        //Prints false
```

3) **HashSet:**

This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element.

```
HashSet<Integer> hs = new HashSet<>();
```

1. hs.add(3);

Adds the specified element to this set if it is not already present. Done in O(1) time.

2. hs.contains(1);

Returns true if this set contains the specified element. Run time is O(1).

3. hs.size();

Returns the number of elements in this set (its cardinality). Run time is O(1).

4. hs.isEmpty();

Returns true if this set contains no elements.

```
hs.add(1);                               //Adds 1 to the hashset
hs.add(2);                               //Adds 2 to the hashset
System.out.println(hs.contains(1));      //Prints true on the new line.
System.out.println(hs.size());           //Prints 2 on the new line.
System.out.println(hs.isEmpty());        //Prints false on new line.
```

4) **TreeMap**

A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations.

```
TreeMap<Integer, Integer> tm = new TreeMap<>();
```

1. tm.put(key, value);

Puts the value of particular key in the treemap. Overwrite previous value if already present. Operation done in O(log n) time.

2. tm.get(key);

Returns the value of particular key in O(log n) time.

3. tm.containsKey(key);

Returns true if this map contains a mapping for the specified key. The run time is O(log n).

4. tm.isEmpty();

Returns true if this map contains no key-value mappings.

```
tm.put(3, 1);                            //adds value 1 for key 3 in the map
tm.put(4, 2);                            //adds value 2 for key 4 in the map
tm.put(2, 8);                            //adds value 8 for key 2 in the map
System.out.println(tm.get(3));           //Prints value 1 on the new line.
System.out.println(tm.containsKey(1));   //Prints false on new line
System.out.println(tm.isEmpty());        //Prints false on new line
```

Note: TreeMap always keeps the elements in a sorted(increasing) order, while the elements in a HashMap have no order. TreeMap also provides some cool methods for first, last, floor and ceiling of keys.

5) **PriorityQueue:**

An unbounded priority queue based on a priority heap. The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used. A priority queue does not permit null elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects.

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
```

1. pq.add(1);

Inserts the specified element into this priority queue. Done in O(log n) time.

2. pq.remove();

Retrieves and removes the head of this queue. This method differs from poll only in that it throws an exception if this queue is empty. Run time is O(log n).

3. pq.peek();

Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty. Run time is O(1).

4. pq.isEmpty();

Returns true if this collection contains no elements.

```
pq.add(10);                         //Adds 10 to the queue
pq.add(12);                         //Adds 12 to the queue
pq.add(2);                          //Adds 2 to the queue
System.out.println(pq.remove());    //Prints 2 and remove it from the queue
System.out.println(pq.peek());      //Prints 10 on new line
System.out.println(pq.isEmpty());   //Prints false in new line
```

For quick reference, you can refer this curated list of ready-to-use data structures and methods.

List

A list is a collection which is ordered and changeable. In Python lists are written with square brackets.

```
list = ["Apple", "Banana", "Cherry"]
```

1. You access the list items by referring to the index number:

   ```
   print(list[1])       //  Start with 0th index so Output is Banana
   ```

2. To change the value of a specific item, refer to the index number:

   ```
   list[1] = "Orange"
   ```

3. You can loop through the list items by using a `for` loop:

   ```
   for x in list:
         print(x)          // Output is Apple , Orange , Cherry
   ```

4. To determine if a specified item is present in a list use the `in` keyword:

   ```
   if "Apple" in list:

         print ("Yes")                // Yes if Apple is present in list

   else:

         print("No")                  // No  if it is not Present
   ```

5. To determine how many items a list have, use the `len()` method:

   ```
   print(len(list))    // Output is 3 as contains 3 elements
   ```

6. To add an item to the end of the list, use the `append()` method:

```
list.append("Mango")    // Append at the end of list
```

7. To add an item at the specified index, use the `insert()` method:

```
list.insert(1, "Mango")  // insert at index 1 of list
```

8. The `remove()` method removes the specified item:

```
list.remove("Banana")         // Remove the element Banana if present
```

9. The `pop()` method removes the specified index, (or the last item if index is not specified)

```
list.pop()
```

10.The `del` keyword removes the specified index:

```
del list[0]  // removes the specified index
```

Tuple

A tuple is a collection which is ordered and **unchangeable**. In Python tuples are written with round brackets.

```
tuple = ("Apple", "Banana", "Cherry")
```

1. You can access tuple items by referring to the index number, inside square brackets:

```
print(tuple[1])  // Output is Banana the specified index
```

2. Once a tuple is created, you cannot change its values. Tuples are **unchangeable**.

```
tuple[1] = "Orange" // Gives error the value remain unchanged
```

3. You can loop through the tuple items by using a `for` loop.

```
for x in tuple:
        print(x)          // Generate all element present in tuple
```

4. To determine if a specified item is present in a tuple use the `in` keyword:

```
if "Apple" in tuple:
        print("Yes")          // Output is Yes if Apple is present in tuple
```

5. To determine how many items a list have, use the `len()` method:
   ```
   print(len(tuple)) // Output is 3 as 3 element are in tuple
   ```

6. Tuples are **unchangeable**, so you cannot add or remove items from it, but you can delete the tuple completely:

7. Python has two built-in methods that you can use on tuples.

count()	Returns the number of times a specified value occurs in a tuple

index()	Searches the tuple for a specified value and returns the position of where it was found

Set

A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets.

```
set = {"apple", "banana", "cherry"}
```

1. You cannot access items in a set by referring to an index, since sets are unordered the items has no index. But you can loop through the set items using a `for` loop, or ask if a specified value is present in a set, by using the `in` keyword.

   ```
   for x in set:
     print(x)      // Output contains all element present in set
   ```

2. Once a set is created, you cannot change its items, but you can add new items.

   To add one item to a set use the `add()` method.

   ```
        set.add("Orange")                  // Add one element at end
   ```

   To add more than one item to a set use the `update()` method.

   ```
        set.update(["Orange", "Mango", "Grapes"])   // Add all
   ```

   // element in the end

3. To determine how many items a set have, use the `len()` method.

   ```
   print(len(set))       // output is length of set
   ```

4. To remove an item in a set, use the `remove()`, or the `discard()` method.

   ```
   set.remove("Banana") //Remove element if present else raise error

   set.discard("Banana") // Remove element if present else don’t

                         // raise error
   ```

5. Remove last element by using `pop()` method:

   ```
   x = set.pop()    //Remove and Return last element from the set

   print(x)         // print the last element of set
   ```

Dictionary

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

```
dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

1. You can access the items of a dictionary by referring to its key name, inside square brackets:

```
x = dict["model"]            // Return the value of the key
```

2. You can change the value of a specific item by referring to its key name:

   ```
   dict["year"] = 2018
   ```

3. You can loop through a dictionary by using a `for` loop. When looping through a dictionary, the return value are the *keys* of the dictionary, but there are methods to return the *values* as well.

   ```
   for x in dict:
         print(x)         // Print all key names in the dictionary

   for x in dict:
         print(dict[x])  // Print all values of the dictionary

   for x, y in dict.items():
         print(x, y)       // Print both keys and value of the dictionary
   ```

4. Adding an item to the dictionary is done by using a new index key and assigning a value to it:

```
dict["color"] = "red"
      print(dict)             // Add new key and value to dictionary
```

5. The `pop()` method removes the item with specified key name:

```
dict.pop("model")       // Removes model key/value pair in dictionary
```

Note: Rat in a Maze - Code Correction

Initialisation of solution 2D array

The code developed in the video may not work on your local IDE. This happens because the solution 2D array is not initialized with default value, before making recursive calls. Hence, to resolve it, please initialize the 2D-array, before initiating the recursive calls. The following code helps you understand where the initialization of solution 2D array has to be done:

```
void ratInAMaze(int maze[][20], int n){

  int** solution = new int*[n];
  for(int i=0;i<n;i++){
      solution[i] = new int[n];
  }
  // initialization of solution 2D arrays goes here.

  mazeHelp(maze,n,solution,0,0);

}
```

Complete code with initialisation of solution 2D array:

```
#include<bits/stdc++.h>
using namespace std;


void printSolution(int** solution,int n){
      for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                  cout << solution[i][j] << " ";
            }
      }
      cout<<endl;
}
void mazeHelp(int maze[][20],int n,int** solution,int x,int y){
      if(x == n-1 && y == n-1){
            solution[x][y] =1;
            printSolution(solution,n);
            solution[x][y] =0;
            return;
      }
      if(x>=n || x<0 || y>=n || y<0 || maze[x][y] ==0 || solution[x][y]
==1){
            return;
      }
      solution[x][y] = 1;
      mazeHelp(maze,n,solution,x-1,y);
      mazeHelp(maze,n,solution,x+1,y);
      mazeHelp(maze,n,solution,x,y-1);
      mazeHelp(maze,n,solution,x,y+1);
      solution[x][y] = 0;
}
void ratInAMaze(int maze[][20], int n){

  int** solution = new int*[n];
  for(int i=0;i<n;i++){
      solution[i] = new int[n];
  }
  // Initialization of solution 2D array with 0
  for(int i=0; i<n; i++){
    memset(solution[i], 0, n*sizeof(int));
  }

  mazeHelp(maze,n,solution,0,0);

}
```