**Week 6 (Recursion) *– Summary***

**Recursion** is an ***extremely powerful*** programming technique employed to solve problems.

* Breaks a problem into ***smaller but similar*** problem size which may be easier to solve
* At the end of the ‘division’ of problem -> a smallest problem whereby answer is trivial
* Using that answer enables you to solve the previous problems

In programming, recursion takes the form of a ***function/method that calls itself***

Each time function ***calls a copy of itself*** to work on smaller problem

***->* *Recursive Call / Recursive Step***

**Points to note when using recursion**

1. How to define the problem in terms of a **smaller and similar problem**?
2. What instance of the problem serve as **base case**?
3. Does each **recursive call** reduce the size of the problem?
4. As the problem size reduces, **will it reach the base case**?

**Examples of Recursion**

|  |
| --- |
| // compute (and return) the factorial of a number  **int fac(int n)**  {  if (n == 0) // base case  return 1;  else // recursive step  return n \* **fac(n-1)**;  } |

|  |
| --- |
| // compute (and return) the fibonacci number  **int fib(int n)**  {  if (n == 0 || n == 1) // base case  return n;  else // recursive step  return **fib(n-2)** + **fib(n-1)**;  } |

|  |
| --- |
| // compute (and return) the sum of an array of numbers  int sum(int array[], int n)  {  if (n == 0) // base case (no more data)  return 0;  else // recursive step  return array[n-1] + **sum(array, n-1)**;  } |

|  |
| --- |
| // find (and return) the highest value in an array (version 1)  int maxArray(int array[], int n)  {  if (n == 1) // base case  return array[n-1];  else // recursive step  return max(array[n-1], **maxArray(array, n-1)**);  }  // find (and return) the highest value in an array (version 2)  int maxArray(int array[], int start, int end)  {  if (start == end) // base case  return array[start];  else // recursive step  {  int mid = (start + end) / 2; // divide into 2 halves  return max(maxArray(array, start, mid), maxArray(array, mid+1, end));  }  } |

#define max(x, y) ((x > y)? x : y)

|  |
| --- |
| // print the numbers in an array in forward sequenece  void printForward(int array[], int n)  {  if (n > 0) // recursive step  {  printForward**(array, n-1)**; // print remaining numbers forward  cout << array[n-1] << " "; // print the number  }  // base case - do nothing  } |

|  |
| --- |
| // print the numbers in an array in reverse sequenece  void printBackward(int array[], int n)  {  if (n > 0) // recursive step  {  cout << array[n-1] << " "; // print the number  **printBackward(array, n-1)**; // print remaining numbers backwards  }  // base case - do nothing  } |

|  |
| --- |
| // Tower of Hanoi  **void moverDisks(int n, string source, string destination, string spare)**  {  if (count == 1) // base case  cout << "Move top disk from : " << source << " to " << destination << endl;  else // recursive step  {  moverDisks(n-1, source, spare, destination);  moverDisks(1, source, destination, spare);  moverDisks(n-1, spare, destination, source);  }  } |

e.g. moverDisks(3, "Pole 1", "Pole 2", "Pole 3");

**Tracing a Recursion Function**

A ***box trace*** is a systematic way to trace the actions of a recursive function, very useful for debugging recursive functions.

* Corresponds to an ***activation record***
* Contains a function’s *local environment/ context* at the time of and result of call to the function.
* When a call is made to a function, we say the context/ local environment of current function is switched to new one.
* During *context switching*, previous context must be saved so that it can be reinstated upon return from function call.

E.g.



