**Coding Fundamentals: Bit Manipulation**

**Part 0 – Bitwise Operators:**

|  |  |
| --- | --- |
| Operator | Description |
| & (and) | **Binary AND Operator:** returns a bit to the result if it exists in both operands. |
| | (or) | **Binary OR Operator:** returns a bit if it exists in either operand. |
| ^ (x-or) | **Binary XOR Operator:** returns the bit if it is set in one operand but not both. |
| ~ (compliment) | **Binary Ones Complement:** Flips each bit. |
| << (left shift) | **Binary Left Shift Operator:** The left operands value is moved left by the number of bits specified by the right operand. |
| >> (right shift) | **Binary Right Shift Operator:** The left operands value is moved right by the number of bits specified by the right operand. The most left bit becomes the previous most left bit. This is to preserve the sign. |
| >>> (zero fill right shift) | **Shift right zero fill operator:** The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros. |

**Part 1 – Common Components in Bit Manipulation:**

|  |  |
| --- | --- |
| **Common Component** | **Description** |
| (1 << index) | **General Index Mask:** It returns a bit vector of all 0’s except at the index specified by ‘index’. |
| (1 << index) - 1 | **Clear MSB Through Index Mask:** It returns a bit vector such that indexes (MSB, index) are set to zero and indexes (index – 1, 0) are set to 1. |
| ~ ((1 << (index +1)) - 1); | **Clear Index Through Zero Mask:** It returns a bit vector such that indexes (MSB, index + 1) are set to 1 and indexes (index, 0) are set to 0. |
| (1 << i) | (1 << j) | **Set Multiple Indexes Mask:** Given two indexes (i and j) this will return a bit vector with all index set to zero except at indexes (i and j) which are set to one. Keep this pattern for more than two indexes. For example, (1 << i) | (1 << j) | (1 << k) would set three bits. |

**Note:** For an integer, MSB (most significant bit) is 31.

**Visualizing the Complex Common Components:**

|  |
| --- |
| **Example of ‘**clearBitsMSBthroughI**’ mask with index = 2:**  (1 << 2) yields **🡪 [00.. 00 0100]**  (1 << 2 - 1) yields 🡪 **[00.. 00 0011]**  **Example of** ‘clearBitsIthrough0’ **mask with index = 2:**  (1 << 3) yields -> **[00.. 00 1000]**  (1 << 3) - 1 yields -> **[00..00 0111]**  ~ ((1 << 3) - 1) -> **[11..11 1000]** |

**Part 2 – Bit Manipulation Fundamental Methods:**

|  |
| --- |
| **public** **class** BitManipulationFundamentals {  **public** **static** **boolean** getBit(**int** bitVector, **int** index) {  **int** mask = (1 << index);  **return** (bitVector & mask) != 0;  }  **public** **static** **int** setBit(**int** bitVector, **int** index) {  **int** mask = (1 << index);  **return** bitVector | mask;  }  **public** **static** **int** clearBit(**int** bitVector, **int** index) {  **int** mask = (1 << index);  **return** bitVector & ~mask;  }    **public** **static** **int** toggleBit(**int** bitVector, **int** index){  **int** mask = (1 << index);  **return** bitVector ^= mask;  }  **public static int** clearBitsMSBthroughI**(int** bitVector, **int** index) {  **int** mask = (1 << index) - 1;  **return** bitVector & mask;  }  **public** **static** **int** clearBitsIthrough0(**int** bitVector, **int** index) {  **int** mask = ~ ((1 << (index +1)) - 1);  **return** bitVector & mask;  }  } |

* **Toggle Bit:**

1. Remember that if you “x-or” two components, together, it returns 1 if the two components are different.
2. If you x-or any bit called ‘bit’ with zero, the result is unchanged. This is because if you “x-or” a zero with a zero, the result is zero and if you “x-or” a one with a zero, the result is 1. In both cases, the output is equal to the input.
3. If you “x-or” any input bit called ‘bit’ with a one, it will toggle that bit. If you “x-or” a zero with a one, the result is one. If you “x-or” a one with a one, the result is one. In both cases. The output is the input after being toggled.
4. If you “x-or” a set bit mask and an input bit vector, this will toggle the input index and leave the rest unchanged.

* The ‘clearBitsMSBthroughI’ and ‘clearBitsIthrough0’ both get their corresponding mask and “and” it with the bit vector to get the desired result.

**Part 2 – Setting, Clearing and Toggling Multiple Bits at a Time:**

Instead of iterating over all the bits you need to set, you can create a mask that performs the operations on all bits at once. Below are examples.

**public** **static** **int** set3Bits(**int** bitVector, **int** i, **int** j, **int** k){

**int** multiIndexMask = ( 1 << i) | (1 << j) | (1 << k);

**return** bitVector | multiIndexMask;

}

**public** **static** **int** clear3Bits(**int** bitVector, **int** i, **int** j, **int** k){

**int** multiIndexMask = ( 1 << i) | (1 << j) | (1 << k);

**return** bitVector & ~ multiIndexMask;

}

**public** **static** **int** clearBits(**int** bitVector, List<Integer> indexesToSet){

**int** multiIndexMask = 0;

**for**(Integer index : indexesToSet){

multiIndexMask |= ( 1 << index);

}

**return** bitVector & ~ multiIndexMask;

}

**Part 3 – Skills to Master:**

|  |
| --- |
| **Skill To Master** |
| 1. Traversing forward, backward, and tree traversals. |
| 1. Retrieving the item at the first, last or a particular index |
| 1. Inserting an item at the first, last or a particular index. |
| 1. Removing an item at the first, last or a particular index. |
| 1. Linear Searchig and Binary Searching |
| 1. Basic Sort (Bubble Sort) |
| 1. Complex Sort (i.e. Quick Sort) |
| 1. Complete Re-arrange |

The only ones that are applicable for bit vectors are traversing.

|  |
| --- |
| **Traversing:**    **Traversing Forward:**  **public** **static** **void** traverseForward(**int** bitVector) {  **final** **int** LENGTH = 32;  **for** (**int** i = 0; i < LENGTH; i++) {  }  }  **Traversing Backward:**  **public** **static** **void** traverseBackward(**int** bitVector) {  **final** **int** LENGTH = 32;  **for** (**int** i = LENGTH - 1; i >= 0; i--) {  }  }  **Traversing Inward from Edges:**  **public** **static** **int** traverseInward(**int** bitVector){  **int** LENGTH = 32;  **for**(**int** i = 0; i < (LENGTH / 2) ; i++){  **int** j = (LENGTH - 1) - i;  }  }    Image of Traversing Inward |

**Common Operations:**

|  |
| --- |
| **public** **static** **short** countSetBits(**int** bitVector) {  **final** **int** LENGTH = 32;  **short** numberOfBits = 0;  **for** (**int** i = 0; i < LENGTH; i++) {  **if**(*getBit*(bitVector, i)){  numberOfBits++;  }  }  **return** numberOfBits;  }    **public** **static** **boolean** bitsDiffer(**int** bitVector, **int** i, **int** j) {  **return** (bitVector & (1 << i)) != (bitVector & (1 << j));  }  **public** **static** **int** swapBits(**int** bitVector, **int** i , **int** j) {  **if**(*bitsDiffer*(bitVector, i, j)){  **int** mask = (1 << i) | (1 << j);  bitVector ^= mask;  }    **return** bitVector;  }  **public** **static** **int** reverse(**int** bitVector){  **int** LENGTH = 32;  **for**(**int** i = 0; i < (LENGTH / 2) ; i++){  **int** j = (LENGTH - 1) - i;  bitVector = *swapBits*(bitVector, i, j);  }  **return** bitVector;  } |

* The bits differ method uses the same logic as the get bit method.
* The toggle bits method check if the two bits differ. If they are, it toggles both bits at once using a multiple index mask.
* To reverse the bit vector, you traverse inward and swap the two elements.