Test

Problem J4/S2: Fraction Action

Many advanced calculators have a fraction feature that will simplify fractions for you.
You are to write a program that will accept for input a non-negative integer as a numerator and a positive integer as a denominator, and output the fraction in simplest form. That is, the fraction cannot be reduced any further, and the numerator will be less than the denominator. You can assume that all input numerators and denominators will produce valid fractions.

Examples

Input	**Input**	**Input**	**Input**
28 7	13 5	0 7	55 10
Output	**Output**	**Output**	**Output**
4	2 3/5	0	5 1/2

- Greatest Common Divisor

Problem J4/S2: Fraction Action

Input	Input	Input	Input
28 7	13 5	0 7	55 10
Output	**Output**	**Output**	**Output**
4	2 3/5	0	5 1/2

- Greatest Common Divisor
- 13/5
- 1st: 13%5 = 3 -> not divisible
- 2nd: whole = (int)13/5 = 2
- 3rd: GCF = 1 -> 13/5
- 4th: 13%5 = 3
- 5th: whole (2) remainder(3)/denominator

- N = 55 / D = 10
- 1st: 55%10 = 5 -> 55/10 = 5.xyz -> whole = 5
- 2nd: GCF = 5 -> N = 11/ D = 2
- 3rd: 11%2 = 1
- 4th: 5 1/2

```
//deteremine if divisible
If (ans == decimal){
   //Find GCF
   //Simplify
   //Find remainder
   //print
}
```

Problem J4/S2: Fraction Action

Input	Input	Input	Input
28 7	13 5	0 7	55 10
Output	**Output**	**Output**	**Output**
4	2 3/5	0	5 1/2

- 0/7
- 1st: 0/7 = 0 (no decimal)
- 2nd: 0

- 28/7
- 1st: 28/7 = 4 (integer)
- 2nd: 4

```
If (ans == 0 || ans == integer){
   print ans;
}
```

```
void fraction(){
    //declaration and input
    int denominator = 0, numerator = 0, whole = 0, simplifiedN = 0, simplifiedD = 0, GCF = 1, remainder = 0;
    cin >> numerator >> denominator;
    /* IN JAVA
    Scanner input = new Scanner(system.in);
    numerator = input.nextInt();
    denominator = input.nextInt();
    */
}
```

```
void fraction(){
    //declaration and input
    int denominator = 0, numerator = 0, whole = 0, simplifiedN = 0, simplifiedD = 0, GCF = 1, remainder = 0;
    cin >> numerator >> denominator;

    //divisible?
    if (numerator%denominator == 0){
            //Case 0 or integer
            cout<<numerator/denominator<<endl;
    }else{
          //find whole and GCF
          //simplify
          //find remainder
          //print
    }
```

```
void fraction(){
    //declaration and input
    int denominator = 0, numerator = 0, whole = 0, simplifiedN = 0, simplifiedD = 0, GCF = 1, remainder = 0;
    cin >> numerator >> denominator;

    //divisible?
    if (numerator%denominator == 0){
            //Case 0 or integer
            cout<<numerator/denominator<<endl;
    }else{
            //find whole and GCF
            whole = (int)(numerator/denominator);
            //find greatest common factor
            for (int i = min(numerator, denominator); i > 0; i--){
                    if (numerator % i == 0 && denominator % i == 0){
                            GCF = i;
                            break;
                    }
            }
            //simplify
            //find remainder
            //print
    }
```

```
5 10
5 4 3 2 1
```

```
void fraction(){
        //find whole and GCF
        whole = (int)(numerator/denominator);
        //find greatest common factor
        for (int i = min(numerator, denominator); i > 0; i--){
                if (numerator % i == 0 && denominator % i == 0){
                        GCF = i;
                         break;
                }
         }
        //simplify
        simplifiedN = numerator/GCF;
        simplifiedD = denominator/GCF;

        //find remainder
        remainder = simplifiedN/simplifiedD;
        //print
        if (whole == 0){
            //print in one format
        }else{//print in the other format}
   }
```

Problem S2: Multiple Choice

Your teacher likes to give multiple choice tests. One benefit of giving these tests is that they are easy to mark, given an answer key. The other benefit is that students believe they have a one-in-five chance of getting the correct answer, assuming the multiple choice possibilities are A, B, C, D or E. Write a program that your teacher can use to grade one multiple choice test.

Input Format

The input will contain the number N ($0 < N < 10000$) followed by $2N$ lines. The $2N$ lines are composed of N lines of student responses (with one of A, B, C, D or E on each line), followed by N lines of correct answers (with one of A, B, C, D or E on each line), in the same order as the student answered the questions (that is, if line i is the student response, then line $N + i$ contains the correct answer to that question).

Output Format

Output the number of questions the student answered correctly.

Sample Cases

Input	Input
3 A B C A C B	3 A A A A B A
Output	**Output**
1	2

```
void multipleChoice(){
    int numOfQ = 0, correctCount = 0;
    cin>>numOfQ;
    char student[numOfQ];
    char answer[numOfQ];

    for(int i = 0; i < numOfQ; i++){
        cin >> student[i];
    }

    for(int i = 0; i < numOfQ; i++){
        cin >> answer[i];
    }

    for(int i = 0; i < numOfQ; i++){
        if(student[i] == answer[i])
            correctCount++;
    }
    cout<<correctCount<<endl;
}
```

Input/output

C++:

```
cin >> int >> char;
```

Java:

```
Scanner input = new Scanner (system.in);
input.nextInt();
input.nextInt(); //take in next Integer
input.nextLine();//will take in a new line char
('\n')
Input.nextLine();//will take in next line
```

Problem C: Pattern Generator

Write a program that repeatedly reads two numbers *n* and *k* and prints all bit patterns of length *n* with *k* ones in **descending order** (when the bit patterns are considered as binary numbers). You may assume that $30 \geq n > 0$, $8 > k \geq 0$, and $n \geq k$. The first number in the input gives the number of pairs *n* and *k*. The numbers *n* and *k* are separated by a single space. Leading zeroes in a bit pattern should be included. See the example below.

Sample Input

```
3
2 1
2 0
4 2
```

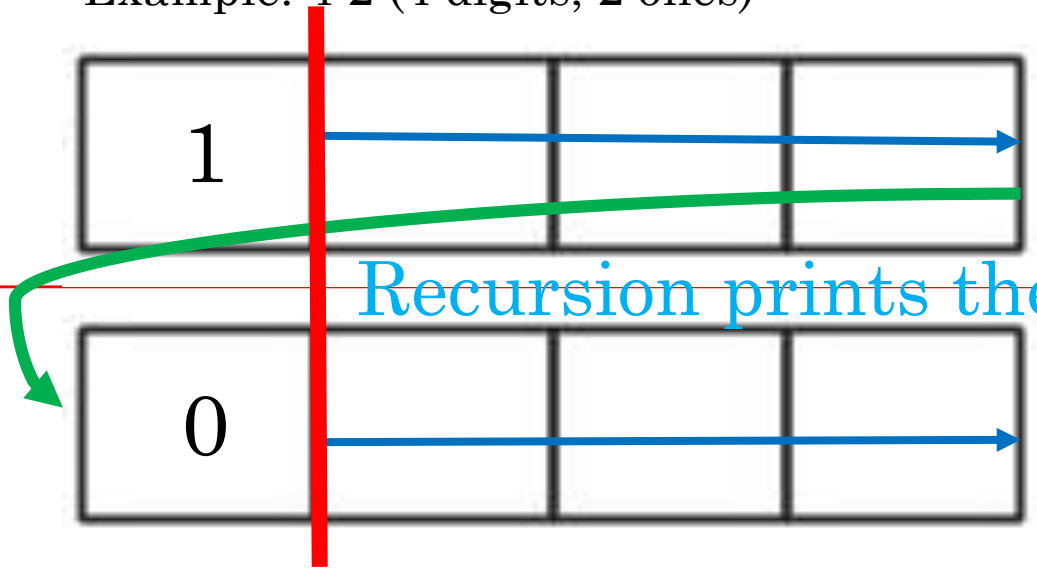
Sample Output

```
The bit patterns are
10
01
The bit patterns are
00
The bit patterns are
1100
1010
1001
0110
0101
0011
```

```
void pattern(){
    cin >> numPair;
    int pair[numPair*2];

    //take in all pairs
    for(int i = 0; i < numPair*2; i = i + 2){
        cin >> pair[i] >> pair[i+1]; //pair[i] = n; pair[i+1] = k
    }

    //for each pair, pring the array for them
    for (int i = 0; i < numPair*2; i = i + 2){
        cout<<endl;
        cout<<"The bit patterns are"<<endl;
        n = pair[i];
        k = pair[i+1];
        printarray(n,k);
    }

}
```

Understand Recursion ➔ print(4,2)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 1 = 3 2 – 1 = 1
➔print(3,1); //print 3 digits with 1 one

Recursion prints the rest for me

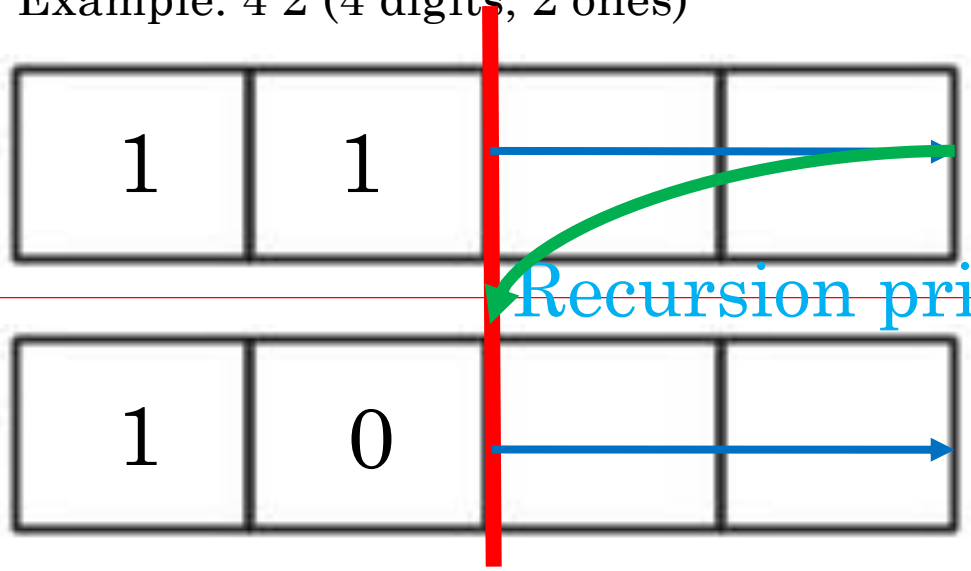
Digit Left(n) Ones left(k)
4 – 1 = 3 2 – 0 = 2
➔print(3,2); //print 3 digits with 2 ones

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔print(3,1)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 2 = 2 2 – 2 = 0
➔print(2,0); //print 2 digits with 0 one

Recursion prints the rest for me

Digit Left(n) Ones left(k)
4 – 2 = 2 2 – 1 = 1
➔print(2,1); //print 2 digits with 1 one

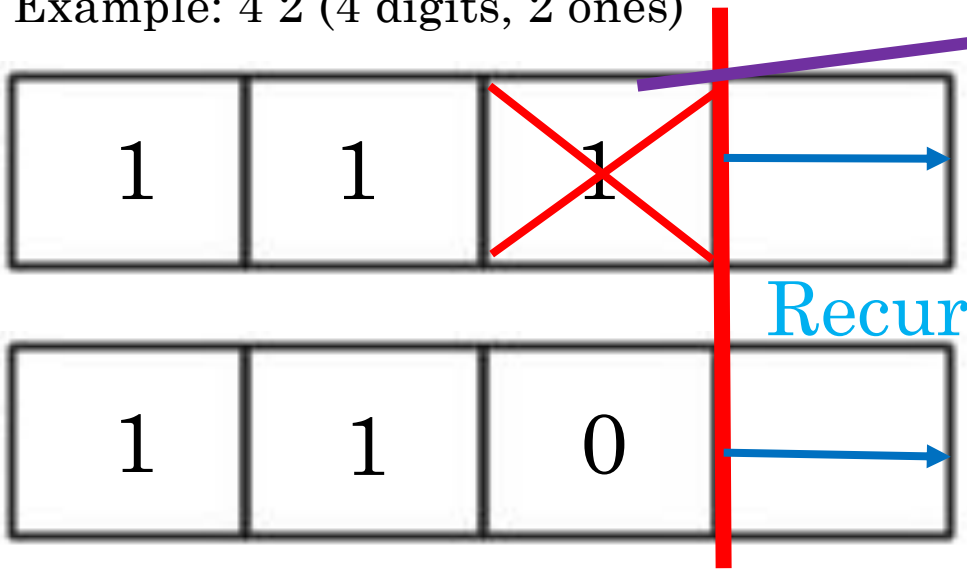
1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion➔print(2,0)

Example: 4 2 (4 digits, 2 ones)

Condition: if k > 0, then print 1



Recursion prints the rest for me

Digit Left(n) Ones left(k)

4 – 3 = 1 2 – 2 = 0

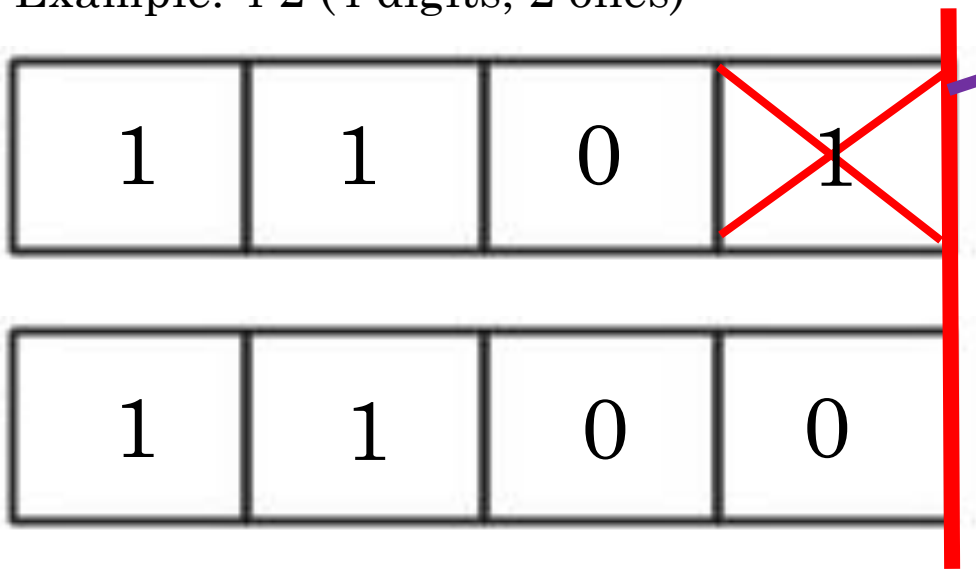
➔print(1,0); //print 1 digits with 0 one

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion➔print(1,0)

Example: 4 2 (4 digits, 2 ones)



Condition not met: k needs to be > 0

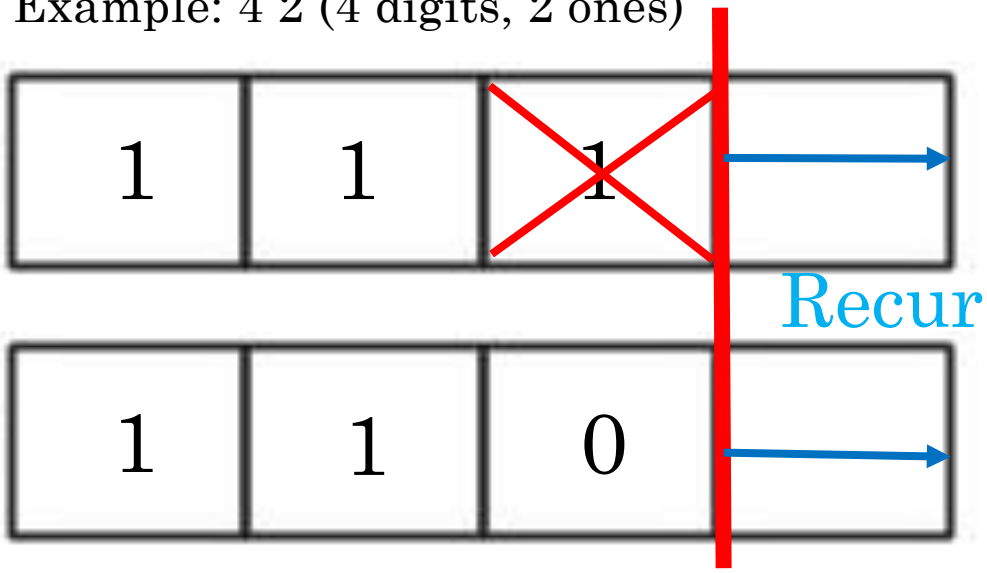
Digit Left(n) Ones left(k)
4 – 4 = 0 2 – 2 = 0
➔print(0,0); //print 0 digits with 0 one ➔
STOP ➔ print this array

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion➔print(2,0)

Example: 4 2 (4 digits, 2 ones)



Recursion prints the rest for me

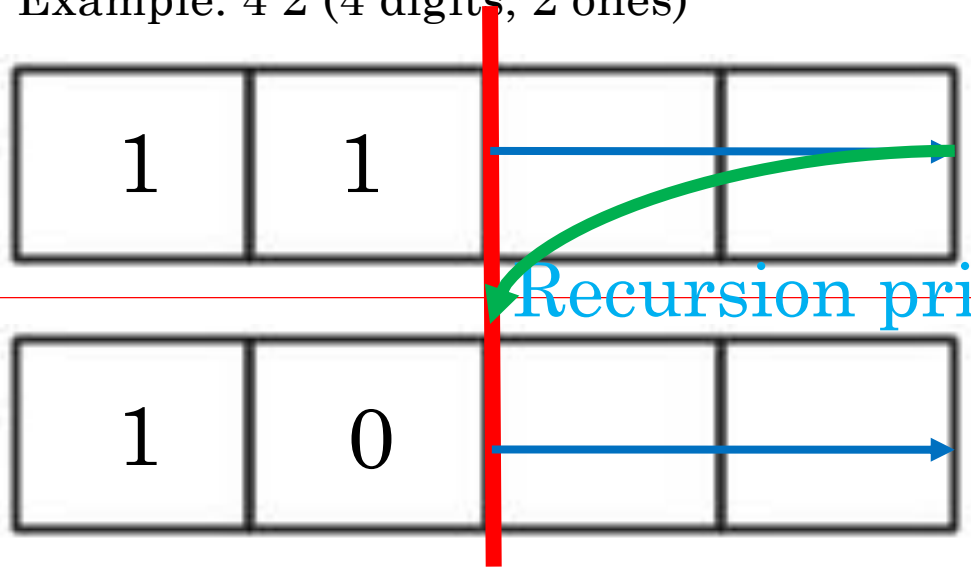
Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 2 = 0
➔print(1,0); //print 1 digits with 0 one

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔print(3,1)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 2 = 2 2 – 2 = 0
➔print(2,0); //print 2 digits with 0 one

Recursion prints the rest for me

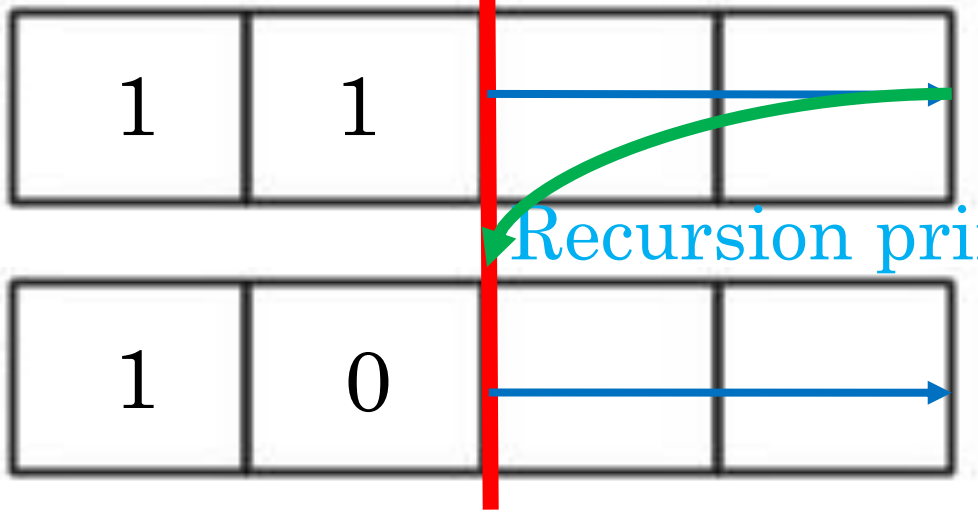
Digit Left(n) Ones left(k)
4 – 2 = 2 2 – 1 = 1
➔print(2,1); //print 2 digits with 1 one

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔print(3,1)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n)	Ones left(k)
4 – 2 = 2	2 – 2 = 0

➔print(2,0); //print 2 digits with 0 one

Done

Recursion prints the rest for me

Digit Left(n)	Ones left(k)
4 – 2 = 2	2 – 1 = 1

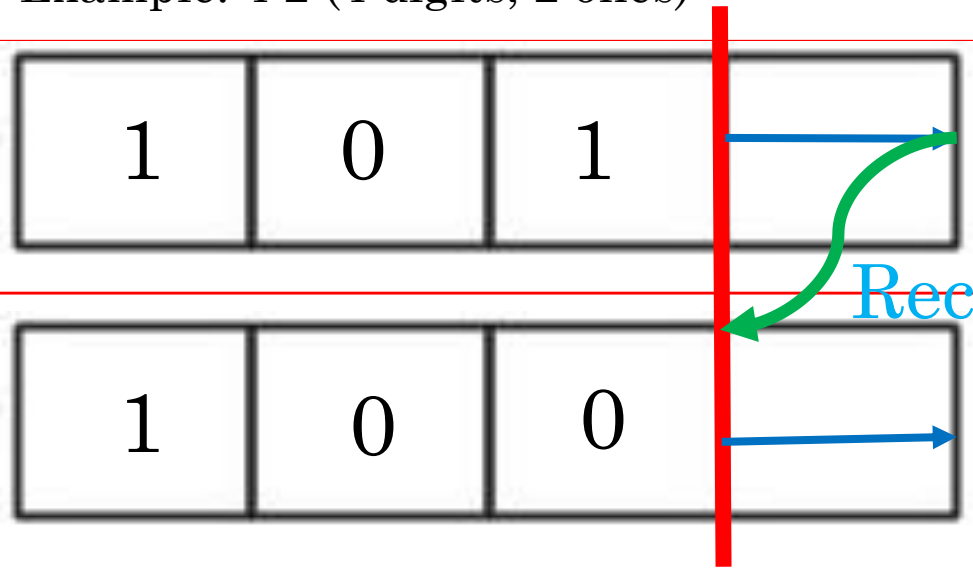
➔print(2,1); //print 2 digits with 1 one

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔print(2,1)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 2 = 0
➔print(1,0); //print 1 digit with 0 one

Recursion prints the rest for me

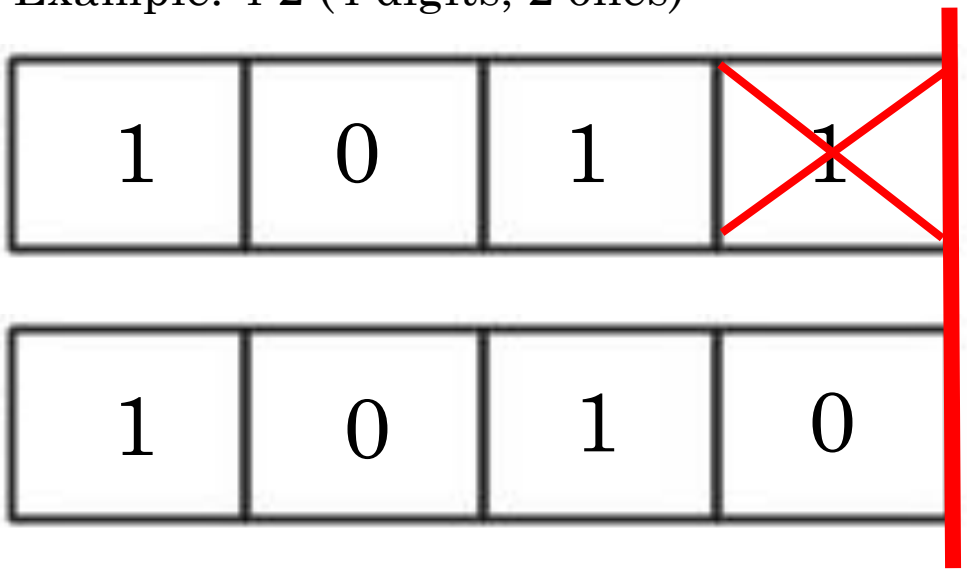
Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 1 = 1
➔print(1,1); //print 1 digit with 1 one

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔print(1,0)

Example: 4 2 (4 digits, 2 ones)



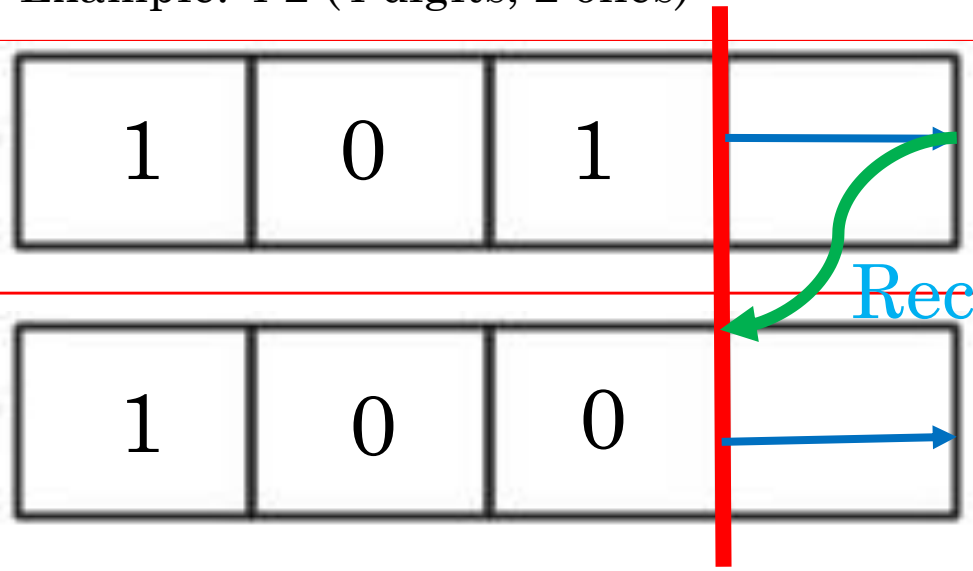
Digit Left(n) Ones left(k)
4 – 4 = 0 2 – 2 = 0
➔print(0,0); //print 0 digits with 0 one ➔
STOP ➔ print this array

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔print(2,1)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 2 = 0
➔print(1,0); //print 1 digit with 0 one

Recursion prints the rest for me

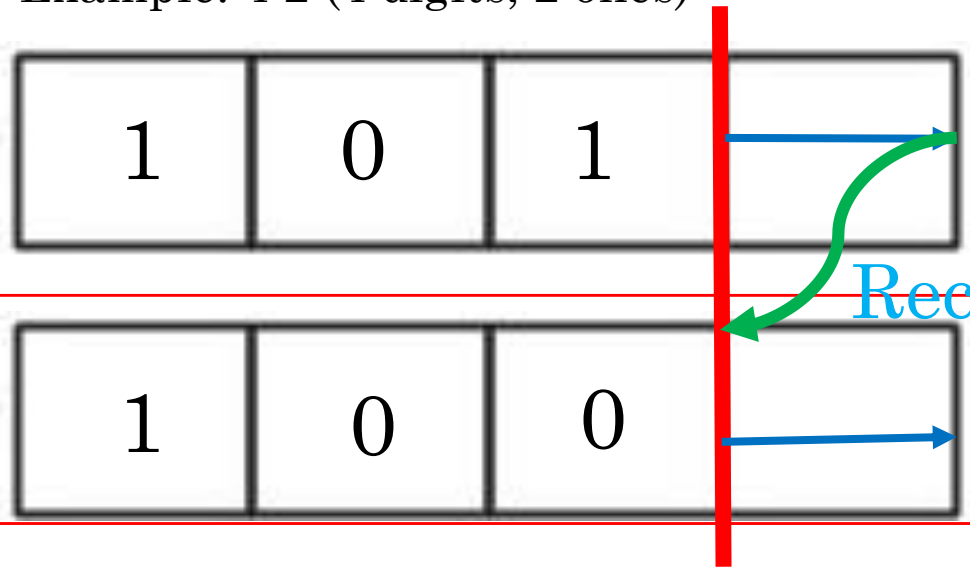
Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 1 = 1
➔print(1,1); //print 1 digit with 1 one

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔print(2,1)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 2 = 0
➔print(1,0); //print 1 digit with 0 one

Recursion prints the rest for me

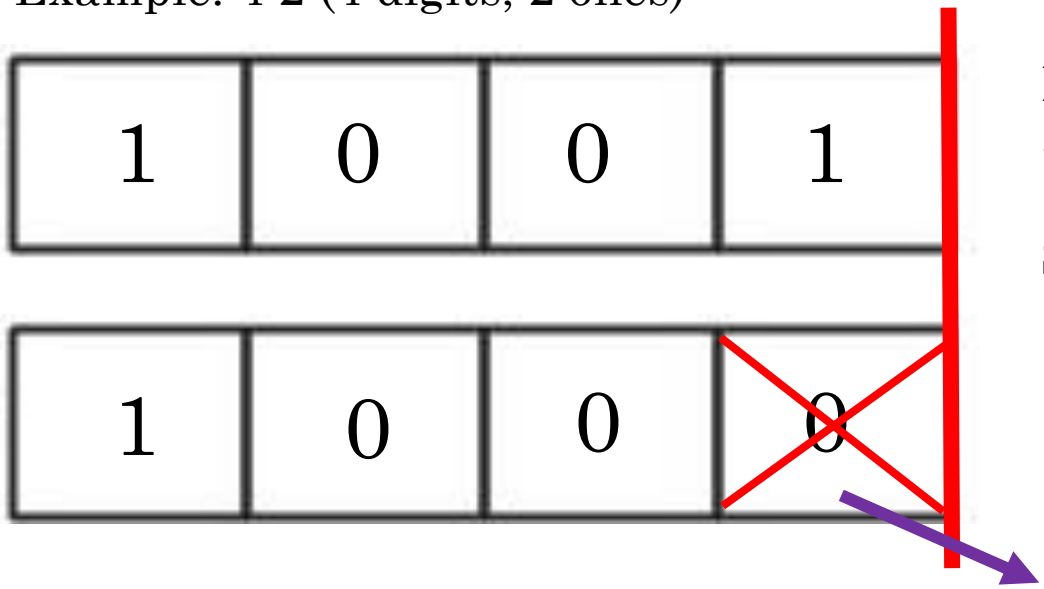
Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 1 = 1
➔print(1,1); //print 1 digit with 1 one

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔print(1,1)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)

4 – 4 = 0 2 – 2 = 0

➔print(0,0); //print 0 digits with 0 one ➔

STOP ➔ print this array

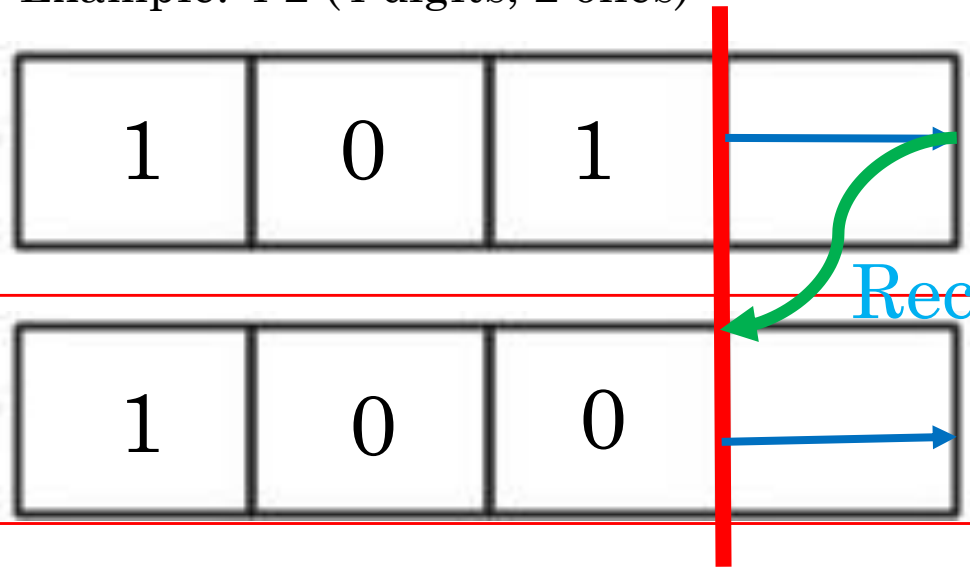
Condition: if n > k (i.e. after we print zero here, if we still have enough space for ones, then print zero)

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔print(2,1)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 2 = 0
➔print(1,0); //print 1 digit with 0 one

Recursion prints the rest for me

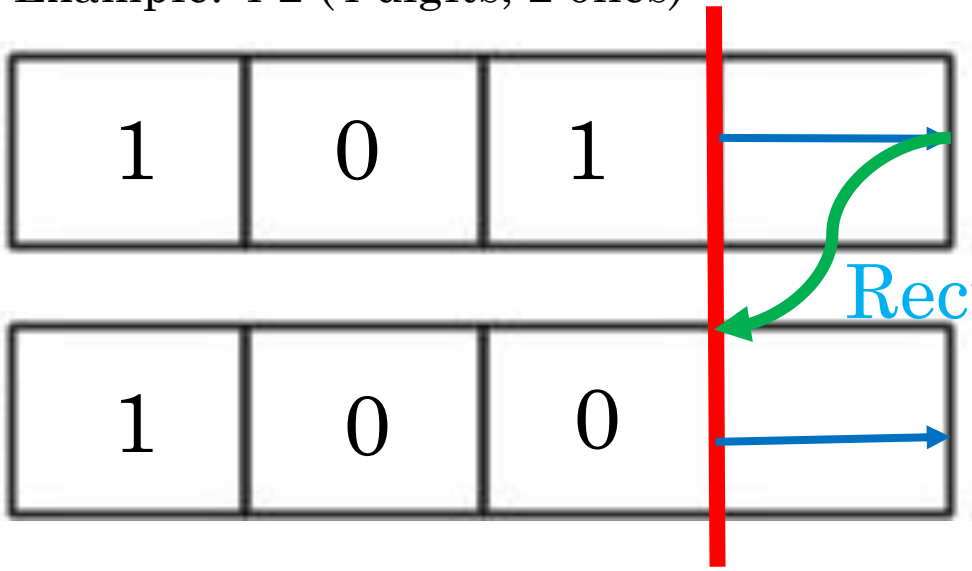
Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 1 = 1
➔print(1,1); //print 1 digit with 1 one

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔print(2,1)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 2 = 0
➔print(1,0); //print 1 digit with 0 one

Recursion prints the rest for me

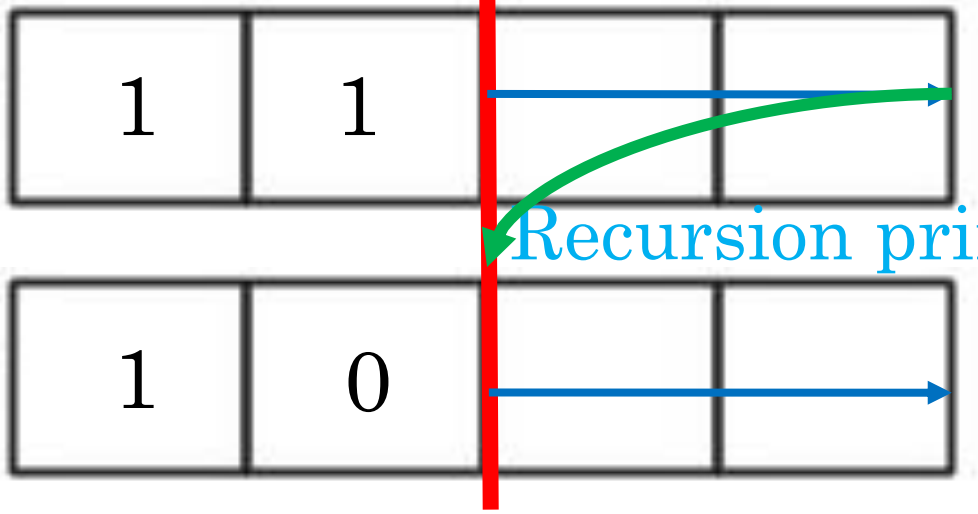
Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 1 = 1
➔print(1,1); //print 1 digit with 1 one

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔print(3,1)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n)	Ones left(k)
4 – 2 = 2	2 – 2 = 0

➔print(2,0); //print 2 digits with 0 one

Done

Recursion prints the rest for me

Digit Left(n)	Ones left(k)
4 – 2 = 2	2 – 1 = 1

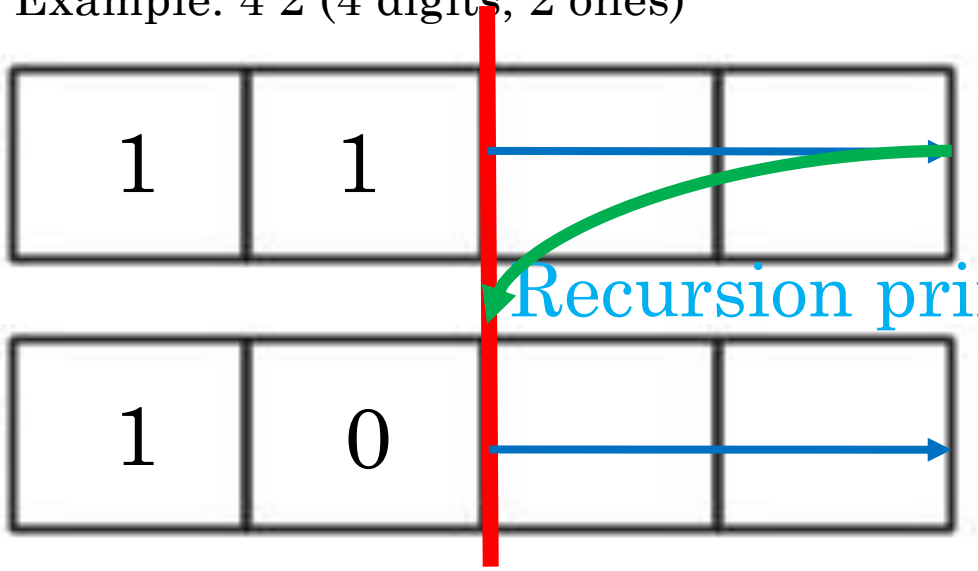
➔print(2,1); //print 2 digits with 1 one

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔print(3,1)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 2 = 2 2 – 2 = 0
➔print(2,0); //print 2 digits with 0 one

Done

Recursion prints the rest for me

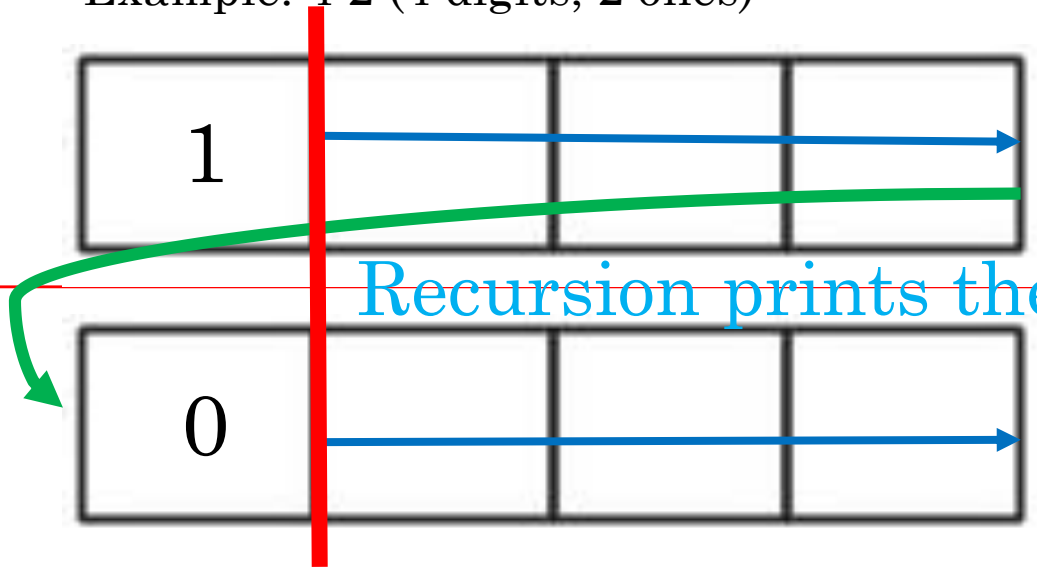
Digit Left(n) Ones left(k)
4 – 2 = 2 2 – 1 = 1
➔print(2,1); //print 2 digits with 1 one

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔ print(4,2)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 1 = 3 2 – 1 = 1
➔print(3,1); //print 3 digits with 1 one

Recursion prints the rest for me

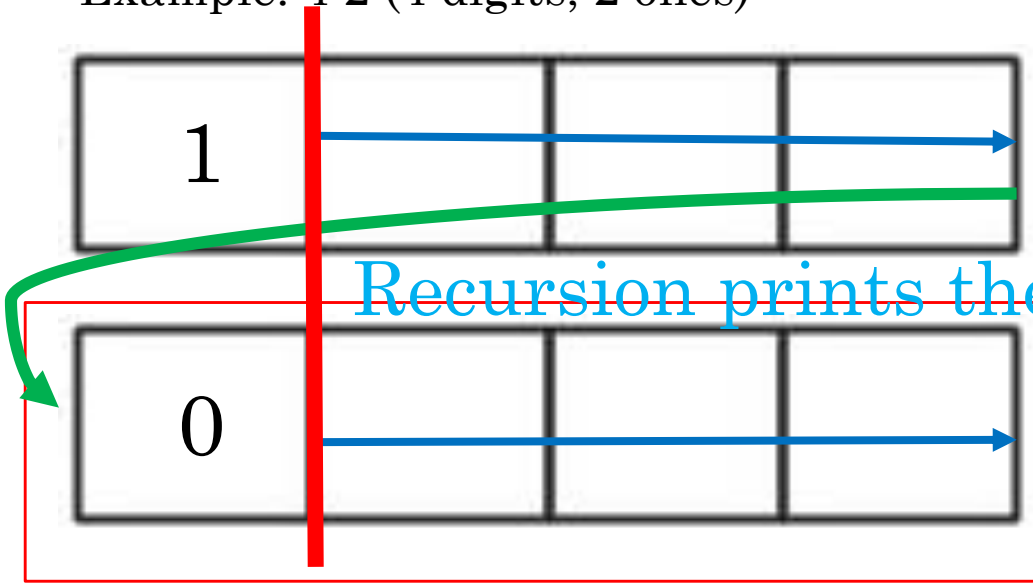
Digit Left(n) Ones left(k)
4 – 1 = 3 2 – 0 = 2
➔print(3,2); //print 3 digits with 2 ones

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔ print(4,2)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 1 = 3 2 – 1 = 1
➔print(3,1); //print 3 digits with 1 one

Recursion prints the rest for me

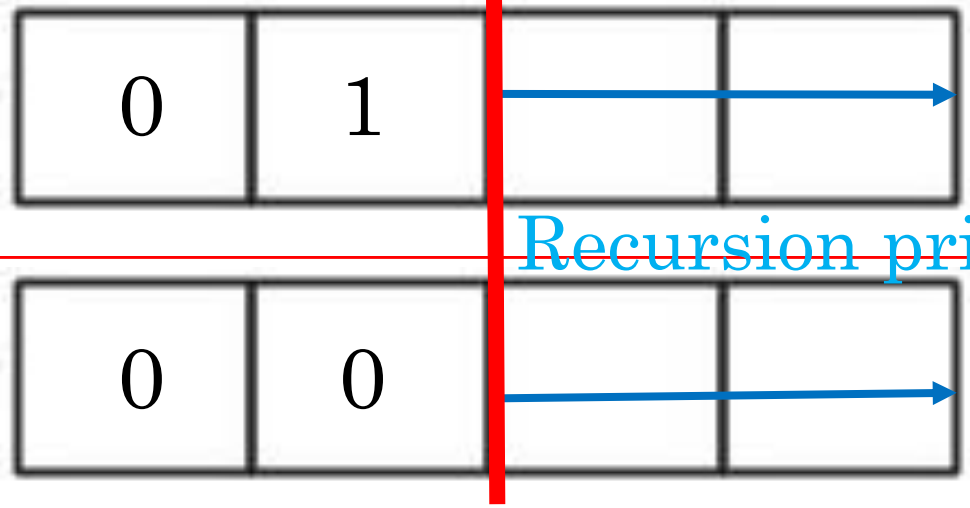
Digit Left(n) Ones left(k)
4 – 1 = 3 2 – 0 = 2
➔print(3,2); //print 3 digits with 2 ones

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔ print(3,2)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 2 = 2 2 – 1 = 1
➔print(2,1); //print 2 digits with 1 one

Recursion prints the rest for me

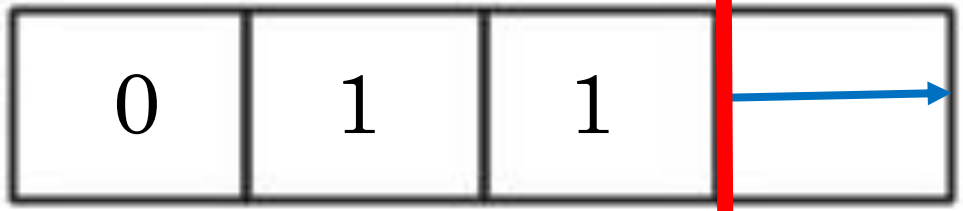
Digit Left(n) Ones left(k)
4 – 2 = 2 2 – 0 = 2
➔print(2,2); //print 2 digits with 2 ones

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

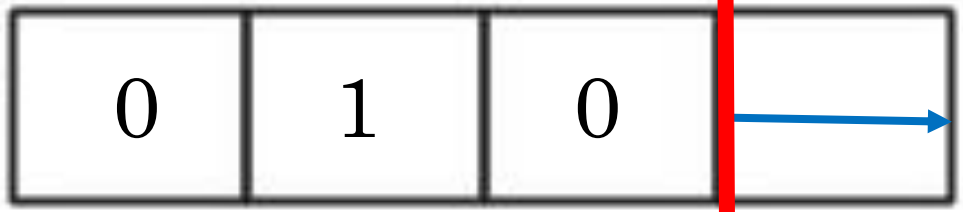
Understand Recursion ➔ print(2,1)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 2 = 0
➔print(1,0); //print 1 digits with 0 one

Recursion prints the rest for me



Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 1 = 1
➔print(1,1); //print 2 digits with 1 one

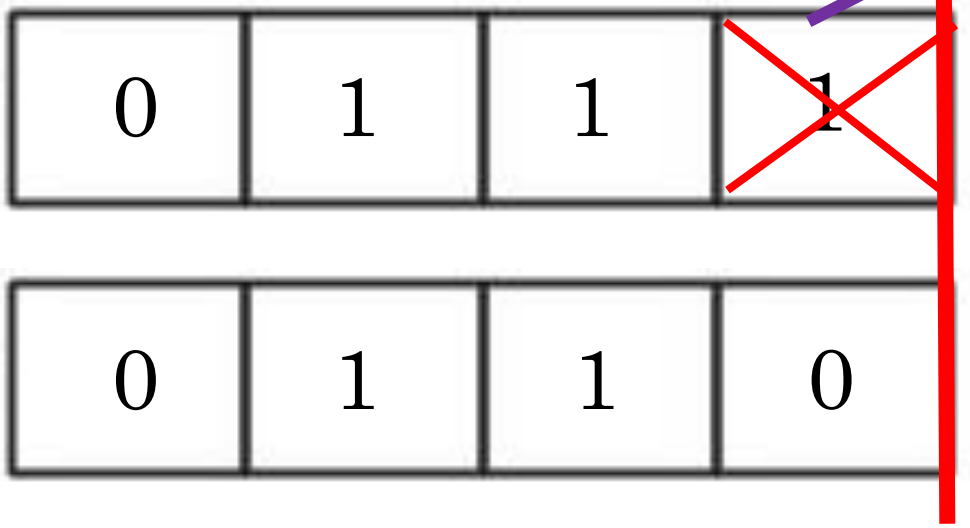
1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔ print(1,0)

Example: 4 2 (4 digits, 2 ones)

Condition: if k > 0 not met



Digit Left(n) Ones left(k)

4 – 4 = 0 2 – 2 = 0

➔print(0,0); //print 0 digits with 0 one

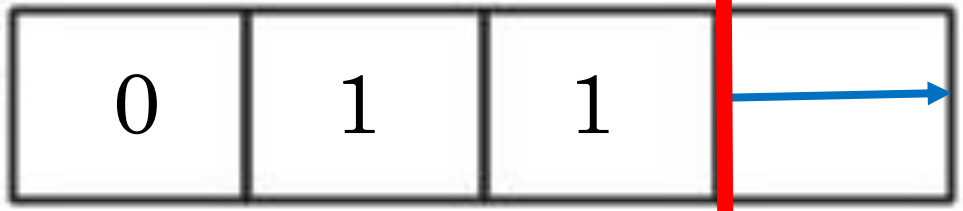
➔STOP ➔ print

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

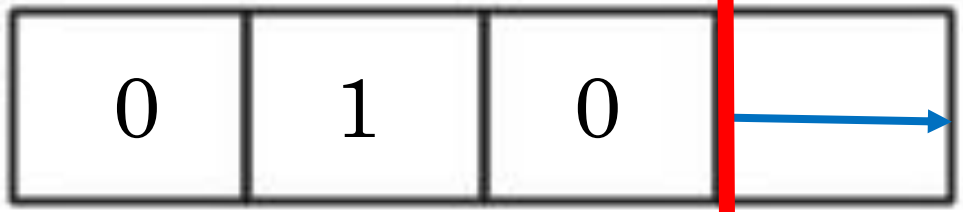
Understand Recursion ➔ print(2,1)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 2 = 0
➔print(1,0); //print 1 digits with 0 one

Recursion prints the rest for me



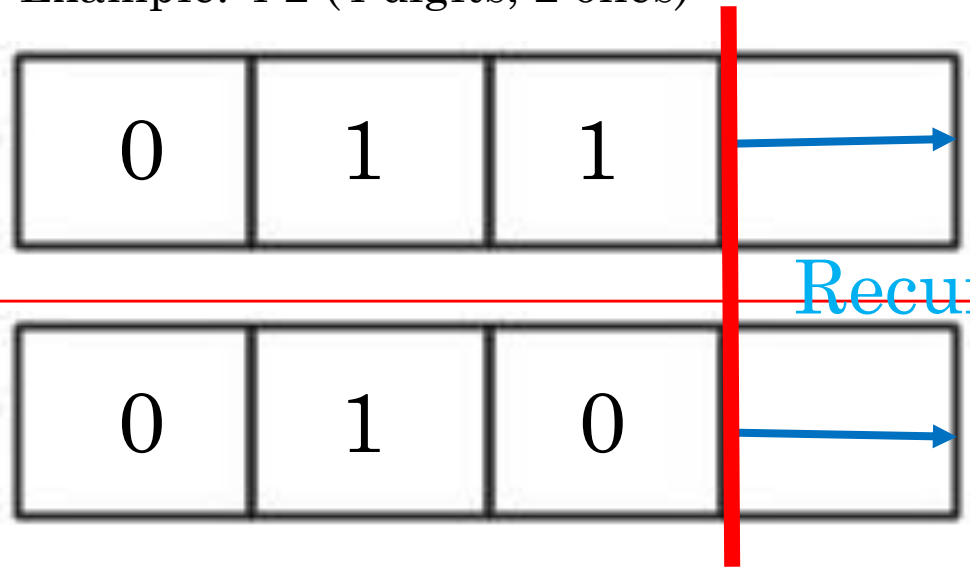
Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 1 = 1
➔print(1,1); //print 2 digits with 1 one

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔ print(2,1)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 2 = 0
➔print(1,0); //print 1 digits with 0 one

Recursion prints the rest for me

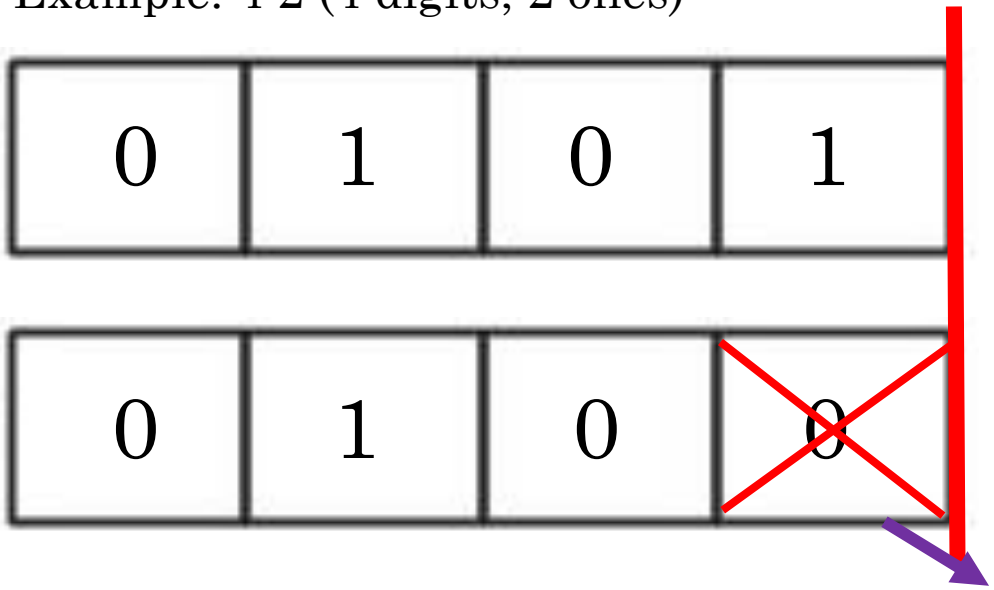
Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 1 = 1
➔print(1,1); //print 2 digits with 1 one

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔ print(1,1)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)

4 – 4 = 0 2 – 2 = 0

➔print(0,0); //print 0 digits with 0 one

➔Stop➔print

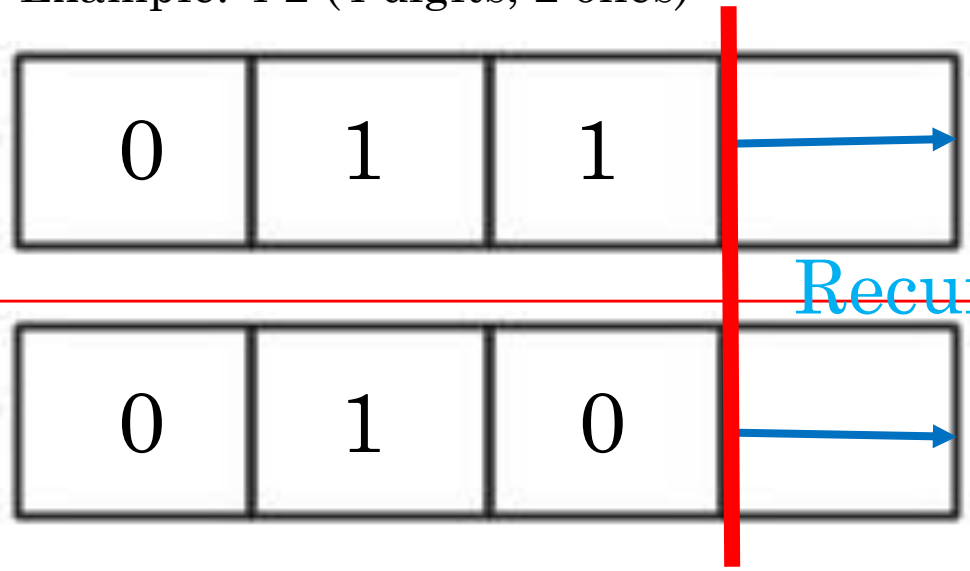
Condition: if n > k not met

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔ print(2,1)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 2 = 0
➔print(1,1); //print 1 digits with 0 one

Recursion prints the rest for me

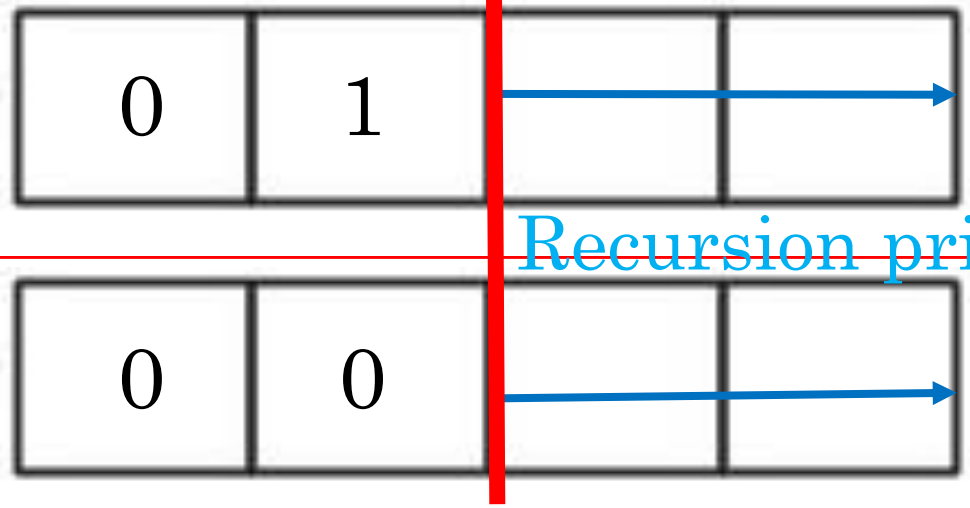
Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 1 = 1
➔print(1,1); //print 2 digits with 1 one

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔ print(3,2)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 2 = 2 2 – 1 = 1
➔print(2,1); //print 2 digits with 1 one

Recursion prints the rest for me

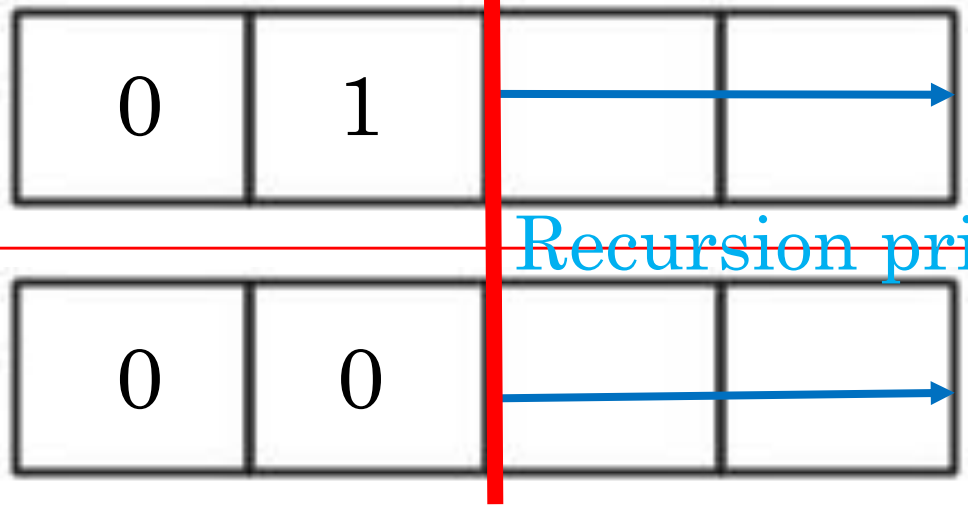
Digit Left(n) Ones left(k)
4 – 2 = 2 2 – 0 = 2
➔print(2,2); //print 2 digits with 2 ones

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔ print(3,2)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 2 = 2 2 – 1 = 1
➔print(2,1); //print 2 digits with 1 one

Recursion prints the rest for me

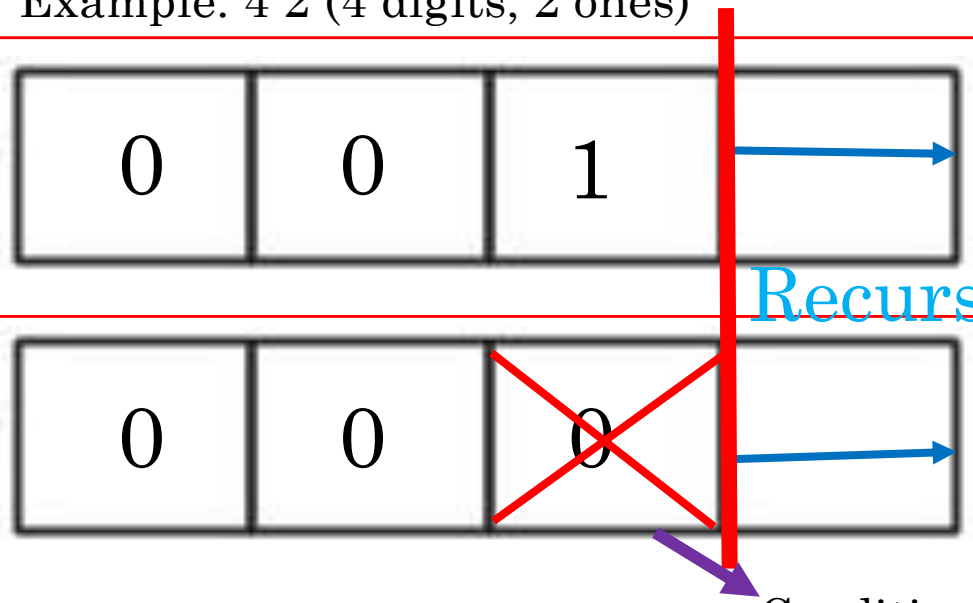
Digit Left(n) Ones left(k)
4 – 2 = 2 2 – 0 = 2
➔print(2,2); //print 2 digits with 2 ones

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔ print(2,2)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 1 = 1
➔print(1,1); //print 1 digit with 1 one

Recursion prints the rest for me

Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 0 = 2
➔print(1,2); //print 1 digit with 2 ones

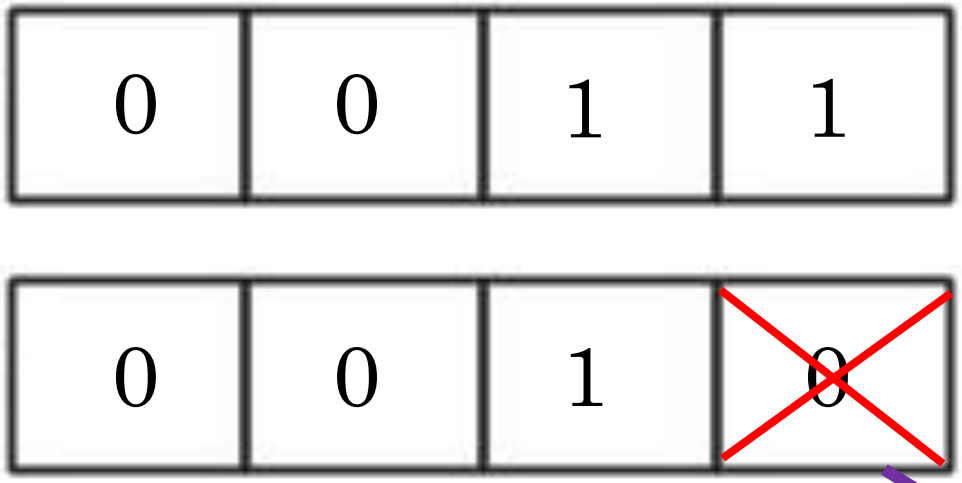
Condition: if n > k not met

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔ print(1,1)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 3 = 0 2 – 1 = 0
➔print(0,0); //print 0 digit with 0 one
➔stop ➔ print

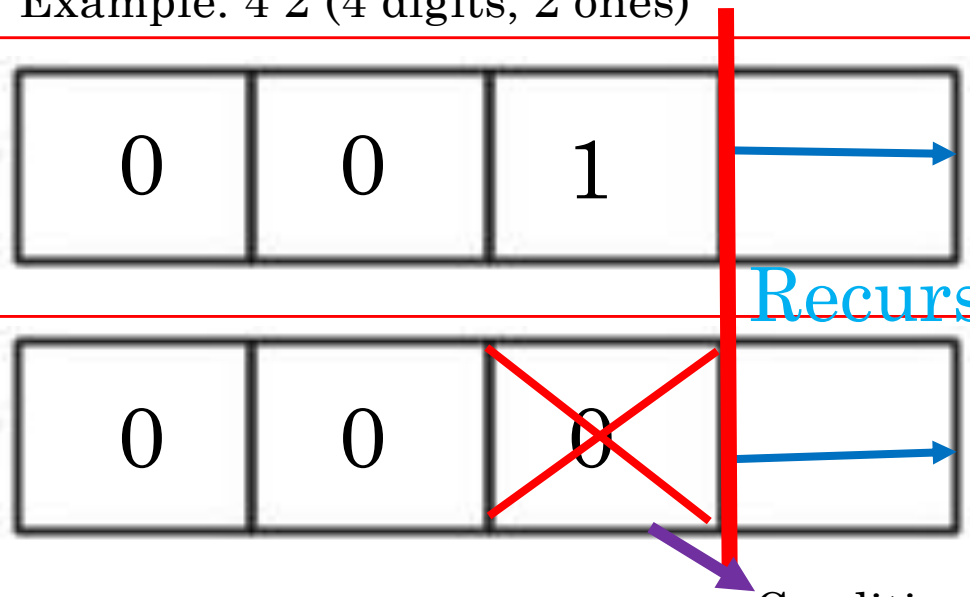
Condition: if n > k not met

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔ print(2,2)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 1 = 1
➔print(1,1); //print 1 digit with 1 one

Recursion prints the rest for me

Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 0 = 2
➔print(1,2); //print 1 digit with 2 ones

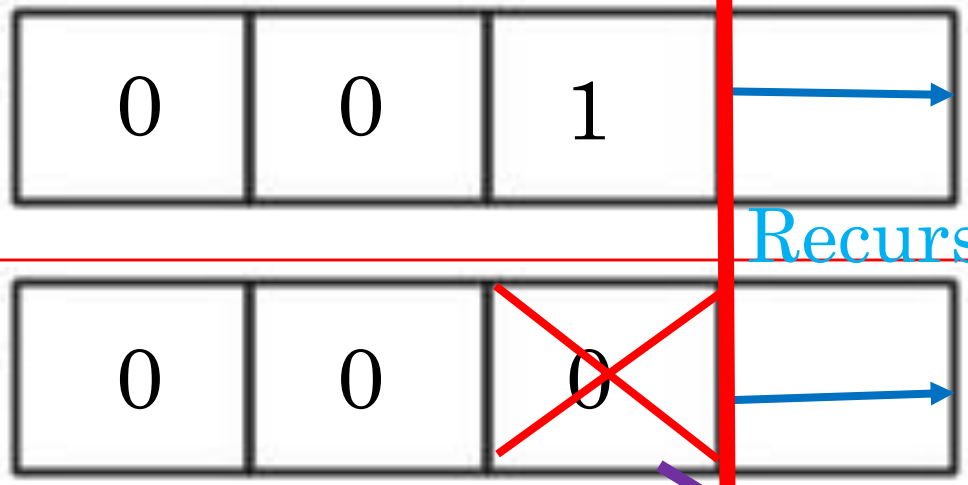
Condition: if n > k not met

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔ print(2,2)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 1 = 1
➔print(1,1); //print 1 digit with 1 one

Recursion prints the rest for me

Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 0 = 2
➔print(1,2); //print 1 digit with 2 ones

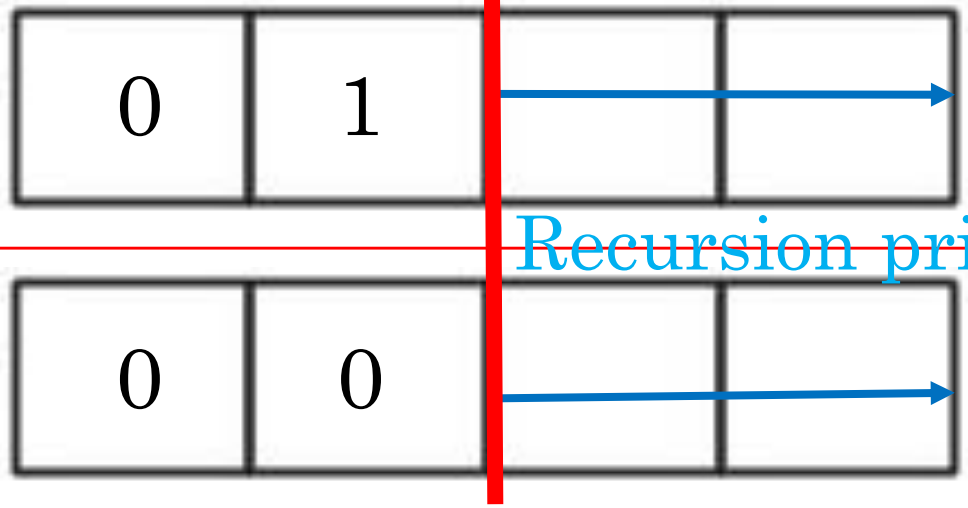
Condition: if n > k not met

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔ print(3,2)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 2 = 2 2 – 1 = 1
➔print(2,1); //print 2 digits with 1 one

Recursion prints the rest for me

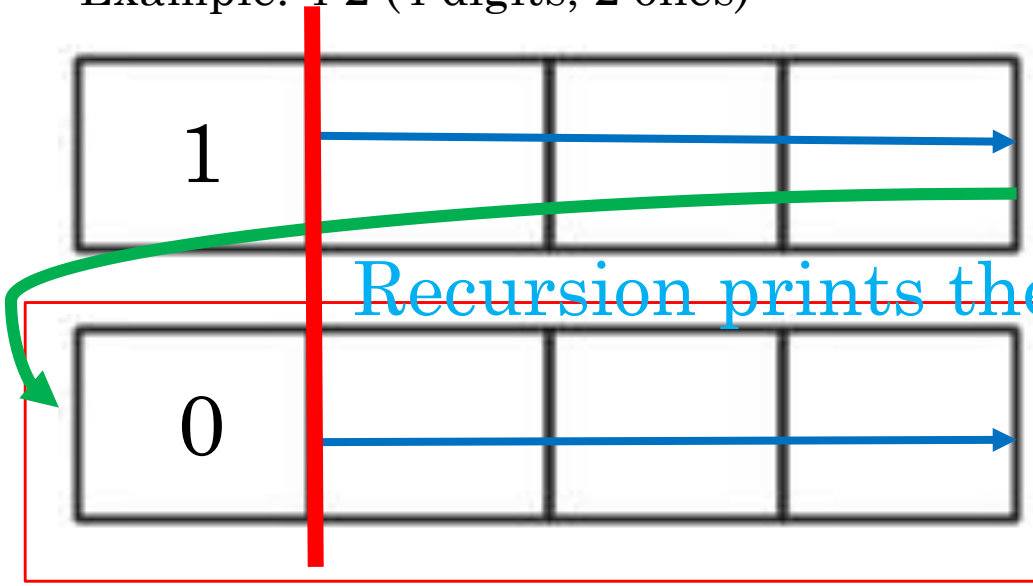
Digit Left(n) Ones left(k)
4 – 2 = 2 2 – 0 = 2
➔print(2,2); //print 2 digits with 2 ones

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

Understand Recursion ➔ print(4,2)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 1 = 3 2 – 1 = 1
➔print(3,1); //print 3 digits with 1 one

Recursion prints the rest for me

Digit Left(n) Ones left(k)
4 – 1 = 3 2 – 0 = 2
➔print(3,2); //print 3 digits with 2 ones

1100
1010
1001
0110
0101
0011

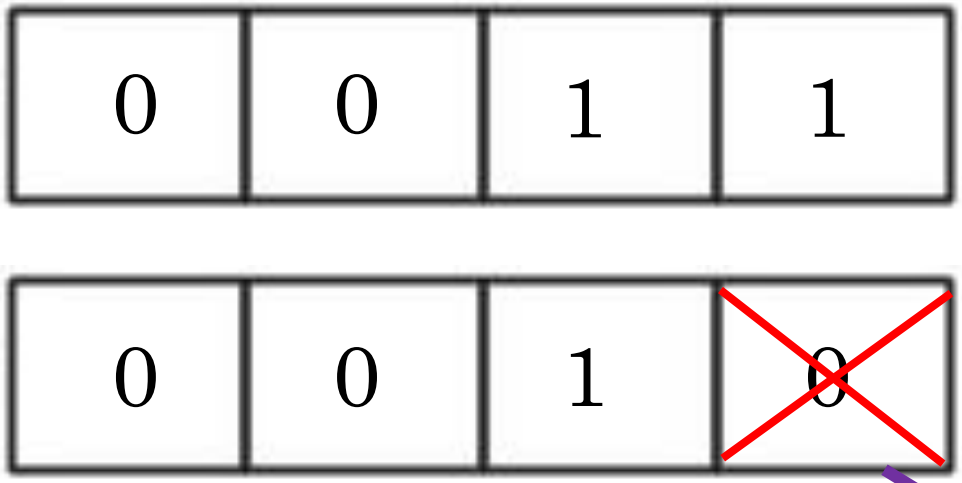
Nothing more to choose➔ End of recursion

After printing all the ones, print all the zeros

```
void printarray(int i, int j){
    if(i == 0 && j == 0){
        for(int a = 0; a < n; a++){
            cout<<array[a];
        }
        cout<<endl;
    }else{
        if (j > 0){
            array[n - i] = 1;
            printarray(i - 1, j - 1);
        }
        if (i > j){
            array[n - i] = 0;
            printarray(i-1,j);
        }
    }
}
```

Understand Recursion ➔ print(1,1)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 3 = 0 2 – 1 = 0
➔print(0,0); //print 0 digit with 0 one
➔stop ➔ print

Condition: if n > k not met

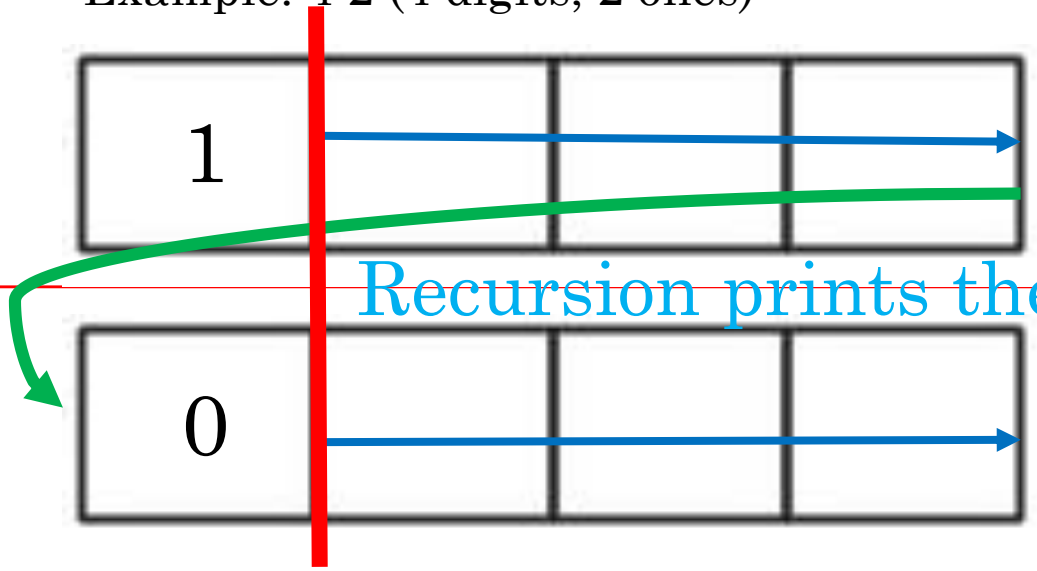
1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

```
void printarray(int i, int j){
    if(i == 0 && j == 0){
        for(int a = 0; a < n; a++){
            cout<<array[a];
        }
        cout<<endl;
    }else{
        //if has one left = print one series first
        if (j > 0){
            //n – i = current position
            array[n - i] = 1;
            printarray(i - 1, j - 1);
        }
        //then print zero series, print zero if and only if
        //the number of spots left > number of one left
        if (i > j){
            array[n - i] = 0;
            printarray(i-1,j);
        }
    }
}
```

Understand Recursion ➔ print(4,2)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 1 = 3 2 – 1 = 1
➔print(3,1); //print 3 digits with 1 one

Recursion prints the rest for me

Digit Left(n) Ones left(k)
4 – 1 = 3 2 – 0 = 2
➔print(3,2); //print 3 digits with 2 ones

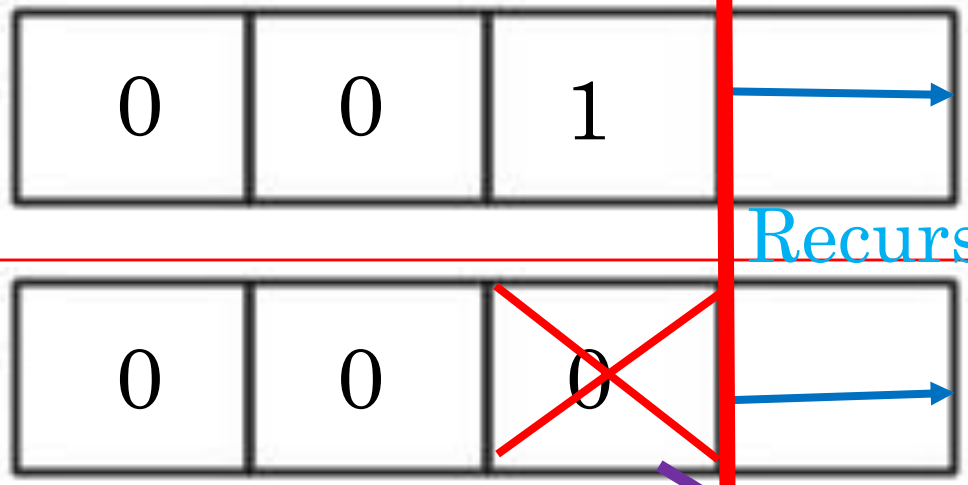
1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

```
void printarray(int i, int j){
    if(i == 0 && j == 0){
        for(int a = 0; a < n; a++){
            cout<<array[a];
        }
        cout<<endl;
    }else{
        //if has one left = print one series first
        if (j > 0){
            //n – i = current position
            array[n - i] = 1;
            printarray(i - 1, j - 1);
        }
        //then print zero series, print zero if and only if
        //the number of spots left > number of one left
        if (i > j){
            array[n - i] = 0;
            printarray(i-1,j);
        }
    }
}
```

Understand Recursion ➔ print(2,2)

Example: 4 2 (4 digits, 2 ones)



Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 1 = 1
➔print(1,1); //print 1 digit with 1 one

Recursion prints the rest for me

Digit Left(n) Ones left(k)
4 – 3 = 1 2 – 0 = 2
➔print(1,2); //print 1 digit with 2 ones

Condition: if n > k not met

1100
1010
1001
0110
0101
0011

After printing all the ones, print all the zeros

```
void printarray(int i, int j){
    if(i == 0 && j == 0){
        for(int a = 0; a < n; a++){
            cout<<array[a];
        }
        cout<<endl;
    }else{
        //if has one left = print one series first
        if (j > 0){
            //n – i = current position
            array[n - i] = 1;
            printarray(i - 1, j - 1);
        }
        //then print zero series, print zero if and only if
        //the number of spots left > number of one left
        if (i > j){
            array[n - i] = 0;
            printarray(i-1,j);
        }
    }
}
```

Problem S3: Absolutely Acidic

You are gathering readings of acidity level in a very long river in order to determine the health of the river. You have placed N sensors in the river, and each sensor gives an integer reading R. For the purposes of your research, you would like to know the frequency of each reading, and find **the absolute difference between the two most frequent readings.**

If there are more than two readings that have the highest frequency, the difference computed should be the *largest* such absolute difference between two readings with this frequency.

If there is only one reading with the largest frequency, but more than one reading with the second largest frequency, the difference computed should be the *largest* absolute difference between the most frequently occurring reading and any of the readings which occur with second-highest frequency.

Input Format

The first line of input will be the integer N ($2 \le N \le 2 \times 10^6$), the number of sensors. The next N lines each contain the reading for that sensor, which is an integer R ($1 \le R \le 1000$). You should assume that there are at least two different readings in the input.

Output Format

Output the positive integer value representing the absolute difference between the two most frequently occurring readings, subject to the tie-breaking rules outlined above.

Sample

Sample Input 1

5 1 1 1 4 3

Sample Output 1

3

Sample Input 2

4 10 6 1 8

Sample Output 2

9

Input: 2 1 1 1 3 4 3 10 12 3

Using array

	0	1	2	3	4	5	6	7	8	9
Sensor value	1	1	1	2	3	3	3	4	10	12
Frequency	3	3	3	1	3	3	3	1	1	1
HighestIndex	0	4								
SecondIndex										

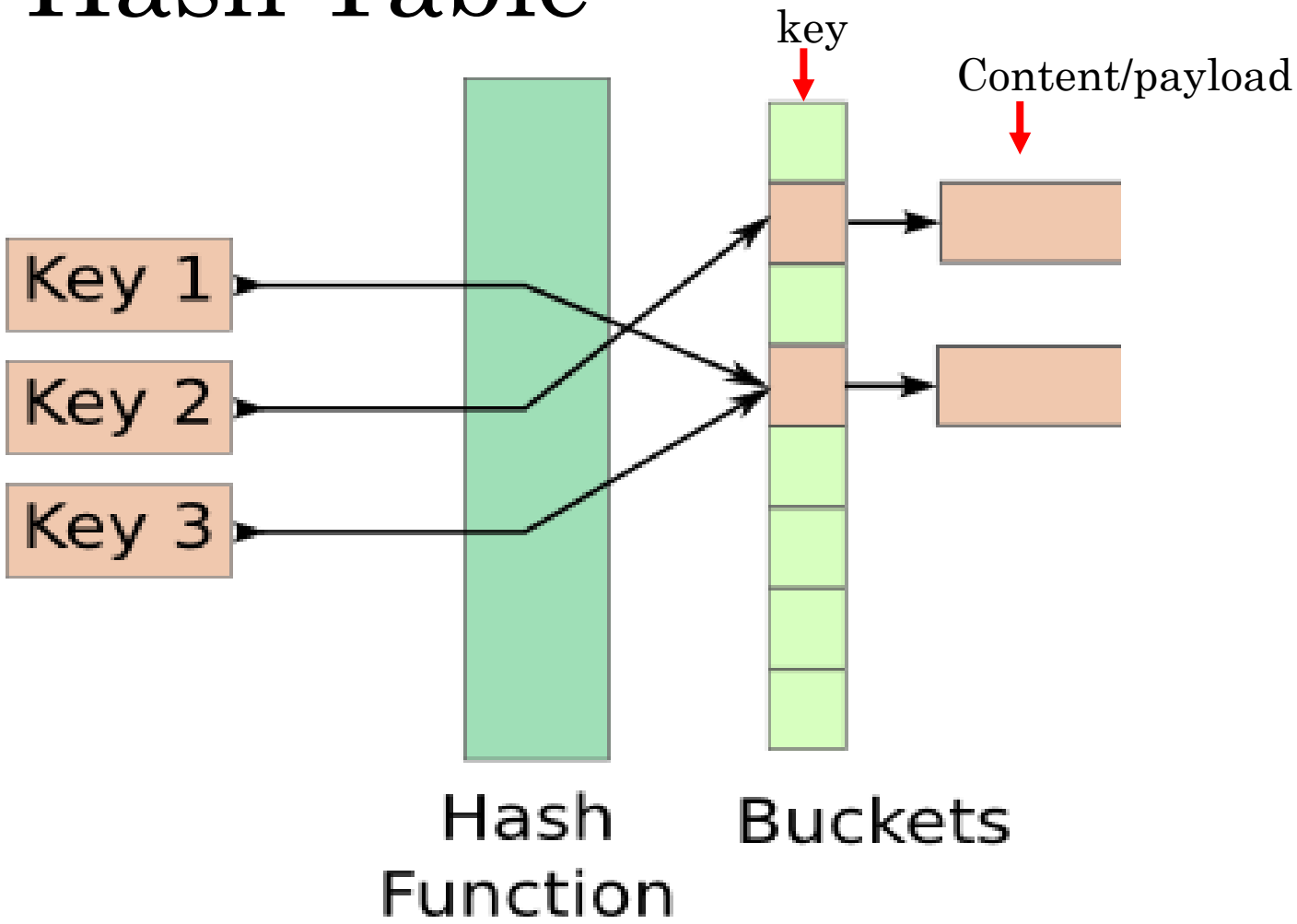
Input: 2 1 1 1 3 4 3 10 12 3

Using array

	0	1	2	3	4	5	6	7	8	9
Sensor value	1	1	1	2	3	3	3	4	10	12
Frequency	3	3	3	1	3	3	3	1	1	1
HighestIndex	0	4								
SecondIndex										

```
if (highestIndex.length() > 1){ //loop through highestIndex, find largest abs}
else{
        find largest abs of difference between
        sensorvalue[highestIndex[0]] and sensorvalue[secondIndex[i]]
}
```

Hash Table



Practice

- http://wcipeg.com/problem/ccc10j2
- http://wcipeg.com/problem/ccc14s1
- http://wcipeg.com/problem/ccc10j3