Contents

[Basic 1](#_Toc1495425)

[2s complement: 1](#_Toc1495426)

[Fibonacci: 1](#_Toc1495427)

[FibonacciUsingStreamIterator: 2](#_Toc1495428)

[FibonacciUsingForkJoinPool: 2](#_Toc1495429)

[Prime: 3](#_Toc1495430)

[Recursion 3](#_Toc1495431)

[Reverse A String: 3](#_Toc1495432)

[Count spaces in a string: 3](#_Toc1495433)

[Fibonacci: 4](#_Toc1495434)

[Power set backtracking: 4](#_Toc1495435)

[Advanced: 5](#_Toc1495436)

[All permutations using Heap’s algorithim: 5](#_Toc1495437)

[PowerSet By Masking: 5](#_Toc1495438)

# Basic

### 2s complement:

**Two's Complement binary for Positive Integers:** 0 is written as "0",1 is written as "1",2 is written as "10",3 is "11",4 is "100",5 is "101",1029 is "10000000101" == 2\*\*10 + 2\*\*2 + 2\*\*0 == 1024 + 4 + 1

**Two's Complement binary for Negative Integers:**Negative numbers are written with a leading one instead of a leading zero. So if you are using only 8 bits for your twos-complement numbers, then you treat patterns from "00000000" to "01111111" as the whole numbers from 0 to 127, and reserve "1xxxxxxx" for writing negative numbers. A negative number, -x, is written using the bit pattern for (x-1) with all of the bits complemented (switched from 1 to 0 or 0 to 1). So -1 is complement(1 - 1) = complement(0) = "11111111", and -10 is complement(10 - 1) = complement(9) = complement("00001001") = "11110110". This means that negative numbers go all the way down to -128 ("10000000").

### Fibonacci:

**def** fibonacci(N,a,b):  
 L=[]  
 **while** len(L) < N:  
 a,b=b,a+b *# the value of a+b that is assigned to b is the a, before the #value of b is assigned to a.*

L.append(a)  
 **return** L  
  
print (fibonacci(10,0,1))

*# [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]*

### FibonacciUsingStreamIterator:

Stream.*iterate*(**new int**[]{0, 1}, t -> **new int**[]{t[1], t[0] + t[1]})  
 .limit(10)  
 .forEach(x -> System.***out***.println(**"{"** + x[0] + **","** + x[1] + **"}"**));

### FibonacciUsingForkJoinPool:

**package** com.derivstreet;  
  
**import** java.util.concurrent.ForkJoinPool;  
**import** java.util.concurrent.ForkJoinTask;  
**import** java.util.concurrent.RecursiveAction;  
  
**public class** ForkJoinFib **extends** RecursiveAction {  
  
 **private static final long *threshold*** = 10;  
 **private volatile long number**;  
  
 **public** ForkJoinFib(**long** number) {  
 **this**.**number** = number;  
 }  
  
 **public long** getNumber() {  
 **return number**;  
 }  
  
 @Override  
 **protected void** compute() {  
 **long** n = **number**;  
 **if** (n <= ***threshold***) {  
 **number** = *fib*(n);  
 } **else** {  
 ForkJoinFib f1 = **new** ForkJoinFib(n - 1);  
 ForkJoinFib f2 = **new** ForkJoinFib(n - 2);  
 ForkJoinTask.*invokeAll*(f1, f2);  
 **number** = f1.**number** + f2.**number**;  
 }  
 }  
  
 **private static long** fib(**long** n) {  
 **if** (n <= 1) **return** n;  
 **else return** *fib*(n - 1) + *fib*(n - 2);  
 }  
  
 **public static void** main(String[] args) {  
  
 ForkJoinFib task = **new** ForkJoinFib(50);  
 **new** ForkJoinPool().invoke(task);  
  
 System.***out***.println(task.getNumber());  
  
 }  
  
}

### Prime:

**def** prime(nmax):  
 L = []  
 **for** n **in** range(2, nmax):  
 **for** factor **in** L:  
 **if** n % factor == 0:  
 **break  
 else**:  
 L.append(n)  
 print(L)  
  
prime(70)  
*# [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67]***def** gen\_primes(N):  
 primes = set()  
 **for** n **in** range(2, N):  
 **if** all(n % p > 0 **for** p **in** primes): *# all will return true on an empty iterable so when primes is empty to begin with primes get populated. See any/all pseudo code below* primes.add(n)  
 **yield** n  
  
print(\*gen\_primes(70))  
*# 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67  
  
# def any(iterable):  
# for item in iterable:  
# if item:  
# return True  
# return False  
#  
# def all(iterable):  
# for item in iterable:  
# if not item:  
# return False  
# return True*

# Recursion

### Reverse A String:

**def** reverse(s):  
 l = len(s)  
 **if** l > 1:  
 reversed = reverse(s[1:l])  
 **return** reversed + s[0]  
 **else**:  
 **return** s  
 **return** s  
print (reverse(**"poker"**))  
*#rekop*s = **"poker"**print (s[::-1])  
*#*

*rekop*

### Count spaces in a string:

**def** countSpacesInString(s,count):  
 print (s, count)  
 *#base case* **if not** s:  
 **return** count  
 **if**(s[0]==**" "**):  
 count += 1  
 *#recursive decomposition* count = countSpacesInString(s[1:len(s)],count)  
  
 **return** count  
  
count = 0  
count = countSpacesInString(**"fjaj ajf "**,count)  
print (count)  
*# fjaj ajf 0  
# jaj ajf 0  
# aj ajf 0  
# j ajf 0  
# ajf 0  
# ajf 1  
# jf 1  
# f 1  
# 1  
# 2  
# 3  
# 3*

### Fibonacci:

**def** fibonacciRecursive(n):  
 **if** n == 0:  
 **return** 0  
 **if** n == 1:  
 **return** 1  
 **else**:  
 **return** fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2)  
print (fibonacciRecursive(4))  
*#3*

### Power set backtracking:

**def** backTracking(at, bits, set, powerSets):  
 *#base case* **if** (at==len(bits)):  
 *#iterate through bits and include every bit that is 1* ps = []  
 **for** idx, val **in** enumerate(bits):  
 **if** (val == 1):  
 print (set[idx], end=**" "**)  
 ps.append(set[idx])  
 print (**"\n"**)  
 powerSets.append(ps)  
  
 *#recursive decomposition* **else**:  
 bits[at]=0  
 backTracking(at+1,bits,set, powerSets)  
 bits[at]=1  
 backTracking(at+1, bits, set,powerSets)  
powerSets = []  
backTracking(0,[0,0,0],[**'A'**,**'K'**,**'Q'**],powerSets)

*#backTracking(0,np.zeros(32),range(0,32),powerSets) #['A','K','Q']*

print (powerSets)

*# Q   
# K   
# K Q   
# A   
# A Q   
# A K   
# A K Q   
# [[], ['Q'], ['K'], ['K', 'Q'], ['A'], ['A', 'Q'], ['A', 'K'], ['A', 'K', 'Q']]*

# Advanced:

### All permutations using Heap’s algorithim:

### PowerSet By Masking:

**def** powerSetByMasking(set):  
  
 N = len(set);  
 N\_PS = 1 << N; *#2^N i.e. number of sets in the power set* **for** r **in** range (0,N\_PS): *#000, 010, 100, 011 etc* **for** i **in** range (0,N):  
 mask = 1 << i; *# 2^i 000, 010, 100, 011 etc* **if** ((r & mask) == mask ): *# for the rth set in the power set take those iths that are the 1s in the rth's binary representation* print(set[i], end=**" "**)  
 print(**"\r"**)  
  
powerSetByMasking([**'A'**, **'K'**, **'Q'**])  
*# A   
# K   
# A K   
# Q   
# A Q   
# K Q   
# A K Q*

### 