**Mathematical Algorithms**

1. Write an Efficient Method to Check if a Number is Multiple of 3

The very first solution that comes to our mind is the one that we learned in school. If

sum of digits in a number is multiple of 3 then number is multiple of 3 e.g., for 612 sum

of digits is 9 so it’s a multiple of 3. But this solution is not efficient. You have to get all

decimal digits one by one, add them and then check if sum is multiple of 3.

There is a pattern in binary representation of the number that can be used to find if

number is a multiple of 3. If difference between count of odd set bits (Bits set at odd

positions) and even set bits is multiple of 3 then is the number.

Example: 23 (00..10111)

1) Get count of all set bits at odd positions (For 23 it’s 3).

2) Get count of all set bits at even positions (For 23 it’s 1).

3) If difference of above two counts is a multiple of 3 then number is also a multiple of 3.

(For 23 it’s 2 so 23 is not a multiple of 3)

Take some more examples like 21, 15, etc…

Algorithm: isMutlipleOf3(n)

1) Make n positive if n is negative.

2) If number is 0 then return 1

3) If number is 1 then return 0

4) Initialize: odd\_count = 0, even\_count = 0

5) Loop while n != 0

a) If rightmost bit is set then increment odd count.

b) Right-shift n by 1 bit

c) If rightmost bit is set then increment even count.

d) Right-shift n by 1 bit

6) return isMutlipleOf3(odd\_count - even\_count)

**Proof:**

Above can be proved by taking the example of 11 in decimal numbers. (In this context

11 in decimal numbers is same as 3 in binary numbers)

If difference between sum of odd digits and even digits is multiple of 11 then decimal

number is multiple of 11. Let’s see how.

Let’s take the example of 2 digit numbers in decimal

AB = 11A -A + B = 11A + (B – A)

So if (B – A) is a multiple of 11 then is AB.

Let us take 3 digit numbers.

ABC = 99A + A + 11B – B + C = (99A + 11B) + (A + C – B)

So if (A + C – B) is a multiple of 11 then is (A+C-B)

Let us take 4 digit numbers now.

ABCD = 1001A + D + 11C – C + 999B + B – A

= (1001A – 999B + 11C) + (D + B – A -C )

So, if (B + D – A – C) is a multiple of 11 then is ABCD.

This can be continued for all decimal numbers.

Above concept can be proved for 3 in binary numbers in the same way.

**Time Complexity:** O(logn)

**Program:**

2. Efficient way to multiply with 7

We can multiply a number by 7 using bitwise operator. First left shift the number by 3 bits

(you will get 8n) then subtract the original numberfrom the shifted number and return the

difference (8n – n).

**Program:**

#include<stdio.h>

/\* Fnction to check if n is a multiple of 3\*/

**int** isMultipleOf3(**int** n)

{

**int** odd\_count = 0;

**int** even\_count = 0;

/\* Make no positive if +n is multiple of 3

then is -n. We are doing this to avoid

stack overflow in recursion\*/

**if**(n < 0) n = -n;

**if**(n == 0) **return** 1;

**if**(n == 1) **return** 0;

**while**(n)

{

/\* If odd bit is set then

increment odd counter \*/

**if**(n & 1)

odd\_count++;

n = n>>1;

/\* If even bit is set then

increment even counter \*/

**if**(n & 1)

even\_count++;

n = n>>1;

}

**return** isMultipleOf3(**abs**(odd\_count - even\_count));

}

/\* Program to test function isMultipleOf3 \*/

**int** main()

{

**int** num = 23;

**if** (isMultipleOf3(num))

**printf**("num is multiple of 3");

**else**

**printf**("num is not a multiple of 3");

**getchar**();

**return** 0;

}

**Time Complexity:** O(1)

**Space Complexity:** O(1)

Note: Works only for positive integers.

Same concept can be used for fast multiplication by 9 or other numbers.

3. Write a C program to print all permutations of a given string

A permutation, also called an “arrangement number” or “order,” is a rearrangement of the

elements of an ordered list S into a one-to-one correspondence with S itself. A string of

length n has n! permutation.

Source: Mathword(http://mathworld.wolfram.com/Permutation.html)

Below are the permutations of string ABC.

ABC, ACB, BAC, BCA, CAB, CBA

Here is a solution using backtracking.

# include<stdio.h>

**int** multiplyBySeven(unsigned **int** n)

{

/\* Note the inner bracket here. This is needed

because precedence of '-' operator is higher

than '<<' \*/

**return** ((n<<3) - n);

}

/\* Driver program to test above function \*/

**int** main()

{

unsigned **int** n = 4;

**printf**("%u", multiplyBySeven(n));

**getchar**();

**return** 0;

}

Output:

ABC

ACB

BAC

BCA

CBA

CAB

**Algorithm Paradigm:** Backtracking

**Time Complexity:** O(n\*n!)

Please write comments if you find the above codes/algorithms incorrect, or find other

ways to solve the same problem.

# include <stdio.h>

/\* Function to swap values at two pointers \*/

**void** swap (**char** \*x, **char** \*y)

{

**char** temp;

temp = \*x;

\*x = \*y;

\*y = temp;

}

/\* Function to print permutations of string

This function takes three parameters:

1. String

2. Starting index of the string

3. Ending index of the string. \*/

**void** permute(**char** \*a, **int** i, **int** n)

{

**int** j;

**if** (i == n)

**printf**("%s\n", a);

**else**

{

**for** (j = i; j <= n; j++)

{

swap((a+i), (a+j));

permute(a, i+1, n);

swap((a+i), (a+j)); //backtrack

}

}

}

/\* Driver program to test above functions \*/

**int** main()

{

**char** a[] = "ABC";

permute(a, 0, 2);

**getchar**();

**return** 0;

}

4. Lucky Numbers

Lucky numbers are subset of integers. Rather than going into much theory, let us see the

process of arriving at lucky numbers,

Take the set of integers

1,2,3,4,5,6,7,8,9,10,11,12,14,15,16,17,18,19,……

First, delete every second number, we get following reduced set.

1,3,5,7,9,11,13,15,17,19,…………

Now, delete every third number, we get

1, 3, 7, 9, 13, 15, 19,….….

Continue this process indefinitely……

Any number that does NOT get deleted due to above process is called “lucky”.

Therefore, set of lucky numbers is 1, 3, 7, 13,………

Now, given an integer ‘n’, write a function to say whether this number is lucky or not.

bool isLucky(int n)

**Algorithm:**

Before every iteration, if we calculate position of the given no, then in a given iteration,

we can determine if the no will be deleted. Suppose calculated position for the given no.

is P before some iteration, and each Ith no. is going to be removed in this iteration, if P

< I then input no is lucky, if P is such that P%I == 0 (I is a divisor of P), then input no is

not lucky.

**Recursive Way:**

**Example:**

Let’s us take an example of 19

1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,15,17,18,19,20,21,……

1,3,5,7,9,11,13,15,17,19,…..

1,3,7,9,13,15,19,……….

1,3,7,13,15,19,………

1,3,7,13,19,………

In next step every 6th no .in sequence will be deleted. 19 will not be deleted after this

step because position of 19 is 5th after this step. Therefore, 19 is lucky. Let’s see how

above C code finds out:

**Current function call Position after this call Counter for next call Next Call**

isLucky(19 ) 10 3 isLucky(10)

isLucky(10) 7 4 isLucky(7)

isLucky(7) 6 5 isLucky(6)

isLucky(6) 5 6 isLucky(5)

When isLucky(6) is called, it returns 1 (because counter > n).

**Iterative Way:**

Please see this comment for another simple and elegant implementation of the above

algorithm.

#include <stdio.h>

#define bool int

/\* Returns 1 if n is a lucky no. ohterwise returns 0\*/

**bool** isLucky(**int** n)

{

**static int** counter = 2;

/\*variable next\_position is just for readability of

the program we can remove it and use n only \*/

**int** next\_position = n;

**if**(counter > n)

**return** 1;

**if**(n%counter == 0)

**return** 0;

/\*calculate next position of input no\*/

next\_position -= next\_position/counter;

counter++;

**return** isLucky(next\_position);

}

/\*Driver function to test above function\*/

**int** main()

{

**int** x = 5;

**if**( isLucky(x) )

**printf**("%d is a lucky no.", x);

**else**

**printf**("%d is not a lucky no.", x);

**getchar**();

}

Please write comments if you find any bug in the given programs or other ways to solve

the same problem.

5. Write a program to add two numbers in base 14

Asked by Anshya.

Below are the different ways to add base 14 numbers.

**Method 1**

Thanks to Raj for suggesting this method.

1. Convert both i/p base 14 numbers to base 10.

2. Add numbers.

3. Convert the result back to base 14.

**Method 2**

Just add the numbers in base 14 in same way we add in base 10. Add numerals of both

numbers one by one from right to left. If there is a carry while adding two numerals,

consider the carry for adding next numerals.

Let us consider the presentation of base 14 numbers same as hexadecimal numbers

A --> 10

B --> 11

C --> 12

D --> 13

Example:

num1 = 1 2 A

num2 = C D 3

1. Add A and 3, we get 13(D). Since 13 is smaller than

14, carry becomes 0 and resultant numeral becomes D

2. Add 2, D and carry(0). we get 15. Since 15 is greater

than 13, carry becomes 1 and resultant numeral is 15 - 14 = 1

3. Add 1, C and carry(1). we get 14. Since 14 is greater

than 13, carry becomes 1 and resultant numeral is 14 - 14 = 0

Finally, there is a carry, so 1 is added as leftmost numeral and the result becomes

101D

**Implementation of Method 2**

# include <stdio.h>

# include <stdlib.h>

# define bool int

**int** getNumeralValue(**char** );

**char** getNumeral(**int** );

/\* Function to add two numbers in base 14 \*/

**char** \*sumBase14(**char** \*num1, **char** \*num2)

{

**int** l1 = **strlen**(num1);

**int** l2 = **strlen**(num2);

**char** \*res;

**int** i;

**int** nml1, nml2, res\_nml;

**bool** carry = 0;

**if**(l1 != l2)

{

**printf**("Function doesn't support numbers of different"

" lengths. If you want to add such numbers then"

" prefix smaller number with required no. of zeroes");

**getchar**();

**assert**(0);

}

/\* Note the size of the allocated memory is one

more than i/p lenghts for the cases where we

have carry at the last like adding D1 and A1 \*/

res = (**char** \*)**malloc**(**sizeof**(**char**)\*(l1 + 1));

/\* Add all numerals from right to left \*/

**for**(i = l1-1; i >= 0; i--)

{

/\* Get decimal values of the numerals of

i/p numbers\*/

nml1 = getNumeralValue(num1[i]);

nml2 = getNumeralValue(num2[i]);

/\* Add decimal values of numerals and carry \*/

res\_nml = carry + nml1 + nml2;

/\* Check if we have carry for next addition

of numerals \*/

**if**(res\_nml >= 14)

{

carry = 1;

res\_nml -= 14;

}

**else**

{

carry = 0;

}

res[i+1] = getNumeral(res\_nml);

}

/\* if there is no carry after last iteration

**Notes:**

Above approach can be used to add numbers in any base. We don’t have to do string

operations if base is smaller than 10.

You can try extending the above program for numbers of different lengths.

Please comment if you find any bug in the program or a better approach to do the

same.

/\* if there is no carry after last iteration

then result should not include 0th character

of the resultant string \*/

**if**(carry == 0)

**return** (res + 1);

/\* if we have carry after last iteration then

result should include 0th character \*/

res[0] = '1';

**return** res;

}

/\* Function to get value of a numeral

For example it returns 10 for input 'A'

1 for '1', etc \*/

**int** getNumeralValue(**char** num)

{

**if**( num >= '0' && num <= '9')

**return** (num - '0');

**if**( num >= 'A' && num <= 'D')

**return** (num - 'A' + 10);

/\* If we reach this line caller is giving

invalid character so we assert and fail\*/

**assert**(0);

}

/\* Function to get numeral for a value.

For example it returns 'A' for input 10

'1' for 1, etc \*/

**char** getNumeral(**int** val)

{

**if**( val >= 0 && val <= 9)

**return** (val + '0');

**if**( val >= 10 && val <= 14)

**return** (val + 'A' - 10);

/\* If we reach this line caller is giving

invalid no. so we assert and fail\*/

**assert**(0);

}

/\*Driver program to test above functions\*/

**int** main()

{

**char** \*num1 = "DC2";

**char** \*num2 = "0A3";

**printf**("Result is %s", sumBase14(num1, num2));

**getchar**();

**return** 0;

}

6. Babylonian method for square root

**Algorithm:**

This method can be derived from (but predates) Newton–Raphson method.

1 Start with an arbitrary positive start value x (the closer to the

root, the better).

2 Initialize y = 1.

3. Do following until desired approximation is achieved.

a) Get the next approximation for root using average of x and y

b) Set y = n/x

**Implementation:**

**Example:**

n = 4 /\*n itself is used for initial approximation\*/

Initialize x = 4, y = 1

Next Approximation x = (x + y)/2 (= 2.500000),

y = n/x (=1.600000)

Next Approximation x = 2.050000,

y = 1.951220

Next Approximation x = 2.000610,

/\*Returns the square root of n. Note that the function \*/

**float** squareRoot(**float** n)

{

/\*We are using n itself as initial approximation

This can definitely be improved \*/

**float** x = n;

**float** y = 1;

**float** e = 0.000001; /\* e decides the accuracy level\*/

**while**(x - y > e)

{

x = (x + y)/2;

y = n/x;

}

**return** x;

}

/\* Driver program to test above function\*/

**int** main()

{

**int** n = 50;

**printf** ("Square root of %d is %f", n, squareRoot(n));

**getchar**();

}

y = 1.999390

Next Approximation x = 2.000000,

y = 2.000000

Terminate as (x - y) > e now.

If we are sure that n is a perfect square, then we can use following method. The method

can go in infinite loop for non-perfect-square numbers. For example, for 3 the below

while loop will never terminate.

**References;**

http://en.wikipedia.org/wiki/Square\_root

http://en.wikipedia.org/wiki/Babylonian\_method#Babylonian\_method

Asked by Snehal

Please write comments if you find any bug in the above program/algorithm, or if you

want to share more information about Babylonian method.

7. Multiply two integers without using multiplication, division and

bitwise operators, and no loops

Asked by Kapil

By making use of recursion, we can multiply two integers with the given constraints.

To multiply x and y, recursively add x y times.

/\*Returns the square root of n. Note that the function

will not work for numbers which are not perfect squares\*/

unsigned **int** squareRoot(**int** n)

{

**int** x = n;

**int** y = 1;

**while**(x > y)

{

x = (x + y)/2;

y = n/x;

}

**return** x;

}

/\* Driver program to test above function\*/

**int** main()

{

**int** n = 49;

**printf** (" root of %d is %d", n, squareRoot(n));

**getchar**();

}

Thanks to geek4u for suggesting this method.

Time Complexity: O(y) where y is the second argument to function multiply().

Please write comments if you find any of the above code/algorithm incorrect, or find

better ways to solve the same problem.

8. Print all combinations of points that can compose a given number

You can win three kinds of basketball points, 1 point, 2 points, and 3 points. Given a total

score n, print out all the combination to compose n.

Examples:

For n = 1, the program should print following:

1

For n = 2, the program should print following:

1 1

2

For n = 3, the program should print following:

1 1 1

1 2

2 1

#include<stdio.h>

/\* function to multiply two numbers x and y\*/

**int** multiply(**int** x, **int** y)

{

/\* 0 multiplied with anything gives 0 \*/

**if**(y == 0)

**return** 0;

/\* Add x one by one \*/

**if**(y > 0 )

**return** (x + multiply(x, y-1));

/\* the case where y is negative \*/

**if**(y < 0 )

**return** -multiply(x, -y);

}

**int** main()

{

**printf**("\n %d", multiply(5, -11));

**getchar**();

**return** 0;

}

3

For n = 4, the program should print following:

1 1 1 1

1 1 2

1 2 1

1 3

2 1 1

2 2

3 1

and so on …

Algorithm:

At first position we can have three numbers 1 or 2 or 3.

First put 1 at first position and recursively call for n-1.

Then put 2 at first position and recursively call for n-2.

Then put 3 at first position and recursively call for n-3.

If n becomes 0 then we have formed a combination that compose n, so print the current

combination.

Below is a generalized implementation. In the below implementation, we can change

MAX\_POINT if there are higher points (more than 3) in the basketball game.

Asked by Aloe

Please write comments if you find any bug in above code/algorithm, or find other ways

to solve the same problem.

#define MAX\_POINT 3

#define ARR\_SIZE 100

#include<stdio.h>

/\* Utility function to print array arr[] \*/

**void** printArray(**int** arr[], **int** arr\_size);

/\* The function prints all combinations of numbers 1, 2, ...MAX\_POINT

that sum up to n.

i is used in recursion keep track of index in arr[] where next

element is to be added. Initital value of i must be passed as 0 \*/

**void** printCompositions(**int** n, **int** i)

{

/\* array must be static as we want to keep track

of values stored in arr[] using current calls of

printCompositions() in function call stack\*/

**static int** arr[ARR\_SIZE];

**if** (n == 0)

{

printArray(arr, i);

}

**else if**(n > 0)

{

**int** k;

**for** (k = 1; k <= MAX\_POINT; k++)

{

arr[i]= k;

printCompositions(n-k, i+1);

}

}

}

/\* UTILITY FUNCTIONS \*/

/\* Utility function to print array arr[] \*/

**void** printArray(**int** arr[], **int** arr\_size)

{

**int** i;

**for** (i = 0; i < arr\_size; i++)

**printf**("%d ", arr[i]);

**printf**("\n");

}

/\* Driver function to test above functions \*/

**int** main()

{

**int** n = 5;

**printf**("Differnt compositions formed by 1, 2 and 3 of %d are\n", n);

printCompositions(n, 0);

**getchar**();

**return** 0;

}

9. Write you own Power without using multiplication(\*) and division(/)

operators

**Method 1 (Using Nested Loops)**

We can calculate power by using repeated addition.

For example to calculate 5^6.

1) First 5 times add 5, we get 25. (5^2)

2) Then 5 times add 25, we get 125. (5^3)

3) Then 5 time add 125, we get 625 (5^4)

4) Then 5 times add 625, we get 3125 (5^5)

5) Then 5 times add 3125, we get 15625 (5^6)

**Method 2 (Using Recursion)**

Recursively add a to get the multiplication of two numbers. And recursively multiply to

get *a* raise to the power *b*.

/\* Works only if a >= 0 and b >= 0 \*/

**int pow**(**int** a, **int** b)

{

**if** (b == 0)

**return** 1;

**int** answer = a;

**int** increment = a;

**int** i, j;

**for**(i = 1; i < b; i++)

{

**for**(j = 1; j < a; j++)

{

answer += increment;

}

increment = answer;

}

**return** answer;

}

/\* driver program to test above function \*/

**int** main()

{

**printf**("\n %d", **pow**(5, 3));

**getchar**();

**return** 0;

}

Please write comments if you find any bug in above code/algorithm, or find other ways

to solve the same problem.

10. Program for Fibonacci numbers

The Fibonacci numbers are the numbers in the following integer sequence.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 141, ……..

In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the

recurrence relation

with seed values

Write a function *int fib(int n)* that returns . For example, if *n* = 0, then *fib()* should return

0. If n = 1, then it should return 1. For n > 1, it should return

Following are different methods to get the nth Fibonacci number.

**Method 1 ( Use recursion )**

#include<stdio.h>

/\* A recursive function to get a^b

Works only if a >= 0 and b >= 0 \*/

**int pow**(**int** a, **int** b)

{

**if**(b)

**return** multiply(a, **pow**(a, b-1));

**else**

**return** 1;

}

/\* A recursive function to get x\*y \*/

**int** multiply(**int** x, **int** y)

{

**if**(y)

**return** (x + multiply(x, y-1));

**else**

**return** 0;

}

/\* driver program to test above functions \*/

**int** main()

{

**printf**("\n %d", **pow**(5, 3));

**getchar**();

**return** 0;

}

A simple method that is a direct recusrive implementation mathematical recurance

relation given above.

*Time Complexity:* T(n) = T(n-1) + T(n-2) which is exponential.

We can observe that this implementation does a lot of repeated work (see the following

recursion tree). So this is a bad implementation for nth Fibonacci number.

fib(5)

/ \

fib(4) fib(3)

/ \ / \

fib(3) fib(2) fib(2) fib(1)

/ \ / \ / \

fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)

/ \

fib(1) fib(0)

*Extra Space:* O(n) if we consider the fuinction call stack size, otherwise O(1).

**Method 2 ( Use Dynamic Programming )**

We can avoid the repeated work done is the method 1 by storing the Fibonacci numbers

calculated so far.

#include<stdio.h>

**int** fib(**int** n)

{

**if** (n <= 1)

**return** n;

**return** fib(n-1) + fib(n-2);

}

**int** main ()

{

**int** n = 9;

**printf**("%d", fib(n));

**getchar**();

**return** 0;

}

*Time Complexity:* O(n)

*Extra Space:* O(n)

**Method 3 ( Space Otimized Method 2 )**

We can optimize the space used in method 2 by storing the previous two numbers only

because that is all we need to get the next Fibannaci number in series.

#include<stdio.h>

**int** fib(**int** n)

{

/\* Declare an array to store fibonacci numbers. \*/

**int** f[n+1];

**int** i;

/\* 0th and 1st number of the series are 0 and 1\*/

f[0] = 0;

f[1] = 1;

**for** (i = 2; i <= n; i++)

{

/\* Add the previous 2 numbers in the series

and store it \*/

f[i] = f[i-1] + f[i-2];

}

**return** f[n];

}

**int** main ()

{

**int** n = 9;

**printf**("%d", fib(n));

**getchar**();

**return** 0;

}

#include<stdio.h>

**int** fib(**int** n)

{

**int** a = 0, b = 1, c, i;

**if**( n == 0)

**return** a;

**for** (i = 2; i <= n; i++)

{

c = a + b;

a = b;

b = c;

}

**return** b;

}

**int** main ()

{

**int** n = 9;

**printf**("%d", fib(n));

**getchar**();

**return** 0;

}

*Time Complexity:* O(n)

*Extra Space:* O(1)

**Method 4 ( Using power of the matrix {{1,1},{1,0}} )**

This another O(n) which relies on the fact that if we n times multiply the matrix M = {{1,1},

{1,0}} to itself (in other words calculate power(M, n )), then we get the (n+1)th Fibonacci

number as the element at row and column (0, 0) in the resultant matrix.

The matrix representation gives the following closed expression for the Fibonacci

numbers:

*Time Complexity:* O(n)

*Extra Space:* O(1)

**Method 5 ( Optimized Method 4 )**

The method 4 can be optimized to work in O(Logn) time complexity. We can do

#include <stdio.h>

/\* Helper function that multiplies 2 matricies F and M of size 2\*2, and

puts the multiplication result back to F[][] \*/

**void** multiply(**int** F[2][2], **int** M[2][2]);

/\* Helper function that calculates F[][] raise to the power n and puts the

result in F[][]

Note that this function is desinged only for fib() and won't work as general

power function \*/

**void** power(**int** F[2][2], **int** n);

**int** fib(**int** n)

{

**int** F[2][2] = {{1,1},{1,0}};

**if** (n == 0)

**return** 0;

power(F, n-1);

**return** F[0][0];

}

**void** multiply(**int** F[2][2], **int** M[2][2])

{

**int** x = F[0][0]\*M[0][0] + F[0][1]\*M[1][0];

**int** y = F[0][0]\*M[0][1] + F[0][1]\*M[1][1];

**int** z = F[1][0]\*M[0][0] + F[1][1]\*M[1][0];

**int** w = F[1][0]\*M[0][1] + F[1][1]\*M[1][1];

F[0][0] = x;

F[0][1] = y;

F[1][0] = z;

F[1][1] = w;

}

**void** power(**int** F[2][2], **int** n)

{

**int** i;

**int** M[2][2] = {{1,1},{1,0}};

// n - 1 times multiply the matrix to {{1,0},{0,1}}

**for** (i = 2; i <= n; i++)

multiply(F, M);

}

/\* Driver program to test above function \*/

**int** main()

{

**int** n = 9;

**printf**("%d", fib(n));

**getchar**();

**return** 0;

}

recursive multiplication to get power(M, n) in the prevous method (Similar to the

optimization done in this post)

***Time Complexity:* O(Logn)**

*Extra Space:* O(Logn) if we consider the function call stack size, otherwise O(1).

Please write comments if you find the above codes/algorithms incorrect, or find other

ways to solve the same problem.

#include <stdio.h>

**void** multiply(**int** F[2][2], **int** M[2][2]);

**void** power(**int** F[2][2], **int** n);

/\* function that returns nth Fibonacci number \*/

**int** fib(**int** n)

{

**int** F[2][2] = {{1,1},{1,0}};

**if** (n == 0)

**return** 0;

power(F, n-1);

**return** F[0][0];

}

/\* Optimized version of power() in method 4 \*/

**void** power(**int** F[2][2], **int** n)

{

**if**( n == 0 || n == 1)

**return**;

**int** M[2][2] = {{1,1},{1,0}};

power(F, n/2);

multiply(F, F);

**if** (n%2 != 0)

multiply(F, M);

}

**void** multiply(**int** F[2][2], **int** M[2][2])

{

**int** x = F[0][0]\*M[0][0] + F[0][1]\*M[1][0];

**int** y = F[0][0]\*M[0][1] + F[0][1]\*M[1][1];

**int** z = F[1][0]\*M[0][0] + F[1][1]\*M[1][0];

**int** w = F[1][0]\*M[0][1] + F[1][1]\*M[1][1];

F[0][0] = x;

F[0][1] = y;

F[1][0] = z;

F[1][1] = w;

}

/\* Driver program to test above function \*/

**int** main()

{

**int** n = 9;

**printf**("%d", fib(9));

**getchar**();

**return** 0;

}

**References:**

http://en.wikipedia.org/wiki/Fibonacci\_number

http://www.ics.uci.edu/~eppstein/161/960109.html

11. Average of a stream of numbers

Difficulty Level: Rookie

Given a stream of numbers, print average (or mean) of the stream at every point. For

example, let us consider the stream as 10, 20, 30, 40, 50, 60, …

Average of 1 numbers is 10.00

Average of 2 numbers is 15.00

Average of 3 numbers is 20.00

Average of 4 numbers is 25.00

Average of 5 numbers is 30.00

Average of 6 numbers is 35.00

..................

To print mean of a stream, we need to find out how to find average when a new number

is being added to the stream. To do this, all we need is count of numbers seen so far in

the stream, previous average and new number. Let *n* be the count, *prev\_avg* be the

previous average and x be the new number being added. The average after including *x*

number can be written as *(prev\_avg\*n + x)/(n+1)*.

The above function getAvg() can be optimized using following changes. We can avoid

the use of prev\_avg and number of elements by using static variables (Assuming that

only this function is called for average of stream). Following is the oprimnized version.

#include <stdio.h>

// Returns the new average after including x

**float** getAvg(**float** prev\_avg, **int** x, **int** n)

{

**return** (prev\_avg\*n + x)/(n+1);

}

// Prints average of a stream of numbers

**void** streamAvg(**float** arr[], **int** n)

{

**float** avg = 0;

**for**(**int** i = 0; i < n; i++)

{

avg = getAvg(avg, arr[i], i);

**printf**("Average of %d numbers is %f \n", i+1, avg);

}

**return**;

}

// Driver program to test above functions

**int** main()

{

**float** arr[] = {10, 20, 30, 40, 50, 60};

**int** n = **sizeof**(arr)/**sizeof**(arr[0]);

streamAvg(arr, n);

**return** 0;

}

Thanks to Abhijeet Deshpande for suggesting this optimized version.

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above.

12. Check whether a given point lies inside a triangle or not

Given three corner points of a triangle, and one more point P. Write a function to check

whether P lies within the triangle or not.

For example, consider the following program, the function should return true for P(10,

15) and false for P'(30, 15)

B(10,30)

/ \

/ \

/ \

/ P \ P'

#include <stdio.h>

// Returns the new average after including x

**float** getAvg (**int** x)

{

**static int** sum, n;

sum += x;

**return** (((**float**)sum)/++n);

}

// Prints average of a stream of numbers

**void** streamAvg(**float** arr[], **int** n)

{

**float** avg = 0;

**for**(**int** i = 0; i < n; i++)

{

avg = getAvg(arr[i]);

**printf**("Average of %d numbers is %f \n", i+1, avg);

}

**return**;

}

// Driver program to test above functions

**int** main()

{

**float** arr[] = {10, 20, 30, 40, 50, 60};

**int** n = **sizeof**(arr)/**sizeof**(arr[0]);

streamAvg(arr, n);

**return** 0;

}

/ \

A(0,0) ----------- C(20,0)

Source: Microsoft Interview Question

**Solution:**

Let the coordinates of three corners be (x1, y1), (x2, y2) and (x3, y3). And coordinates

of the given point P be (x, y)

1) Calculate area of the given triangle, i.e., area of the triangle ABC in the above

diagram. Area A = [ x1(y2 – y3) + x2(y3 – y1) + x3(y1-y2)]/2

2) Calculate area of the triangle PAB. We can use the same formula for this. Let this

area be A1.

3) Calculate area of the triangle PBC. Let this area be A2.

4) Calculate area of the triangle PAC. Let this area be A3.

5) If P lies inside the triangle, then A1 + A2 + A3 must be equal to A.

Ouptut:

Inside

**Exercise:** Given coordinates of four corners of a rectangle, and a point P. Write a

function to check whether P lies inside the given rectangle or not.

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above.

13. Count numbers that don’t contain 3

#include <stdio.h>

#include <stdlib.h>

/\* A utility function to calculate area of triangle formed by (x1, y1),

(x2, y2) and (x3, y3) \*/

**float** area(**int** x1, **int** y1, **int** x2, **int** y2, **int** x3, **int** y3)

{

**return abs**((x1\*(y2-y3) + x2\*(y3-y1)+ x3\*(y1-y2))/2.0);

}

/\* A function to check whether point P(x, y) lies inside the triangle formed

by A(x1, y1), B(x2, y2) and C(x3, y3) \*/

**bool** isInside(**int** x1, **int** y1, **int** x2, **int** y2, **int** x3, **int** y3, **int** x, **int**

{

/\* Calculate area of triangle ABC \*/

**float** A = area (x1, y1, x2, y2, x3, y3);

/\* Calculate area of triangle PBC \*/

**float** A1 = area (x, y, x2, y2, x3, y3);

/\* Calculate area of triangle PAC \*/

**float** A2 = area (x1, y1, x, y, x3, y3);

/\* Calculate area of triangle PAB \*/

**float** A3 = area (x1, y1, x2, y2, x, y);

/\* Check if sum of A1, A2 and A3 is same as A \*/

**return** (A == A1 + A2 + A3);

}

/\* Driver program to test above function \*/

**int** main()

{

/\* Let us check whether the point P(10, 15) lies inside the triangle

formed by A(0, 0), B(20, 0) and C(10, 30) \*/

**if** (isInside(0, 0, 20, 0, 10, 30, 10, 15))

**printf** ("Inside");

**else**

**printf** ("Not Inside");

**return** 0;

}

Given a number n, write a function that returns count of numbers from 1 to n that don’t

contain digit 3 in their decimal representation.

Examples:

Input: n = 10

Output: 9

Input: n = 45

Output: 31

// Numbers 3, 13, 23, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 43 contain digit 3.

Input: n = 578

Ouput: 385

**Solution:**

We can solve it recursively. Let count(n) be the function that counts such numbers.

'msd' --> the most significant digit in n

'd' --> number of digits in n.

count(n) = n if n < 3

count(n) = n - 1 if 3 <= n < 10

count(n) = count(msd) \* count(10^(d-1) - 1) +

count(msd) +

count(n % (10^(d-1)))

if n > 10 and msd is not 3

count(n) = count( msd \* (10^(d-1)) - 1)

if n > 10 and msd is 3

Let us understand the solution with n = 578.

count(578) = 4\*count(99) + 4 + count(78)

The middle term 4 is added to include numbers 100, 200, 400 and 500.

Let us take n = 35 as another example.

count(35) = count (3\*10 - 1) = count(29)

Output:

385

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above

14. Magic Square

A magic square of order n is an arrangement of n^2 numbers, usually distinct integers, in

a square, such that the n numbers in all rows, all columns, and both diagonals sum to the

same constant. A magic square contains the integers from 1 to n^2.

The constant sum in every row, column and diagonal is called the magic constant or

magic sum, M. The magic constant of a normal magic square depends only on n and

#include <stdio.h>

/\* returns count of numbers which are in range from 1 to n and don't contain 3

as a digit \*/

**int** count(**int** n)

{

// Base cases (Assuming n is not negative)

**if** (n < 3)

**return** n;

**if** (n >= 3 && n < 10)

**return** n-1;

// Calculate 10^(d-1) (10 raise to the power d-1) where d is

// number of digits in n. po will be 100 for n = 578

**int** po = 1;

**while** (n/po > 9)

po = po\*10;

// find the most significant digit (msd is 5 for 578)

**int** msd = n/po;

**if** (msd != 3)

// For 578, total will be 4\*count(10^2 - 1) + 4 + count(78)

**return** count(msd)\*count(po - 1) + count(msd) + count(n%po);

**else**

// For 35, total will be equal to count(29)

**return** count(msd\*po - 1);

}

// Driver program to test above function

**int** main()

{

**printf** ("%d ", count(578));

**return** 0;

}

has the following value:

M = n(n^2+1)/2

For normal magic squares of order n = 3, 4, 5, …, the magic constants are: 15, 34, 65,

111, 175, 260, …

In this post, we will discuss how programmatically we can generate a magic square of

size n. Before we go further, consider the below examples:

Magic Square of size 3

-----------------------

2 7 6

9 5 1

4 3 8

Sum in each row & each column = 3\*(3^2+1)/2 = 15

Magic Square of size 5

----------------------

9 3 22 16 15

2 21 20 14 8

25 19 13 7 1

18 12 6 5 24

11 10 4 23 17

Sum in each row & each column = 5\*(5^2+1)/2 = 65

Magic Square of size 7

----------------------

20 12 4 45 37 29 28

11 3 44 36 35 27 19

2 43 42 34 26 18 10

49 41 33 25 17 9 1

40 32 24 16 8 7 48

31 23 15 14 6 47 39

22 21 13 5 46 38 30

Sum in each row & each column = 7\*(7^2+1)/2 = 175

Did you find any pattern in which the numbers are stored?

In any magic square, the first number i.e. 1 is stored at position (n/2, n-1). Let this

position be (i,j). The next number is stored at position (i-1, j+1) where we can consider

each row & column as circular array i.e. they wrap around.

Three conditions hold:

1. The position of next number is calculated by decrementing row number of previous

number by 1, and incrementing the column number of previous number by 1. At any time,

if the calculated row position becomes -1, it will wrap around to n-1. Similarly, if the

calculated column position becomes n, it will wrap around to 0.

2. If the magic square already contains a number at the calculated position, calculated

column position will be decremented by 2, and calculated row position will be

incremented by 1.

3. If the calculated row position is -1 & calculated column position is n, the new position

would be: (0, n-2).

Example:

Magic Square of size 3

----------------------

2 7 6

9 5 1

4 3 8

Steps:

1. position of number 1 = (3/2, 3-1) = (1, 2)

2. position of number 2 = (1-1, 2+1) = (0, 0)

3. position of number 3 = (0-1, 0+1) = (3-1, 1) = (2, 1)

4. position of number 4 = (2-1, 1+1) = (1, 2)

Since, at this position, 1 is there. So, apply condition 2.

new position=(1+1,2-2)=(2,0)

5. position of number 5=(2-1,0+1)=(1,1)

6. position of number 6=(1-1,1+1)=(0,2)

7. position of number 7 = (0-1, 2+1) = (-1,3) // this is tricky, see condition 3

new position = (0, 3-2) = (0,1)

8. position of number 8=(0-1,1+1)=(-1,2)=(2,2) //wrap around

9. position of number 9=(2-1,2+1)=(1,3)=(1,0) //wrap around

Based on the above approach, following is the working code:

#include<stdio.h>

#include<string.h>

// A function to generate odd sized magic squares

**void** generateSquare(**int** n)

{

**int** magicSquare[n][n];

// set all slots as 0

**memset**(magicSquare, 0, **sizeof**(magicSquare));

// Initialize position for 1

**int** i = n/2;

**int** j = n-1;

// One by one put all values in magic square

**for** (**int** num=1; num <= n\*n; )

{

**if** (i==-1 && j==n) //3rd condition

Output:

The Magic Square for n=7:

Sum of each row or column 175:

20 12 4 45 37 29 28

11 3 44 36 35 27 19

2 43 42 34 26 18 10

49 41 33 25 17 9 1

40 32 24 16 8 7 48

31 23 15 14 6 47 39

22 21 13 5 46 38 30

**if** (i==-1 && j==n) //3rd condition

{

j = n-2;

i = 0;

}

**else**

{

//1st condition helper if next number goes to out of square's right side

**if** (j == n)

j = 0;

//1st condition helper if next number is goes to out of square's upper **if** (i < 0)

i=n-1;

}

**if** (magicSquare[i][j]) //2nd condition

{

j -= 2;

i++;

**continue**;

}

**else**

magicSquare[i][j] = num++; //set number

j++; i--; //1st condition

}

// print magic square

**printf**("The Magic Square for n=%d:\nSum of each row or column %d:\n\n"

n, n\*(n\*n+1)/2);

**for**(i=0; i<n; i++)

{

**for**(j=0; j<n; j++)

**printf**("%3d ", magicSquare[i][j]);

**printf**("\n");

}

}

// Driver program to test above function

**int** main()

{

**int** n = 7; // Works only when n is odd

generateSquare (n);

**return** 0;

}

NOTE: This approach works only for odd values of n.

References:

http://en.wikipedia.org/wiki/Magic\_square

This article is compiled by **Aashish Barnwal** and reviewed by GeeksforGeeks team.

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above

15. Sieve of Eratosthenes

Given a number n, print all primes smaller than or equal to n. It is also given that n is a

small number.

For example, if n is 10, the output should be “2, 3, 5, 7″. If n is 20, the output should be

“2, 3, 5, 7, 11, 13, 17, 19″.

The sieve of Eratosthenes is one of the most efficient ways to find all primes smaller

than n when n is smaller than 10 million or so (Ref Wiki).

Following is the algorithm to find all the prime numbers less than or equal to a given

integer *n* by Eratosthenes’ method:

1. Create a list of consecutive integers from 2 to *n*: (2, 3, 4, …, *n*).

2. Initially, let *p* equal 2, the first prime number.

3. Starting from *p*, count up in increments of *p* and mark each of these numbers

greater than *p* itself in the list. These numbers will be 2*p*, 3*p*, 4*p*, etc.; note that some

of them may have already been marked.

4. Find the first number greater than *p* in the list that is not marked. If there was no such

number, stop. Otherwise, let *p* now equal this number (which is the next prime), and

repeat from step 3.

When the algorithm terminates, all the numbers in the list that are not marked are prime.

Following is C++ implementation of the above algorithm. In the following

implementation, a boolean array arr[] of size n is used to mark multiples of prime

numbers.

Output:

Following are the prime numbers below 30

2 3 5 7 11 13 17 19 23 29

References:

http://en.wikipedia.org/wiki/Sieve\_of\_Eratosthenes

This article is compiled by **Abhinav Priyadarshi** and reviewed by GeeksforGeeks team.

Please write comments if you find anything incorrect, or you want to share more

#include <stdio.h>

#include <string.h>

// marks all mutiples of 'a' ( greater than 'a' but less than equal to 'n') as 1.

**void** markMultiples(**bool** arr[], **int** a, **int** n)

{

**int** i = 2, num;

**while** ( (num = i\*a) <= n )

{

arr[ num-1 ] = 1; // minus 1 because index starts from 0.

++i;

}

}

// A function to print all prime numbers smaller than n

**void** SieveOfEratosthenes(**int** n)

{

// There are no prime numbers smaller than 2

**if** (n >= 2)

{

// Create an array of size n and initialize all elements as 0

**bool** arr[n];

**memset**(arr, 0, **sizeof**(arr));

/\* Following property is maintained in the below for loop

arr[i] == 0 means i + 1 is prime

arr[i] == 1 means i + 1 is not prime \*/

**for** (**int** i=1; i<n; ++i)

{

**if** ( arr[i] == 0 )

{

//(i+1) is prime, print it and mark its multiples

**printf**("%d ", i+1);

markMultiples(arr, i+1, n);

}

}

}

}

// Driver Program to test above function

**int** main()

{

**int** n = 30;

**printf**("Following are the prime numbers below %d\n", n);

SieveOfEratosthenes(n);

**return** 0;

}

information about the topic discussed above

16. Find day of the week for a given date

Write a function that calculates the day of the week for any particular date in the past or

future. A typical application is to calculate the day of the week on which someone was

born or some other special event occurred.

Following is a simple C function suggested by Sakamoto, Lachman, Keith and Craver to

calculate day. The following function returns 0 for Sunday, 1 for Monday, etc.

Output: 1 (Monday)

See this for explanation of the above function.

References:

http://en.wikipedia.org/wiki/Determination\_of\_the\_day\_of\_the\_week

This article is compiled by **Dheeraj Jain** and reviewed by GeeksforGeeks team. Please

write comments if you find anything incorrect, or you want to share more information

about the topic discussed above

17. DFA based division

Deterministic Finite Automaton (DFA) can be used to check whether a number “num” is

/\* A program to find day of a given date \*/

#include<stdio.h>

**int** dayofweek(**int** d, **int** m, **int** y)

{

**static int** t[] = { 0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4 };

y -= m < 3;

**return** ( y + y/4 - y/100 + y/400 + t[m-1] + d) % 7;

}

/\* Driver function to test above function \*/

**int** main()

{

**int** day = dayofweek(30, 8, 2010);

**printf** ("%d", day);

**return** 0;

}

divisible by “k” or not. If the number is not divisible, remainder can also be obtained

using DFA.

We consider the binary representation of ‘num’ and build a DFA with k states. The DFA

has transition function for both 0 and 1. Once the DFA is built, we process ‘num’ over the

DFA to get remainder.

Let us walk through an example. Suppose we want to check whether a given number

‘num’ is divisible by 3 or not. Any number can be written in the form: num = 3\*a + b

where ‘a’ is the quotient and ‘b’ is the remainder.

For 3, there can be 3 states in DFA, each corresponding to remainder 0, 1 and 2. And

each state can have two transitions corresponding 0 and 1 (considering the binary

representation of given ‘num’).

The transition function F(p, x) = q tells that on reading alphabet x, we move from state p

to state q. Let us name the states as 0, 1 and 2. The initial state will always be 0. The

final state indicates the remainder. If the final state is 0, the number is divisible.

In the above diagram, double circled state is final state.

1. When we are at state 0 and read 0, we remain at state 0.

2. When we are at state 0 and read 1, we move to state 1, why? The number so

formed(1) in decimal gives remainder 1.

3. When we are at state 1 and read 0, we move to state 2, why? The number so

formed(10) in decimal gives remainder 2.

4. When we are at state 1 and read 1, we move to state 0, why? The number so

formed(11) in decimal gives remainder 0.

5. When we are at state 2 and read 0, we move to state 1, why? The number so

formed(100) in decimal gives remainder 1.

6. When we are at state 2 and read 1, we remain at state 2, why? The number so

formed(101) in decimal gves remainder 2.

The transition table looks like following:

state 0 1

\_\_\_\_\_\_\_\_\_\_\_\_\_

0 0 1

1 2 0

2 1 2

Let us check whether 6 is divisible by 3?

Binary representation of 6 is 110

state = 0

1. state=0, we read 1, new state=1

2. state=1, we read 1, new state=0

3. state=0, we read 0, new state=0

Since the final state is 0, the number is divisible by 3.

Let us take another example number as 4

state=0

1. state=0, we read 1, new state=1

2. state=1, we read 0, new state=2

3. state=2, we read 0, new state=1

Since, the final state is not 0, the number is not divisible by 3. The remainder is 1.

*Note that the final state gives the remainder.*

We can extend the above solution for any value of k. For a value k, the states would be

0, 1, …. , k-1. How to calculate the transition if the decimal equivalent of the binary bits

seen so far, crosses the range k? If we are at state p, we have read p (in decimal). Now

we read 0, new read number becomes 2\*p. If we read 1, new read number becomes

2\*p+1. The new state can be obtained by subtracting k from these values (2p or 2p+1)

where 0 <= p < k.

Based on the above approach, following is the working code:

#include <stdio.h>

#include <stdlib.h>

// Function to build DFA for divisor k

**void** preprocess(**int** k, **int** Table[][2])

{

**int** trans0, trans1;

// The following loop calculates the two transitions for each state,

// starting from state 0

**for** (**int** state=0; state<k; ++state)

{

// Calculate next state for bit 0

trans0 = state<<1;

Table[state][0] = (trans0 < k)? trans0: trans0-k;

// Calculate next state for bit 1

trans1 = (state<<1) + 1;

Table[state][1] = (trans1 < k)? trans1: trans1-k;

}

}

// A recursive utility function that takes a ‘num’ and DFA (transition

// table) as input and process ‘num’ bit by bit over DFA

**void** isDivisibleUtil(**int** num, **int**\* state, **int** Table[][2])

{

// process "num" bit by bit from MSB to LSB

**if** (num != 0)

{

isDivisibleUtil(num>>1, state, Table);

\*state = Table[\*state][num&1];

}

}

Output:

Not Divisible: Remainder is 2

DFA based division can be useful if we have a binary stream as input and we want to

check for divisibility of the decimal value of stream at any time.

This article is compiled by **Aashish Barnwal** and reviewed by GeeksforGeeks team.

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above

18. Generate integer from 1 to 7 with equal probability

Given a function foo() that returns integers from 1 to 5 with equal probability, write a

function that returns integers from 1 to 7 with equal probability using foo() only. Minimize

the number of calls to foo() method. Also, use of any other library function is not allowed

and no floating point arithmetic allowed.

// The main function that divides ‘num’ by k and returns the remainder

**int** isDivisible (**int** num, **int** k)

{

// Allocate memory for transition table. The table will have k\*2 entries

**int** (\*Table)[2] = (**int** (\*)[2])**malloc**(k\***sizeof**(\*Table));

// Fill the transition table

preprocess(k, Table);

// Process ‘num’ over DFA and get the remainder

**int** state = 0;

isDivisibleUtil(num, &state, Table);

// Note that the final value of state is the remainder

**return** state;

}

// Driver program to test above functions

**int** main()

{

**int** num = 47; // Number to be divided

**int** k = 5; // Divisor

**int** remainder = isDivisible (num, k);

**if** (remainder == 0)

**printf**("Divisible\n");

**else**

**printf**("Not Divisible: Remainder is %d\n", remainder);

**return** 0;

}

**Solution:**

We know foo() returns integers from 1 to 5. How we can ensure that integers from 1 to 7

occur with equal probability?

If we somehow generate integers from 1 to a-multiple-of-7 (like 7, 14, 21, …) with equal

probability, we can use modulo division by 7 followed by adding 1 to get the numbers

from 1 to 7 with equal probability.

We can generate from 1 to 21 with equal probability using the following expression.

5\*foo() + foo() -5

Let us see how above expression can be used.

1. For each value of first foo(), there can be 5 possible combinations for values of

second foo(). So, there are total 25 combinations possible.

2. The range of values returned by the above equation is 1 to 25, each integer occurring

exactly once.

3. If the value of the equation comes out to be less than 22, return modulo division by 7

followed by adding 1. Else, again call the method recursively. The probability of returning

each integer thus becomes 1/7.

The below program shows that the expression returns each integer from 1 to 25 exactly

once.

Output:

1

2

.

.

24

25

The below program depicts how we can use foo() to return 1 to 7 with equal probability.

#include <stdio.h>

**int** main()

{

**int** first, second;

**for** ( first=1; first<=5; ++first )

**for** ( second=1; second<=5; ++second )

**printf** ("%d \n", 5\*first + second - 5);

**return** 0;

}

This article is compiled by **Aashish Barnwal** and reviewed by GeeksforGeeks team.

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above

19. Given a number, find the next smallest palindrome

Given a number, find the next smallest palindrome larger than this number. For example,

if the input number is “2 3 5 4 5″, the output should be “2 3 6 3 2″. And if the input number

is “9 9 9″, the output should be “1 0 0 1″.

The input is assumed to be an array. Every entry in array represents a digit in input

number. Let the array be ‘num[]’ and size of array be ‘n’

There can be three different types of inputs that need to be handled separately.

**1)** The input number is palindrome and has all 9s. For example “9 9 9″. Output should be

“1 0 0 1″

**2)** The input number is not palindrome. For example “1 2 3 4″. Output should be “1 3 3 1″

**3)** The input number is palindrome and doesn’t have all 9s. For example “1 2 2 1″. Output

should be “1 3 3 1″.

Solution for input type 1 is easy. The output contains n + 1 digits where the corner digits

are 1, and all digits between corner digits are 0.

Now let us first talk about input type 2 and 3. How to convert a given number to a greater

palindrome? To understand the solution, let us first define the following two terms:

*Left Side:* The left half of given number. Left side of “1 2 3 4 5 6″ is “1 2 3″ and left side

#include <stdio.h>

**int** foo() // given method that returns 1 to 5 with equal probability

{

// some code here

}

**int** my\_rand() // returns 1 to 7 with equal probability

{

**int** i;

i = 5\*foo() + foo() - 5;

**if** (i < 22)

**return** i%7 + 1;

**return** my\_rand();

}

**int** main()

{

**printf** ("%d ", my\_rand());

**return** 0;

}

of “1 2 3 4 5″ is “1 2″

*Right Side:* The right half of given number. Right side of “1 2 3 4 5 6″ is “4 5 6″ and right

side of “1 2 3 4 5″ is “4 5″

To convert to palindrome, we can either take the mirror of its left side or take mirror of

its right side. However, if we take the mirror of the right side, then the palindrome so

formed is not guaranteed to be next larger palindrome. So, we must take the mirror of

left side and copy it to right side. But there are some cases that must be handled in

different ways. See the following steps.

We will start with two indices i and j. i pointing to the two middle elements (or pointing to

two elements around the middle element in case of n being odd). We one by one move i

and j away from each other.

**Step 1.** Initially, ignore the part of left side which is same as the corresponding part of

right side. For example, if the number is “8 3 **4 2 2 4** 6 9″, we ignore the middle four

elements. i now points to element 3 and j now points to element 6.

**Step 2.** After step 1, following cases arise:

**Case 1:** Indices i & j cross the boundary.

This case occurs when the input number is palindrome. In this case, we just add 1 to the

middle digit (or digits in case n is even) propagate the carry towards MSB digit of left

side and simultaneously copy mirror of the left side to the right side.

For example, if the given number is “1 2 9 2 1″, we increment 9 to 10 and propagate the

carry. So the number becomes “1 3 0 3 1″

**Case 2:** There are digits left between left side and right side which are not same. So, we

just mirror the left side to the right side & try to minimize the number formed to

guarantee the next smallest palindrome.

In this case, there can be **two sub-cases**.

**2.1)** Copying the left side to the right side is sufficient, we don’t need to increment any

digits and the result is just mirror of left side. Following are some examples of this subcase.

Next palindrome for “7 **8** 3 3 2 2″ is “7 8 3 3 8 7″

Next palindrome for “1 2 **5** 3 2 2″ is “1 2 5 5 2 1″

Next palindrome for “1 4 **5** 8 7 6 7 8 3 2 2″ is “1 4 5 8 7 6 7 8 5 4 1″

How do we check for this sub-case? All we need to check is the digit just after the

ignored part in step 1. This digit is highlighted in above examples. If this digit is greater

than the corresponding digit in right side digit, then copying the left side to the right side

is sufficient and we don’t need to do anything else.

**2.2)** Copying the left side to the right side is NOT sufficient. This happens when the

above defined digit of left side is smaller. Following are some examples of this case.

Next palindrome for “7 **1** 3 3 2 2″ is “7 1 4 4 1 7″

Next palindrome for “1 2 **3** 4 6 2 8″ is “1 2 3 5 3 2 1″

Next palindrome for “9 4 **1** 8 7 9 7 8 3 2 2″ is “9 4 1 8 8 0 8 8 1 4 9″

We handle this subcase like Case 1. We just add 1 to the middle digit (or digits in ase n

is even) propagate the carry towards MSB digit of left side and simultaneously copy

mirror of the left side to the right side.

#include <stdio.h>

// A utility function to print an array

**void** printArray (**int** arr[], **int** n);

// A utility function to check if num has all 9s

**int** AreAll9s (**int** num[], **int** n );

// Returns next palindrome of a given number num[].

// This function is for input type 2 and 3

**void** generateNextPalindromeUtil (**int** num[], **int** n )

{

// find the index of mid digit

**int** mid = n/2;

// A bool variable to check if copy of left side to right is sufficient or not

**bool** leftsmaller = **false**;

// end of left side is always 'mid -1'

**int** i = mid - 1;

// Begining of right side depends if n is odd or even

**int** j = (n % 2)? mid + 1 : mid;

// Initially, ignore the middle same digits

**while** (i >= 0 && num[i] == num[j])

i--,j++;

// Find if the middle digit(s) need to be incremented or not (or copying left

// side is not sufficient)

**if** ( i < 0 || num[i] < num[j])

leftsmaller = **true**;

// Copy the mirror of left to tight

**while** (i >= 0)

{

num[j] = num[i];

j++;

i--;

}

// Handle the case where middle digit(s) must be incremented.

// This part of code is for CASE 1 and CASE 2.2

**if** (leftsmaller == **true**)

{

**int** carry = 1;

i = mid - 1;

// If there are odd digits, then increment

// the middle digit and store the carry

**if** (n%2 == 1)

{

num[mid] += carry;

carry = num[mid] / 10;

num[mid] %= 10;

j = mid + 1;

}

**else**

**else**

j = mid;

// Add 1 to the rightmost digit of the left side, propagate the carry

// towards MSB digit and simultaneously copying mirror of the left side

// to the right side.

**while** (i >= 0)

{

num[i] += carry;

carry = num[i] / 10;

num[i] %= 10;

num[j++] = num[i--]; // copy mirror to right

}

}

}

// The function that prints next palindrome of a given number num[]

// with n digits.

**void** generateNextPalindrome( **int** num[], **int** n )

{

**int** i;

**printf**("\nNext palindrome is:\n");

// Input type 1: All the digits are 9, simply o/p 1

// followed by n-1 0's followed by 1.

**if**( AreAll9s( num, n ) )

{

**printf**( "1 ");

**for**( i = 1; i < n; i++ )

**printf**( "0 " );

**printf**( "1" );

}

// Input type 2 and 3

**else**

{

generateNextPalindromeUtil ( num, n );

// print the result

printArray (num, n);

}

}

// A utility function to check if num has all 9s

**int** AreAll9s( **int**\* num, **int** n )

{

**int** i;

**for**( i = 0; i < n; ++i )

**if**( num[i] != 9 )

**return** 0;

**return** 1;

}

/\* Utility that prints out an array on a line \*/

**void** printArray(**int** arr[], **int** n)

{

**int** i;

**for** (i=0; i < n; i++)

**printf**("%d ", arr[i]);

**printf**("\n");

}

Output:

Next palindrome is:

9 4 1 8 8 0 8 8 1 4 9

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above.

20. Make a fair coin from a biased coin

You are given a function foo() that represents a biased coin. When foo() is called, it

returns 0 with 60% probability, and 1 with 40% probability. Write a new function that

returns 0 and 1 with 50% probability each. Your function should use only foo(), no other

library method.

**Solution:**

We know foo() returns 0 with 60% probability. How can we ensure that 0 and 1 are

returned with 50% probability?

The solution is similar to this post. If we can somehow get two cases with equal

probability, then we are done. We call foo() two times. Both calls will return 0 with 60%

probability. So the two pairs (0, 1) and (1, 0) will be generated with equal probability from

two calls of foo(). Let us see how.

**(0, 1):** The probability to get 0 followed by 1 from two calls of foo() = 0.6 \* 0.4 = 0.24

**(1, 0):** The probability to get 1 followed by 0 from two calls of foo() = 0.4 \* 0.6 = 0.24

*So the two cases appear with equal probability. The idea is to return consider only the*

*above two cases, return 0 in one case, return 1 in other case. For other cases [(0, 0)*

*and (1, 1)], recur until you end up in any of the above two cases.*

The below program depicts how we can use foo() to return 0 and 1 with equal

probability.

// Driver Program to test above function

**int** main()

{

**int** num[] = {9, 4, 1, 8, 7, 9, 7, 8, 3, 2, 2};

**int** n = **sizeof** (num)/ **sizeof**(num[0]);

generateNextPalindrome( num, n );

**return** 0;

}

References:

http://en.wikipedia.org/wiki/Fair\_coin#Fair\_results\_from\_a\_biased\_coin

This article is compiled by **Shashank Sinha** and reviewed by GeeksforGeeks team.

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above.

If you like GeeksforGeeks and would like to contribute, you can also write an article and

mail your article to contribute@geeksforgeeks.org. See your article appearing on the

GeeksforGeeks main page and help other Geeks.

21. Check divisibility by 7

Given a number, check if it is divisible by 7. You are not allowed to use modulo operator,

floating point arithmetic is also not allowed.

A simple method is repeated subtraction. Following is another interesting method.

Divisibility by 7 can be checked by a recursive method. A number of the form 10a + b is

divisible by 7 if and only if a – 2b is divisible by 7. In other words, subtract twice the last

digit from the number formed by the remaining digits. Continue to do this until a small

number.

**Example:** the number 371: 37 – (2×1) = 37 – 2 = 35; 3 – (2 × 5) = 3 – 10 = -7; thus,

#include <stdio.h>

**int** foo() // given method that returns 0 with 60% probability and 1 with 40%

{

// some code here

}

// returns both 0 and 1 with 50% probability

**int** my\_fun()

{

**int** val1 = foo();

**int** val2 = foo();

**if** (val1 == 0 && val2 == 1)

**return** 0; // Will reach here with 0.24 probability

**if** (val1 == 1 && val2 == 0)

**return** 1; // // Will reach here with 0.24 probability

**return** my\_fun(); // will reach here with (1 - 0.24 - 0.24) probability

}

**int** main()

{

**printf** ("%d ", my\_fun());

**return** 0;

}

since -7 is divisible by 7, 371 is divisible by 7.

Following is C implementation of the above method

Output:

Divisible

**How does this work?** Let ‘b’ be the last digit of a number ‘n’ and let ‘a’ be the number

we get when we split off ‘b’.

The representation of the number may also be multiplied by any number relatively prime

to the divisor without changing its divisibility. After observing that 7 divides 21, we can

perform the following:

10.a + b

after multiplying by 2, this becomes

20.a + 2.b

and then

21.a - a + 2.b

Eliminating the multiple of 21 gives

-a + 2b

// A Program to check whether a number is divisible by 7

#include <stdio.h>

**int** isDivisibleBy7( **int** num )

{

// If number is negative, make it positive

**if**( num < 0 )

**return** isDivisibleBy7( -num );

// Base cases

**if**( num == 0 || num == 7 )

**return** 1;

**if**( num < 10 )

**return** 0;

// Recur for ( num / 10 - 2 \* num % 10 )

**return** isDivisibleBy7( num / 10 - 2 \* ( num - num / 10 \* 10 ) );

}

// Driver program to test above function

**int** main()

{

**int** num = 616;

**if**( isDivisibleBy7(num ) )

**printf**( "Divisible" );

**else**

**printf**( "Not Divisible" );

**return** 0;

}

and multiplying by -1 gives

a - 2b

There are other interesting methods to check divisibility by 7 and other numbers. See

following Wiki page for details.

**References:**

http://en.wikipedia.org/wiki/Divisibility\_rule

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above.

22. Find the largest multiple of 3

Given an array of non-negative integers. Find the largest multiple of 3 that can be

formed from array elements.

For example, if the input array is {8, 1, 9}, the output should be “9 8 1″, and if the input

array is {8, 1, 7, 6, 0}, output should be “8 7 6 0″.

**Method 1 (Brute Force)**

The simple & straight forward approach is to generate all the combinations of the

elements and keep track of the largest number formed which is divisible by 3.

Time Complexity: O(n x 2^n). There will be 2^n combinations of array elements. To

compare each combination with the largest number so far may take O(n) time.

Auxiliary Space: O(n) // to avoid integer overflow, the largest number is assumed to be

stored in the form of array.

**Method 2 (Tricky)**

This problem can be solved efficiently with the help of O(n) extra space. This method is

based on the following facts about numbers which are multiple of 3.

**1)** A number is multiple of 3 if and only if the sum of digits of number is multiple of 3. For

example, let us consider 8760, it is a multiple of 3 because sum of digits is 8 + 7+ 6+ 0

= 21, which is a multiple of 3.

**2)** If a number is multiple of 3, then all permutations of it are also multiple of 3. For

example, since 6078 is a multiple of 3, the numbers 8760, 7608, 7068, ….. are also

multiples of 3.

**3)** We get the same remainder when we divide the number and sum of digits of the

number. For example, if divide number 151 and sum of it digits 7, by 3, we get the same

remainder 1.

*What is the idea behind above facts?*

The value of 10%3 and 100%3 is 1. The same is true for all the higher powers of 10,

because 3 divides 9, 99, 999, … etc.

Let us consider a 3 digit number n to prove above facts. Let the first, second and third

digits of n be ‘a’, ‘b’ and ‘c’ respectively. n can be written as

n = 100.a + 10.b + c

Since (10^x)%3 is 1 for any x, the above expression gives the same remainder as

following expression

1.a + 1.b + c

So the remainder obtained by sum of digits and ‘n’ is same.

Following is a solution based on the above observation.

**1.** Sort the array in non-decreasing order.

**2.** Take three queues. One for storing elements which on dividing by 3 gives remainder

as 0.The second queue stores digits which on dividing by 3 gives remainder as 1. The

third queue stores digits which on dividing by 3 gives remainder as 2. Call them as

queue0, queue1 and queue2

**3.** Find the sum of all the digits.

**4.** Three cases arise:

……**4.1** The sum of digits is divisible by 3. Dequeue all the digits from the three queues.

Sort them in non-increasing order. Output the array.

……**4.2** The sum of digits produces remainder 1 when divided by 3.

Remove one item from queue1. If queue1 is empty, remove two items from queue2. If

queue2 contains less than two items, the number is not possible.

……**4.3** The sum of digits produces remainder 2 when divided by 3.

Remove one item from queue2. If queue2 is empty, remove two items from queue1. If

queue1 contains less than two items, the number is not possible.

**5.** Finally empty all the queues into an auxiliary array. Sort the auxiliary array in nonincreasing

order. Output the auxiliary array.

Based on the above, below is the implementation:

/\* A program to find the largest multiple of 3 from an array of elements \*/

#include <stdio.h>

#include <stdlib.h>

// A queue node

**typedef struct** Queue

{

{

**int** front;

**int** rear;

**int** capacity;

**int**\* array;

} Queue;

// A utility function to create a queue with given capacity

Queue\* createQueue( **int** capacity )

{

Queue\* queue = (Queue \*) **malloc** (**sizeof**(Queue));

queue->capacity = capacity;

queue->front = queue->rear = -1;

queue->array = (**int** \*) **malloc** (queue->capacity \* **sizeof**(**int**));

**return** queue;

}

// A utility function to check if queue is empty

**int** isEmpty (Queue\* queue)

{

**return** queue->front == -1;

}

// A function to add an item to queue

**void** Enqueue (Queue\* queue, **int** item)

{

queue->array[ ++queue->rear ] = item;

**if** ( isEmpty(queue) )

++queue->front;

}

// A function to remove an item from queue

**int** Dequeue (Queue\* queue)

{

**int** item = queue->array[ queue->front ];

**if**( queue->front == queue->rear )

queue->front = queue->rear = -1;

**else**

queue->front++;

**return** item;

}

// A utility function to print array contents

**void** printArr (**int**\* arr, **int** size)

{

**int** i;

**for** (i = 0; i< size; ++i)

**printf** ("%d ", arr[i]);

}

/\* Following two functions are needed for library function qsort().

Refer following link for help of qsort()

http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ \*/

**int** compareAsc( **const void**\* a, **const void**\* b )

{

**return** \*(**int**\*)a > \*(**int**\*)b;

} **int** compareDesc( **const void**\* a, **const void**\* b

)

{

**return** \*(**int**\*)a < \*(**int**\*)b;

}

// This function puts all elements of 3 queues in the auxiliary array

// This function puts all elements of 3 queues in the auxiliary array

**void** populateAux (**int**\* aux, Queue\* queue0, Queue\* queue1,

Queue\* queue2, **int**\* top )

{

// Put all items of first queue in aux[]

**while** ( !isEmpty(queue0) )

aux[ (\*top)++ ] = Dequeue( queue0 );

// Put all items of second queue in aux[]

**while** ( !isEmpty(queue1) )

aux[ (\*top)++ ] = Dequeue( queue1 );

// Put all items of third queue in aux[]

**while** ( !isEmpty(queue2) )

aux[ (\*top)++ ] = Dequeue( queue2 );

}

// The main function that finds the largest possible multiple of

// 3 that can be formed by arr[] elements

**int** findMaxMultupleOf3( **int**\* arr, **int** size )

{

// Step 1: sort the array in non-decreasing order

**qsort**( arr, size, **sizeof**( **int** ), compareAsc );

// Create 3 queues to store numbers with remainder 0, 1

// and 2 respectively

Queue\* queue0 = createQueue( size );

Queue\* queue1 = createQueue( size );

Queue\* queue2 = createQueue( size );

// Step 2 and 3 get the sum of numbers and place them in

// corresponding queues

**int** i, sum;

**for** ( i = 0, sum = 0; i < size; ++i )

{

sum += arr[i];

**if** ( (arr[i] % 3) == 0 )

Enqueue( queue0, arr[i] );

**else if** ( (arr[i] % 3) == 1 )

Enqueue( queue1, arr[i] );

**else**

Enqueue( queue2, arr[i] );

}

// Step 4.2: The sum produces remainder 1

**if** ( (sum % 3) == 1 )

{

// either remove one item from queue1

**if** ( !isEmpty( queue1 ) )

Dequeue( queue1 );

// or remove two items from queue2

**else**

{

**if** ( !isEmpty( queue2 ) )

Dequeue( queue2 );

**else**

**return** 0;

**if** ( !isEmpty( queue2 ) )

Dequeue( queue2 );

**else**

**return** 0;

The above method can be optimized in following ways.

1) We can use Heap Sort or Merge Sort to make the time complexity O(nLogn).

2) We can avoid extra space for queues. We know at most two items will be removed

from the input array. So we can keep track of two items in two variables.

3) At the end, instead of sorting the array again in descending order, we can print the

ascending sorted array in reverse order. While printing in reverse order, we can skip the

**return** 0;

}

}

// Step 4.3: The sum produces remainder 2

**else if** ((sum % 3) == 2)

{

// either remove one item from queue2

**if** ( !isEmpty( queue2 ) )

Dequeue( queue2 );

// or remove two items from queue1

**else**

{

**if** ( !isEmpty( queue1 ) )

Dequeue( queue1 );

**else**

**return** 0;

**if** ( !isEmpty( queue1 ) )

Dequeue( queue1 );

**else**

**return** 0;

}

}

**int** aux[size], top = 0;

// Empty all the queues into an auxiliary array.

populateAux (aux, queue0, queue1, queue2, &top);

// sort the array in non-increasing order

**qsort** (aux, top, **sizeof**( **int** ), compareDesc);

// print the result

printArr (aux, top);

**return** top;

}

// Driver program to test above functions

**int** main()

{

**int** arr[] = {8, 1, 7, 6, 0};

**int** size = **sizeof**(arr)/**sizeof**(arr[0]);

**if** (findMaxMultupleOf3( arr, size ) == 0)

**printf**( "Not Possible" );

**return** 0;

}

two elements to be removed.

The above code works only if the input arrays has numbers from 0 to 9. It can be easily

extended for any positive integer array. We just have to modify the part where we sort

the array in decreasing order, at the end of code.

Time Complexity: O(nLogn), assuming a O(nLogn) algorithm is used for sorting.

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above.

23. Lexicographic rank of a string

Given a string, find its rank among all its permutations sorted lexicographically. For

example, rank of “abc” is 1, rank of “acb” is 2, and rank of “cba” is 6.

For simplicity, let us assume that the string does not contain any duplicated characters.

One simple solution is to initialize rank as 1, generate all permutations in lexicographic

order. After generating a permutation, check if the generated permutation is same as

given string, if same, then return rank, if not, then increment the rank by 1. The time

complexity of this solution will be exponential in worst case. Following is an efficient

solution.

Let the given string be “STRING”. In the input string, ‘S’ is the first character. There are

total 6 characters and 4 of them are smaller than ‘S’. So there can be 4 \* 5! smaller

strings where first character is smaller than ‘S’, like following

R X X X X X

I X X X X X

N X X X X X

G X X X X X

Now let us Fix S’ and find the smaller strings staring with ‘S’.

Repeat the same process for T, rank is 4\*5! + 4\*4! +…

Now fix T and repeat the same process for R, rank is 4\*5! + 4\*4! + 3\*3! +…

Now fix R and repeat the same process for I, rank is 4\*5! + 4\*4! + 3\*3! + 1\*2! +…

Now fix I and repeat the same process for N, rank is 4\*5! + 4\*4! + 3\*3! + 1\*2! + 1\*1! +…

Now fix N and repeat the same process for G, rank is 4\*5! + 4\*4 + 3\*3! + 1\*2! + 1\*1! +

0\*0!

Rank = 4\*5! + 4\*4! + 3\*3! + 1\*2! + 1\*1! + 0\*0! = 597

Since the value of rank starts from 1, the final rank = 1 + 597 = 598

Output

598

#include <stdio.h>

#include <string.h>

// A utility function to find factorial of n

**int** fact(**int** n)

{

**return** (n <= 1)? 1 :n \* fact(n-1);

}

// A utility function to count smaller characters on right

// of arr[low]

**int** findSmallerInRight(**char**\* str, **int** low, **int** high)

{

**int** countRight = 0, i;

**for** (i = low+1; i <= high; ++i)

**if** (str[i] < str[low])

++countRight;

**return** countRight;

}

// A function to find rank of a string in all permutations

// of characters

**int** findRank (**char**\* str)

{

**int** len = **strlen**(str);

**int** mul = fact(len);

**int** rank = 1;

**int** countRight;

**int** i;

**for** (i = 0; i < len; ++i)

{

mul /= len - i;

// count number of chars smaller than str[i]

// fron str[i+1] to str[len-1]

countRight = findSmallerInRight(str, i, len-1);

rank += countRight \* mul ;

}

**return** rank;

}

// Driver program to test above function

**int** main()

{

**char** str[] = "string";

**printf** ("%d", findRank(str));

**return** 0;

}

The time complexity of the above solution is O(n^2). We can reduce the time complexity

to O(n) by creating an auxiliary array of size 256. See following code.

// A O(n) solution for finding rank of string

#include <stdio.h>

#include <string.h>

#define MAX\_CHAR 256

// A utility function to find factorial of n

**int** fact(**int** n)

{

**return** (n <= 1)? 1 :n \* fact(n-1);

}

// Construct a count array where value at every index

// contains count of smaller characters in whole string

**void** populateAndIncreaseCount (**int**\* count, **char**\* str)

{

**int** i;

**for**( i = 0; str[i]; ++i )

++count[ str[i] ];

**for**( i = 1; i < 256; ++i )

count[i] += count[i-1];

}

// Removes a character ch from count[] array

// constructed by populateAndIncreaseCount()

**void** updatecount (**int**\* count, **char** ch)

{

**int** i;

**for**( i = ch; i < MAX\_CHAR; ++i )

--count[i];

}

// A function to find rank of a string in all permutations

// of characters

**int** findRank (**char**\* str)

{

**int** len = **strlen**(str);

**int** mul = fact(len);

**int** rank = 1, i;

**int** count[MAX\_CHAR] = {0}; // all elements of count[] are initialized with 0

// Populate the count array such that count[i] contains count of

// characters which are present in str and are smaller than i

populateAndIncreaseCount( count, str );

**for** (i = 0; i < len; ++i)

{

mul /= len - i;

// count number of chars smaller than str[i]

// fron str[i+1] to str[len-1]

rank += count[ str[i] - 1] \* mul;

// Reduce count of characters greater than str[i]

updatecount (count, str[i]);

}

**return** rank;

The above programs don’t work for duplicate characters. To make them work for

duplicate characters, find all the characters that are smaller (include equal this time also),

do the same as above but, this time divide the rank so formed by p! where p is the count

of occurrences of the repeating character.

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above.

24. Print all permutations in sorted (lexicographic) order

Given a string, print all permutations of it in sorted order. For example, if the input string

is “ABC”, then output should be “ABC, ACB, BAC, BCA, CAB, CBA”.

We have discussed a program to print all permutations in this post, but here we must

print the permutations in increasing order.

Following are the steps to print the permutations lexicographic-ally

**1.** Sort the given string in non-decreasing order and print it. The first permutation is

always the string sorted in non-decreasing order.

**2.** Start generating next higher permutation. Do it until next higher permutation is not

possible. If we reach a permutation where all characters are sorted in non-increasing

order, then that permutation is the last permutation.

**Steps to generate the next higher permutation:**

**1.** Take the previously printed permutation and find the rightmost character in it, which is

smaller than its next character. Let us call this character as ‘first character’.

**2.** Now find the ceiling of the ‘first character’. Ceiling is the smallest character on right of

‘first character’, which is greater than ‘first character’. Let us call the ceil character as

‘second character’.

**3.** Swap the two characters found in above 2 steps.

**4.** Sort the substring (in non-decreasing order) after the original index of ‘first character’.

**return** rank;

}

// Driver program to test above function

**int** main()

{

**char** str[] = "string";

**printf** ("%d", findRank(str));

**return** 0;

}

Let us consider the string “ABCDEF”. Let previously printed permutation be “DCFEBA”.

The next permutation in sorted order should be “DEABCF”. Let us understand above

steps to find next permutation. The ‘first character’ will be ‘C’. The ‘second character’ will

be ‘E’. After swapping these two, we get “DEFCBA”. The final step is to sort the

substring after the character original index of ‘first character’. Finally, we get “DEABCF”.

Following is C++ implementation of the algorithm.

// Program to print all permutations of a string in sorted order.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

/\* Following function is needed for library function qsort(). Refer

http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ \*/

**int** compare (**const void** \*a, **const void** \* b)

{ **return** ( \*(**char** \*)a - \*(**char** \*)b ); }

// A utility function two swap two characters a and b

**void** swap (**char**\* a, **char**\* b)

{

**char** t = \*a;

\*a = \*b;

\*b = t;

}

// This function finds the index of the smallest character

// which is greater than 'first' and is present in str[l..h]

**int** findCeil (**char** str[], **char** first, **int** l, **int** h)

{

// initialize index of ceiling element

**int** ceilIndex = l;

// Now iterate through rest of the elements and find

// the smallest character greater than 'first'

**for** (**int** i = l+1; i <= h; i++)

**if** (str[i] > first && str[i] < str[ceilIndex])

ceilIndex = i;

**return** ceilIndex;

}

// Print all permutations of str in sorted order

**void** sortedPermutations ( **char** str[] )

{

// Get size of string

**int** size = **strlen**(str);

// Sort the string in increasing order

**qsort**( str, size, **sizeof**( str[0] ), compare );

// Print permutations one by one

**bool** isFinished = **false**;

**while** ( ! isFinished )

{

// print this permutation

**printf** ("%s \n", str);

// Find the rightmost character which is smaller than its next

// character. Let us call it 'first char'

Output:

ABCD

ABDC

....

....

DCAB

DCBA

The upper bound on time complexity of the above program is O(n^2 x n!). We can

optimize step 4 of the above algorithm for finding next permutation. Instead of sorting

the subarray after the ‘first character’, we can reverse the subarray, because the

subarray we get after swapping is always sorted in non-increasing order. This

optimization makes the time complexity as O(n x n!). See following optimized code.

// character. Let us call it 'first char'

**int** i;

**for** ( i = size - 2; i >= 0; --i )

**if** (str[i] < str[i+1])

**break**;

// If there is no such chracter, all are sorted in decreasing order,

// means we just printed the last permutation and we are done.

**if** ( i == -1 )

isFinished = **true**;

**else**

{

// Find the ceil of 'first char' in right of first character.

// Ceil of a character is the smallest character greater than it

**int** ceilIndex = findCeil( str, str[i], i + 1, size - 1 );

// Swap first and second characters

swap( &str[i], &str[ceilIndex] );

// Sort the string on right of 'first char'

**qsort**( str + i + 1, size - i - 1, **sizeof**(str[0]), compare );

}

}

}

// Driver program to test above function

**int** main()

{

**char** str[] = "ABCD";

sortedPermutations( str );

**return** 0;

}

The above programs print duplicate permutation when characters are repeated. We can

avoid it by keeping track of the previous permutation. While printing, if the current

permutation is same as previous permutation, we won’t print it.

// An optimized version that uses reverse instead of sort for

// finding the next permutation

// A utility function to reverse a string str[l..h]

**void** reverse(**char** str[], **int** l, **int** h)

{

**while** (l < h)

{

swap(&str[l], &str[h]);

l++;

h--;

}

}

// Print all permutations of str in sorted order

**void** sortedPermutations ( **char** str[] )

{

// Get size of string

**int** size = **strlen**(str);

// Sort the string in increasing order

**qsort**( str, size, **sizeof**( str[0] ), compare );

// Print permutations one by one

**bool** isFinished = **false**;

**while** ( ! isFinished )

{

// print this permutation

**printf** ("%s \n", str);

// Find the rightmost character which is smaller than its next

// character. Let us call it 'first char'

**int** i;

**for** ( i = size - 2; i >= 0; --i )

**if** (str[i] < str[i+1])

**break**;

// If there is no such chracter, all are sorted in decreasing order,

// means we just printed the last permutation and we are done.

**if** ( i == -1 )

isFinished = **true**;

**else**

{

// Find the ceil of 'first char' in right of first character.

// Ceil of a character is the smallest character greater than it

**int** ceilIndex = findCeil( str, str[i], i + 1, size - 1 );

// Swap first and second characters

swap( &str[i], &str[ceilIndex] );

// reverse the string on right of 'first char'

reverse( str, i + 1, size - 1 );

}

}

}

This article is compiled by **Aashish Barnwal** and reviewed by GeeksforGeeks team.

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above.

25. Shuffle a given array

Given an array, write a program to generate a random permutation of array elements.

This question is also asked as “shuffle a deck of cards” or “randomize a given array”.

Let the given array be *arr[]*. A simple solution is to create an auxiliary array *temp[]* which

is initially a copy of *arr[]*. Randomly select an element from *temp[]*, copy the randomly

selected element to *arr[0]* and remove the selected element from *temp[]*. Repeat the

same process n times and keep copying elements to *arr[1], arr[2], … .* The time

complexity of this solution will be O(n^2).

Fisher–Yates shuffle Algorithm works in O(n) time complexity. The assumption here is,

we are given a function rand() that generates random number in O(1) time.

The idea is to start from the last element, swap it with a randomly selected element from

the whole array (including last). Now consider the array from 0 to n-2 (size reduced by

1), and repeat the process till we hit the first element.

Following is the detailed algorithm

To shuffle an array a of n elements (indices 0..n-1):

for i from n - 1 downto 1 do

j = random integer with 0 <= j <= i

exchange a[j] and a[i]

Following is C++ implementation of this algorithm.

Output:

7 8 4 6 3 1 2 5

The above function assumes that rand() generates a random number.

Time Complexity: O(n), assuming that the function rand() takes O(1) time.

// C Program to shuffle a given array

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// A utility function to swap to integers

**void** swap (**int** \*a, **int** \*b)

{

**int** temp = \*a;

\*a = \*b;

\*b = temp;

}

// A utility function to print an array

**void** printArray (**int** arr[], **int** n)

{

**for** (**int** i = 0; i < n; i++)

**printf**("%d ", arr[i]);

**printf**("\n");

}

// A function to generate a random permutation of arr[]

**void** randomize ( **int** arr[], **int** n )

{

// Use a different seed value so that we don't get same

// result each time we run this program

**srand** ( **time**(NULL) );

// Start from the last element and swap one by one. We don't

// need to run for the first element that's why i > 0

**for** (**int** i = n-1; i > 0; i--)

{

// Pick a random index from 0 to i

**int** j = **rand**() % (i+1);

// Swap arr[i] with the element at random index

swap(&arr[i], &arr[j]);

}

}

// Driver program to test above function.

**int** main()

{

**int** arr[] = {1, 2, 3, 4, 5, 6, 7, 8};

**int** n = **sizeof**(arr)/ **sizeof**(arr[0]);

randomize (arr, n);

printArray(arr, n);

**return** 0;

}

**How does this work?**

The probability that ith element (including the last one) goes to last position is 1/n,

because we randomly pick an element in first iteration.

The probability that ith element goes to second last position can be proved to be 1/n by

dividing it in two cases.

*Case 1: i = n-1 (index of last element)*:

The probability of last element going to second last position is = (probability that last

element doesn't stay at its original position) x (probability that the index picked in

previous step is picked again so that the last element is swapped)

So the probability = ((n-1)/n) x (1/(n-1)) = 1/n

*Case 2: 0 < i < n-1 (index of non-last)*:

The probability of ith element going to second position = (probability that ith element is

not picked in previous iteration) x (probability that ith element is picked in this iteration)

So the probability = ((n-1)/n) x (1/(n-1)) = 1/n

We can easily generalize above proof for any other position.

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above.

26. Space and time efficient Binomial Coefficient

Write a function that takes two parameters n and k and returns the value of Binomial

Coefficient C(n, k). For example, your function should return 6 for n = 4 and k = 2, and it

should return 10 for n = 5 and k = 2.

We have discussed a O(n\*k) time and O(k) extra space algorithm in this post. The value

of C(n, k) can be calculated in O(k) time and O(1) extra space.

C(n, k) = n! / (n-k)! \* k!

= [n \* (n-1) \*....\* 1] / [ ( (n-k) \* (n-k-1) \* .... \* 1) \*

( k \* (k-1) \* .... \* 1 ) ]

After simplifying, we get

C(n, k) = [n \* (n-1) \* .... \* (n-k+1)] / [k \* (k-1) \* .... \* 1]

Also, C(n, k) = C(n, n-k) // we can change r to n-r if r > n-r

Following implementation uses above formula to calculate C(n, k)

Value of C(8, 2) is 28

Time Complexity: O(k)

Auxiliary Space: O(1)

This article is compiled by Aashish Barnwal and reviewed by GeeksforGeeks team.

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above.

27. Reservoir Sampling

Reservoir sampling is a family of randomized algorithms for randomly choosing *k*

samples from a list of *n* items, where *n* is either a very large or unknown number.

Typically *n* is large enough that the list doesn’t fit into main memory. For example, a list

of search queries in Google and Facebook.

So we are given a big array (or stream) of numbers (to simplify), and we need to write

an efficient function to randomly select *k* numbers where *1 <= k <= n*. Let the input array

be *stream[].*

// Program to calculate C(n ,k)

#include <stdio.h>

// Returns value of Binomial Coefficient C(n, k)

**int** binomialCoeff(**int** n, **int** k)

{

**int** res = 1;

// Since C(n, k) = C(n, n-k)

**if** ( k > n - k )

k = n - k;

// Calculate value of [n \* (n-1) \*---\* (n-k+1)] / [k \* (k-1) \*----\* 1]

**for** (**int** i = 0; i < k; ++i)

{

res \*= (n - i);

res /= (i + 1);

}

**return** res;

}

/\* Drier program to test above function\*/

**int** main()

{

**int** n = 8, k = 2;

**printf** ("Value of C(%d, %d) is %d ", n, k, binomialCoeff(n, k) );

**return** 0;

}

A **simple solution** is to create an array *reservoir[]* of maximum size *k*. One by one

randomly select an item from *stream[0..n-1]*. If the selected item is not previously

selected, then put it in *reservoir[]*. To check if an item is previously selected or not, we

need to search the item in *reservoir[]*. The time complexity of this algorithm will be

*O(k^2)*. This can be costly if *k* is big. Also, this is not efficient if the input is in the form of

a stream.

It **can be solved in *O(n)* time**. The solution also suits well for input in the form of

stream. The idea is similar to this post. Following are the steps.

**1)** Create an array *reservoir[0..k-1]* and copy first *k* items of *stream[]* to it.

**2)** Now one by one consider all items from *(k+1)*th item to *n*th item.

…**a)** Generate a random number from 0 to *i* where *i* is index of current item in *stream[]*.

Let the generated random number is *j*.

…**b)** If *j* is in range 0 to *k-1*, replace *reservoir[j]* with *arr[i]*

Following is C implementation of the above algorithm.

Output:

Following are k randomly selected items

6 2 11 8 12

// An efficient program to randomly select k items from a stream of items

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// A utility function to print an array

**void** printArray(**int** stream[], **int** n)

{

**for** (**int** i = 0; i < n; i++)

**printf**("%d ", stream[i]);

**printf**("\n");

}

// A function to randomly select k items from stream[0..n-1].

**void** selectKItems(**int** stream[], **int** n, **int** k)

{

**int** i; // index for elements in stream[]

// reservoir[] is the output array. Initialize it with

// first k elements from stream[]

**int** reservoir[k];

**for** (i = 0; i < k; i++)

reservoir[i] = stream[i];

// Use a different seed value so that we don't get

// same result each time we run this program

**srand**(**time**(NULL));

// Iterate from the (k+1)th element to nth element

**for** (; i < n; i++)

{

// Pick a random index from 0 to i.

**int** j = **rand**() % (i+1);

// If the randomly picked index is smaller than k, then replace

// the element present at the index with new element from stream

**if** (j < k)

reservoir[j] = stream[i];

}

**printf**("Following are k randomly selected items \n");

printArray(reservoir, k);

}

// Driver program to test above function.

**int** main()

{

**int** stream[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};

**int** n = **sizeof**(stream)/**sizeof**(stream[0]);

**int** k = 5;

selectKItems(stream, n, k);

**return** 0;

}

Time Complexity: O(n)

**How does this work?**

To prove that this solution works perfectly, we must prove that the probability that any

item *stream[i]* where *0 <= i < n* will be in final *reservoir[]* is *k/n*. Let us divide the proof in

two cases as first *k* items are treated differently.

**Case 1: For last *n-k* stream items, i.e., for *stream[i]* where *k <= i < n***

For every such stream item *stream[i]*, we pick a random index from 0 to *i* and if the

picked index is one of the first *k* indexes, we replace the element at picked index with

*stream[i]*

To simplify the proof, let us first consider the *last item*. The probability that the last item

is in final reservoir = The probability that one of the first *k* indexes is picked for last item

= *k/n* (the probability of picking one of the *k* items from a list of size *n*)

Let us now consider the *second last item*. The probability that the second last item is in

final *reservoir[]* = [Probability that one of the first *k* indexes is picked in iteration for

*stream[n-2]*] X [Probability that the index picked in iteration for *stream[n-1]* is not same

as index picked for *stream[n-2]* ] = [*k/(n-1)]\*[(n-1)/n*] = *k/n*.

Similarly, we can consider other items for all stream items from *stream[n-1]* to *stream[k]*

and generalize the proof.

**Case 2: For first *k* stream items, i.e., for *stream[i]* where *0 <= i < k***

The first *k* items are initially copied to *reservoir[]* and may be removed later in iterations

for *stream[k]* to *stream[n]*.

The probability that an item from *stream[0..k-1]* is in final array = Probability that the

item is not picked when items *stream[k], stream[k+1], …. stream[n-1]* are considered =

*[k/(k+1)] x [(k+1)/(k+2)] x [(k+2)/(k+3)] x … x [(n-1)/n] = k/n*

References:

http://en.wikipedia.org/wiki/Reservoir\_sampling

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above.

28. Pascal’s Triangle

Pascal’s triangle is a triangular array of the binomial coefficients. Write a function that

takes an integer value n as input and prints first n lines of the Pascal’s triangle. Following

are the first 6 rows of Pascal’s Triangle.

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

**Method 1 ( O(n^3) time complexity )**

Number of entries in every line is equal to line number. For example, the first line has “1”,

the second line has “1 1″, the third line has “1 2 1″,.. and so on. Every entry in a line is

value of a Binomial Coefficient. The value of ***i***th entry in line number *line* is *C(line, i)*. The

value can be calculated using following formula.

C(line, i) = line! / ( (line-i)! \* i! )

A simple method is to run two loops and calculate the value of Binomial Coefficient in

inner loop.

Time complexity of this method is O(n^3). Following are optimized methods.

**Method 2( O(n^2) time and O(n^2) extra space )**

If we take a closer at the triangle, we observe that every entry is sum of the two values

above it. So we can create a 2D array that stores previously generated values. To

generate a value in a line, we can use the previously stored values from array.

// A simple O(n^3) program for Pascal's Triangle

#include <stdio.h>

// See http://www.geeksforgeeks.org/archives/25621 for details of this function

**int** binomialCoeff(**int** n, **int** k);

// Function to print first n lines of Pascal's Triangle

**void** printPascal(**int** n)

{

// Iterate through every line and print entries in it

**for** (**int** line = 0; line < n; line++)

{

// Every line has number of integers equal to line number

**for** (**int** i = 0; i <= line; i++)

**printf**("%d ", binomialCoeff(line, i));

**printf**("\n");

}

}

// See http://www.geeksforgeeks.org/archives/25621 for details of this function

**int** binomialCoeff(**int** n, **int** k)

{

**int** res = 1;

**if** (k > n - k)

k = n - k;

**for** (**int** i = 0; i < k; ++i)

{

res \*= (n - i);

res /= (i + 1);

}

**return** res;

}

// Driver program to test above function

**int** main()

{

**int** n = 7;

printPascal(n);

**return** 0;

}

This method can be optimized to use O(n) extra space as we need values only from

previous row. So we can create an auxiliary array of size n and overwrite values.

Following is another method uses only O(1) extra space.

**Method 3 ( O(n^2) time and O(1) extra space )**

This method is based on method 1. We know that ***i***th entry in a line number *line* is

Binomial Coefficient *C(line, i)* and all lines start with value 1. The idea is to calculate

*C(line, i)* using *C(line, i-1)*. It can be calculated in O(1) time using the following.

C(line, i) = line! / ( (line-i)! \* i! )

C(line, i-1) = line! / ( (line - i + 1)! \* (i-1)! )

We can derive following expression from above two expressions.

C(line, i) = C(line, i-1) \* (line - i + 1) / i

So C(line, i) can be calculated from C(line, i-1) in O(1) time

So method 3 is the best method among all, but it may cause integer overflow for large

// A O(n^2) time and O(n^2) extra space method for Pascal's Triangle

**void** printPascal(**int** n)

{

**int** arr[n][n]; // An auxiliary array to store generated pscal triangle values

// Iterate through every line and print integer(s) in it

**for** (**int** line = 0; line < n; line++)

{

// Every line has number of integers equal to line number

**for** (**int** i = 0; i <= line; i++)

{

// First and last values in every row are 1

**if** (line == i || i == 0)

arr[line][i] = 1;

**else** // Other values are sum of values just above and left of above

arr[line][i] = arr[line-1][i-1] + arr[line-1][i];

**printf**("%d ", arr[line][i]);

}

**printf**("\n");

}

}

// A O(n^2) time and O(1) extra space function for Pascal's Triangle

**void** printPascal(**int** n)

{

**for** (**int** line = 1; line <= n; line++)

{

**int** C = 1; // used to represent C(line, i)

**for** (**int** i = 1; i <= line; i++)

{

**printf**("%d ", C); // The first value in a line is always 1

C = C \* (line - i) / i;

}

**printf**("\n");

}

}

values of n as it multiplies two integers to obtain values.

This article is compiled by **Rahul** and reviewed by GeeksforGeeks team. Please write

comments if you find anything incorrect, or you want to share more information about

the topic discussed above.

29. Select a random number from stream, with O(1) space

Given a stream of numbers, generate a random number from the stream. You are

allowed to use only O(1) space and the input is in the form of stream, so can’t store the

previously seen numbers.

So how do we generate a random number from the whole stream such that the

probability of picking any number is 1/n. with O(1) extra space? This problem is a

variation of Reservoir Sampling. Here the value of k is 1.

**1)** Initialize ‘count’ as 0, ‘count’ is used to store count of numbers seen so far in stream.

**2)** For each number ‘x’ from stream, do following

…..**a)** Increment ‘count’ by 1.

…..**b)** If count is 1, set result as x, and return result.

…..**c)** Generate a random number from 0 to ‘count-1′. Let the generated random number

be i.

…..**d)** If i is equal to ‘count – 1′, update the result as x.

Output:

Random number from first 1 numbers is 1

Random number from first 2 numbers is 1

Random number from first 3 numbers is 3

Random number from first 4 numbers is 4

Auxiliary Space: O(1)

**How does this work**

We need to prove that every element is picked with 1/n probability where n is the number

of items seen so far. For every new stream item x, we pick a random number from 0 to

‘count -1′, if the picked number is ‘count-1′, we replace the previous result with x.

To simplify proof, let us first consider the last element, the last element replaces the

// An efficient program to randomly select a number from stream of numbers.

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// A function to randomly select a item from stream[0], stream[1], .. stream[i-1]

**int** selectRandom(**int** x)

{

**static int** res; // The resultant random number

**static int** count = 0; //Count of numbers visited so far in stream

count++; // increment count of numbers seen so far

// If this is the first element from stream, return it

**if** (count == 1)

res = x;

**else**

{

// Generate a random number from 0 to count - 1

**int** i = **rand**() % count;

// Replace the prev random number with new number with 1/count probability

**if** (i == count - 1)

res = x;

}

**return** res;

}

// Driver program to test above function.

**int** main()

{

**int** stream[] = {1, 2, 3, 4};

**int** n = **sizeof**(stream)/**sizeof**(stream[0]);

// Use a different seed value for every run.

**srand**(**time**(NULL));

**for** (**int** i = 0; i < n; ++i)

**printf**("Random number from first %d numbers is %d \n",

i+1, selectRandom(stream[i]));

**return** 0;

}

previously stored result with 1/n probability. So probability of getting last element as

result is 1/n.

Let us now talk about second last element. When second last element processed first

time, the probability that it replaced the previous result is 1/(n-1). The probability that

previous result stays when nth item is considered is (n-1)/n. So probability that the

second last element is picked in last iteration is [1/(n-1)] \* [(n-1)/n] which is 1/n.

Similarly, we can prove for third last element and others.

References:

Reservoir Sampling

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above.

30. Find the largest multiple of 2, 3 and 5

An array of size n is given. The array contains digits from 0 to 9. Generate the largest

number using the digits in the array such that the number is divisible by 2, 3 and 5.

For example, if the arrays is {1, 8, 7, 6, 0}, output must be: 8760. And if the arrays is {7,

7, 7, 6}, output must be: “no number can be formed”.

Source: Amazon Interview | Set 7

This problem is a variation of “Find the largest multiple of 3“.

Since the number has to be divisible by 2 and 5, it has to have last digit as 0. So if the

given array doesn’t contain any zero, then no solution exists.

Once a 0 is available, extract 0 from the given array. Only thing left is, the number should

be is divisible by 3 and the largest of all. Which has been discussed here.

Thanks to shashank for suggesting this solution. Please write comments if you find

anything incorrect, or you want to share more information about the topic discussed

above.

31. Efficient program to calculate e^x

The value of Exponential Function e^x can be expressed using following Taylor Series.

e^x = 1 + x/1! + x^2/2! + x^3/3! + ......

*How to efficiently calculate the sum of above series?*

The series can be re-written as

e^x = 1 + (x/1) (1 + (x/2) (1 + (x/3) (........) ) )

Let the sum needs to be calculated for n terms, we can calculate sum using following

loop.

for (i = n - 1, sum = 1; i > 0; --i )

sum = 1 + x \* sum / i;

Following is implementation of the above idea.

Output:

e^x = 2.718282

This article is compiled by **Rahul** and reviewed by GeeksforGeeks team. Please write

comments if you find anything incorrect, or you want to share more information about

the topic discussed above

32. Measure one litre using two vessels and infinite water supply

// Efficient program to calculate e raise to the power x

#include <stdio.h>

//Returns approximate value of e^x using sum of first n terms of Taylor Series

**float** exponential(**int** n, **float** x)

{

**float** sum = 1.0f; // initialize sum of series

**for** (**int** i = n - 1; i > 0; --i )

sum = 1 + x \* sum / i;

**return** sum;

}

// Driver program to test above function

**int** main()

{

**int** n = 10;

**float** x = 1.0f;

**printf**("e^x = %f", exponential(n, x));

**return** 0;

}

There are two vessels of capacities ‘a’ and ‘b’ respectively. We have infinite water

supply. Give an efficient algorithm to make exactly 1 litre of water in one of the vessels.

You can throw all the water from any vessel any point of time. Assume that ‘a’ and ‘b’

are Coprimes.

Following are the steps:

Let V1 be the vessel of capacity ‘a’ and V2 be the vessel of capacity ‘b’ and ‘a’ is

smaller than ‘b’.

**1)** Do following while the amount of water in V1 is not 1.

….**a)** If V1 is empty, then completely fill V1

….**b)** Transfer water from V1 to V2. If V2 becomes full, then keep the remaining water in

V1 and empty V2

**2)** V1 will have 1 litre after termination of loop in step 1. Return.

Following is C++ implementation of the above algorithm.

/\* Sample run of the Algo for V1 with capacity 3 and V2 with capacity 7

1. Fill V1: V1 = 3, V2 = 0

2. Transfer from V1 to V2, and fill V1: V1 = 3, V2 = 3

2. Transfer from V1 to V2, and fill V1: V1 = 3, V2 = 6

3. Transfer from V1 to V2, and empty V2: V1 = 2, V2 = 0

4. Transfer from V1 to V2, and fill V1: V1 = 3, V2 = 2

5. Transfer from V1 to V2, and fill V1: V1 = 3, V2 = 5

6. Transfer from V1 to V2, and empty V2: V1 = 1, V2 = 0

7. Stop as V1 now contains 1 litre.

Note that V2 was made empty in steps 3 and 6 because it became full \*/

#include <iostream>

**using namespace** std;

// A utility function to get GCD of two numbers

**int** gcd(**int** a, **int** b) { **return** b? gcd(b, a % b) : a; }

// Class to represent a Vessel

**class** Vessel

{

// A vessel has capacity, and current amount of water in it

**int** capacity, current;

**public**:

// Constructor: initializes capacity as given, and current as 0

Vessel(**int** capacity) { **this**->capacity = capacity; current = 0; }

// The main function to fill one litre in this vessel. Capacity of V2

// must be greater than this vessel and two capacities must be co-prime

**void** makeOneLitre(Vessel &V2);

// Fills vessel with given amount and returns the amount of water

// transferred to it. If the vessel becomes full, then the vessel

// is made empty.

**int** transfer(**int** amount);

};

// The main function to fill one litre in this vessel. Capacity

// of V2 must be greater than this vessel and two capacities

// must be coprime

**void** Vessel:: makeOneLitre(Vessel &V2)

**void** Vessel:: makeOneLitre(Vessel &V2)

{

// solution exists iff a and b are co-prime

**if** (gcd(capacity, V2.capacity) != 1)

**return**;

**while** (current != 1)

{

// fill A (smaller vessel)

**if** (current == 0)

current = capacity;

cout << "Vessel 1: " << current << " Vessel 2: "

<< V2.current << endl;

// Transfer water from V1 to V2 and reduce current of V1 by

// the amount equal to transferred water

current = current - V2.transfer(current);

}

// Finally, there will be 1 litre in vessel 1

cout << "Vessel 1: " << current << " Vessel 2: "

<< V2.current << endl;

}

// Fills vessel with given amount and returns the amount of water

// transferred to it. If the vessel becomes full, then the vessel

// is made empty

**int** Vessel::transfer(**int** amount)

{

// If the vessel can accommodate the given amount

**if** (current + amount < capacity)

{

current += amount;

**return** amount;

}

// If the vessel cannot accommodate the given amount, then

// store the amount of water transferred

**int** transferred = capacity - current;

// Since the vessel becomes full, make the vessel

// empty so that it can be filled again

current = 0;

**return** transferred;

}

// Driver program to test above function

**int** main()

{

**int** a = 3, b = 7; // a must be smaller than b

// Create two vessels of capacities a and b

Vessel V1(a), V2(b);

// Get 1 litre in first vessel

V1.makeOneLitre(V2);

**return** 0;

}

Output:

Vessel 1: 3 Vessel 2: 0

Vessel 1: 3 Vessel 2: 3

Vessel 1: 3 Vessel 2: 6

Vessel 1: 2 Vessel 2: 0

Vessel 1: 3 Vessel 2: 2

Vessel 1: 3 Vessel 2: 5

Vessel 1: 1 Vessel 2: 0

**How does this work?**

To prove that the algorithm works, we need to proof that after certain number of

iterations in the while loop, we will get 1 litre in V1.

Let ‘a’ be the capacity of vessel V1 and ‘b’ be the capacity of V2. Since we repeatedly

transfer water from V1 to V2 until V2 becomes full, we will have ‘a – b (mod a)’ water in

V1 when V2 becomes full first time . Once V2 becomes full, it is emptied. We will have ‘a

– 2b (mod a)’ water in V1 when V2 is full second time. We repeat the above steps, and

get ‘a – nb (mod a)’ water in V1 after the vessel V2 is filled and emptied ‘n’ times. We

need to prove that the value of ‘a – nb (mod a)’ will be 1 for a finite integer ‘n’. To prove

this, let us consider the following property of coprime numbers.

For any two coprime integers ‘a’ and ‘b’, the integer ‘b’ has a multiplicative inverse

modulo ‘a’. In other words, there exists an integer ‘y’ such that (See 3rd point

here). After ‘(a – 1)\*y’ iterations, we will have ‘a – [(a-1)\*y\*b (mod a)]’ water in V1, the

value of this expression is ‘a – [(a – 1) \* 1] mod a’ which is 1. So the algorithm

converges and we get 1 litre in V1.

This article is compiled by Aashish Barnwal. Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above.

33. Efficient program to print all prime factors of a given number

Given a number n, write an efficient function to print all prime factors of n. For example,

if the input number is 12, then output should be “2 2 3″. And if the input number is 315,

then output should be “3 3 5 7″.

Following are the steps to find all prime factors.

**1)** While n is divisible by 2, print 2 and divide n by 2.

**2)** After step 1, n must be odd. Now start a loop from i = 3 to square root of n. While i

divides n, print i and divide n by i, increment i by 2 and continue.

**3)** If n is a prime number and is greater than 2, then n will not become 1 by above two

steps. So print n if it is greater than 2.

Output:

3 3 5 7

**How does this work?**

The steps 1 and 2 take care of composite numbers and step 3 takes care of prime

numbers. To prove that the complete algorithm works, we need to prove that steps 1

and 2 actually take care of composite numbers. This is clear that step 1 takes care of

even numbers. And after step 1, all remaining prime factor must be odd (difference of

two prime factors must be at least 2), this explains why i is incremented by 2.

Now the main part is, the loop runs till square root of n not till. To prove that this

optimization works, let us consider the following property of composite numbers.

*Every composite number has at least one prime factor less than or equal to square root*

*of itself.*

This property can be proved using counter statement. Let a and b be two factors of n

such that a\*b = n. If both are greater than , then a.b > which contradicts the

expression “a \* b = n”.

// Program to print all prime factors

# include <stdio.h>

# include <math.h>

// A function to print all prime factors of a given number n

**void** primeFactors(**int** n)

{

// Print the number of 2s that divide n

**while** (n%2 == 0)

{

**printf**("%d ", 2);

n = n/2;

}

// n must be odd at this point. So we can skip one element (Note i = i +2)

**for** (**int** i = 3; i <= **sqrt**(n); i = i+2)

{

// While i divides n, print i and divide n

**while** (n%i == 0)

{

**printf**("%d ", i);

n = n/i;

}

}

// This condition is to handle the case whien n is a prime number

// greater than 2

**if** (n > 2)

**printf** ("%d ", n);

}

/\* Driver program to test above function \*/

**int** main()

{

**int** n = 315;

primeFactors(n);

**return** 0;

}

In step 2 of the above algorithm, we run a loop and do following in loop

a) Find the least prime factor i (must be less than )

b) Remove all occurrences i from n by repeatedly dividing n by i.

c) Repeat steps a and b for divided n and i = i + 2. The steps a and b are repeated till n

becomes either 1 or a prime number.

Thanks to **Vishwas Garg** for suggesting the above algorithm. Please write comments if

you find anything incorrect, or you want to share more information about the topic

discussed above

34. Print all possible combinations of r elements in a given array of

size n

Given an array of size n, generate and print all possible combinations of r elements in

array. For example, if input array is {1, 2, 3, 4} and r is 2, then output should be {1, 2}, {1,

3}, {1, 4}, {2, 3}, {2, 4} and {3, 4}.

Following are two methods to do this.

**Method 1 (Fix Elements and Recur)**

We create a temporary array ‘data[]’ which stores all outputs one by one. The idea is to

start from first index (index = 0) in data[], one by one fix elements at this index and recur

for remaining indexes. Let the input array be {1, 2, 3, 4, 5} and r be 3. We first fix 1 at

index 0 in data[], then recur for remaining indexes, then we fix 2 at index 0 and recur.

Finally, we fix 3 and recur for remaining indexes. When number of elements in data[]

becomes equal to r (size of a combination), we print data[].

Following diagram shows recursion tree for same input.

Following is C++ implementation of above approach.

Output:

1 2 3

1 2 4

1 2 5

1 3 4

1 3 5

1 4 5

// Program to print all combination of size r in an array of size n

#include <stdio.h>

**void** combinationUtil(**int** arr[], **int** data[], **int** start, **int** end, **int** index,

// The main function that prints all combinations of size r

// in arr[] of size n. This function mainly uses combinationUtil()

**void** printCombination(**int** arr[], **int** n, **int** r)

{

// A temporary array to store all combination one by one

**int** data[r];

// Print all combination using temprary array 'data[]'

combinationUtil(arr, data, 0, n-1, 0, r);

}

/\* arr[] ---> Input Array

data[] ---> Temporary array to store current combination

start & end ---> Staring and Ending indexes in arr[]

index ---> Current index in data[]

r ---> Size of a combination to be printed \*/

**void** combinationUtil(**int** arr[], **int** data[], **int** start, **int** end, **int** index,

{

// Current combination is ready to be printed, print it

**if** (index == r)

{

**for** (**int** j=0; j<r; j++)

**printf**("%d ", data[j]);

**printf**("\n");

**return**;

}

// replace index with all possible elements. The condition

// "end-i+1 >= r-index" makes sure that including one element

// at index will make a combination with remaining elements

// at remaining positions

**for** (**int** i=start; i<=end && end-i+1 >= r-index; i++)

{

data[index] = arr[i];

combinationUtil(arr, data, i+1, end, index+1, r);

}

}

// Driver program to test above functions

**int** main()

{

**int** arr[] = {1, 2, 3, 4, 5};

**int** r = 3;

**int** n = **sizeof**(arr)/**sizeof**(arr[0]);

printCombination(arr, n, r);

}

2 3 4

2 3 5

2 4 5

3 4 5

*How to handle duplicates?*

Note that the above method doesn’t handle duplicates. For example, if input array is {1,

2, 1} and r is 2, then the program prints {1, 2} and {2, 1} as two different combinations.

We can avoid duplicates by adding following two additional things to above code.

1) Add code to sort the array before calling combinationUtil() in printCombination()

2) Add following lines at the end of for loop in combinationUtil()

// Since the elements are sorted, all occurrences of an element

// must be together

while (arr[i] == arr[i+1])

i++;

See **this** for an implementation that handles duplicates.

**Method 2 (Include and Exclude every element)**

Like the above method, We create a temporary array data[]. The idea here is similar to

Subset Sum Problem. We one by one consider every element of input array, and recur

for two cases:

1) The element is included in current combination (We put the element in data[] and

increment next available index in data[])

2) The element is excluded in current combination (We do not put the element and do

not change index)

When number of elements in data[] become equal to r (size of a combination), we print

it.

This method is mainly based on Pascal’s Identity, i.e. **ncr = n-1cr + n-1cr-1**

Following is C++ implementation of method 2.

Output:

1 2 3

1 2 4

1 2 5

// Program to print all combination of size r in an array of size n

#include<stdio.h>

**void** combinationUtil(**int** arr[],**int** n,**int** r,**int** index,**int** data[],**int** i);

// The main function that prints all combinations of size r

// in arr[] of size n. This function mainly uses combinationUtil()

**void** printCombination(**int** arr[], **int** n, **int** r)

{

// A temporary array to store all combination one by one

**int** data[r];

// Print all combination using temprary array 'data[]'

combinationUtil(arr, n, r, 0, data, 0);

}

/\* arr[] ---> Input Array

n ---> Size of input array

r ---> Size of a combination to be printed

index ---> Current index in data[]

data[] ---> Temporary array to store current combination

i ---> index of current element in arr[] \*/

**void** combinationUtil(**int** arr[], **int** n, **int** r, **int** index, **int** data[], **int**

{

// Current cobination is ready, print it

**if** (index == r)

{

**for** (**int** j=0; j<r; j++)

**printf**("%d ",data[j]);

**printf**("\n");

**return**;

}

// When no more elements are there to put in data[]

**if** (i >= n)

**return**;

// current is included, put next at next location

data[index] = arr[i];

combinationUtil(arr, n, r, index+1, data, i+1);

// current is excluded, replace it with next (Note that

// i+1 is passed, but index is not changed)

combinationUtil(arr, n, r, index, data, i+1);

}

// Driver program to test above functions

**int** main()

{

**int** arr[] = {1, 2, 3, 4, 5};

**int** r = 3;

**int** n = **sizeof**(arr)/**sizeof**(arr[0]);

printCombination(arr, n, r);

**return** 0;

}

1 3 4

1 3 5

1 4 5

2 3 4

2 3 5

2 4 5

3 4 5

*How to handle duplicates in method 2?*

Like method 1, we can following two things to handle duplicates.

1) Add code to sort the array before calling combinationUtil() in printCombination()

2) Add following lines between two recursive calls of combinationUtil() in

combinationUtil()

// Since the elements are sorted, all occurrences of an element

// must be together

while (arr[i] == arr[i+1])

i++;

See **this** for an implementation that handles duplicates.

This article is contributed by **Bateesh**. Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above

35. Random number generator in arbitrary probability distribution

fashion

Given n numbers, each with some frequency of occurrence. Return a random number

with probability proportional to its frequency of occurrence.

Example:

Let following be the given numbers.

arr[] = {10, 30, 20, 40}

Let following be the frequencies of given numbers.

freq[] = {1, 6, 2, 1}

The output should be

10 with probability 1/10

30 with probability 6/10

20 with probability 2/10

40 with probability 1/10

It is quite clear that the simple random number generator won’t work here as it doesn’t

keep track of the frequency of occurrence.

We need to somehow transform the problem into a problem whose solution is known to

us.

One simple method is to take an auxiliary array (say aux[]) and duplicate the numbers

according to their frequency of occurrence. Generate a random number(say r) between

0 to Sum-1(including both), where Sum represents summation of frequency array (freq[]

in above example). Return the random number aux[r] (Implementation of this method is

left as an exercise to the readers).

The limitation of the above method discussed above is huge memory consumption

when frequency of occurrence is high. If the input is 997, 8761 and 1, this method is

clearly not efficient.

How can we reduce the memory consumption? Following is detailed algorithm that uses

O(n) extra space where n is number of elements in input arrays.

**1.** Take an auxiliary array (say prefix[]) of size n.

**2.** Populate it with prefix sum, such that prefix[i] represents sum of numbers from 0 to i.

**3.** Generate a random number(say r) between 1 to Sum(including both), where Sum

represents summation of input frequency array.

**4.** Find index of Ceil of random number generated in step #3 in the prefix array. Let the

index be index**c**.

**5.** Return the random number arr[indexc], where arr[] contains the input n numbers.

Before we go to the implementation part, let us have quick look at the algorithm with an

example:

arr[]: {10, 20, 30}

freq[]: {2, 3, 1}

Prefix[]: {2, 5, 6}

Since last entry in prefix is 6, all possible values of r are [1, 2, 3, 4, 5, 6]

1: Ceil is 2. Random number generated is 10.

2: Ceil is 2. Random number generated is 10.

3: Ceil is 5. Random number generated is 20.

4: Ceil is 5. Random number generated is 20.

5: Ceil is 5. Random number generated is 20.

6. Ceil is 6. Random number generated is 30.

In the above example

10 is generated with probability 2/6.

20 is generated with probability 3/6.

30 is generated with probability 1/6.

**How does this work?**

Any number input[i] is generated as many times as its frequency of occurrence because

there exists count of integers in range(prefix[i – 1], prefix[i]] is input[i]. Like in the above

example 3 is generated thrice, as there exists 3 integers 3, 4 and 5 whose ceil is 5.

Output: May be different for different runs

4

3

4

//C program to generate random numbers according to given frequency distribution

#include <stdio.h>

#include <stdlib.h>

// Utility function to find ceiling of r in arr[l..h]

**int** findCeil(**int** arr[], **int** r, **int** l, **int** h)

{

**int** mid;

**while** (l < h)

{

mid = l + ((h - l) >> 1); // Same as mid = (l+h)/2

(r > arr[mid]) ? (l = mid + 1) : (h = mid);

}

**return** (arr[l] >= r) ? l : -1;

}

// The main function that returns a random number from arr[] according to

// distribution array defined by freq[]. n is size of arrays.

**int** myRand(**int** arr[], **int** freq[], **int** n)

{

// Create and fill prefix array

**int** prefix[n], i;

prefix[0] = freq[0];

**for** (i = 1; i < n; ++i)

prefix[i] = prefix[i - 1] + freq[i];

// prefix[n-1] is sum of all frequencies. Generate a random number

// with value from 1 to this sum

**int** r = (**rand**() % prefix[n - 1]) + 1;

// Find index of ceiling of r in prefix arrat

**int** indexc = findCeil(prefix, r, 0, n - 1);

**return** arr[indexc];

}

// Driver program to test above functions

**int** main()

{

**int** arr[] = {1, 2, 3, 4};

**int** freq[] = {10, 5, 20, 100};

**int** i, n = **sizeof**(arr) / **sizeof**(arr[0]);

// Use a different seed value for every run.

**srand**(**time**(NULL));

// Let us generate 10 random numbers accroding to

// given distribution

**for** (i = 0; i < 5; i++)

**printf**("%d\n", myRand(arr, freq, n));

**return** 0;

}

4

4

This article is compiled by Aashish Barnwal. Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above.

36. How to check if two given line segments intersect?

Given two line segments (p1, q1) and (p2, q2), find if the given line segments intersect

with each other.

Before we discuss solution, let us define notion of **orientation**. Orientation of an ordered triplet of points in

the plane can be

–counterclockwise

–clockwise

–colinear

The following diagram shows different possible orientations of (a, b, c)

Note the word ‘ordered’ here. Orientation of (a, b, c) may be different from orientation of

(c, b, a).

**How is Orientation useful here?**

Two segments (p1,q1) and (p2,q2) intersect if and only if one of the following two

conditions is verified

**1. *General Case:***

– (p1, q1, p2) and (p1, q1, q2) have different orientations and

– (p2, q2, p1) and (p2, q2, q1) have different orientations

**2. *Special Case***

– (p1, q1, p2), (p1, q1, q2), (p2, q2, p1), and (p2, q2, q1) are all collinear and

– the x-projections of (p1, q1) and (p2, q2) intersect

– the y-projections of (p1, q1) and (p2, q2) intersect

***Examples of General Case:***

***Examples of Special Case:***

Following is C++ implementation based on above idea.

// A C++ program to check if two given line segments intersect

#include <iostream>

**using namespace** std;

**struct** Point

{

**int** x;

**int** y;

};

// Given three colinear points p, q, r, the function checks if

// point q lies on line segment 'pr'

**bool** onSegment(Point p, Point q, Point r)

{

**if** (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&

q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))

**return true**;

**return false**;

}

// To find orientation of ordered triplet (p, q, r).

// The function returns following values

// 0 --> p, q and r are colinear

// 1 --> Clockwise

// 2 --> Counterclockwise

**int** orientation(Point p, Point q, Point r)

{

// See 10th slides from following link for derivation of the formula

// http://www.dcs.gla.ac.uk/~pat/52233/slides/Geometry1x1.pdf

**int** val = (q.y - p.y) \* (r.x - q.x) -

(q.x - p.x) \* (r.y - q.y);

**if** (val == 0) **return** 0; // colinear

Output:

No

Yes

No

**return** (val > 0)? 1: 2; // clock or counterclock wise

}

// The main function that returns true if line segment 'p1q1'

// and 'p2q2' intersect.

**bool** doIntersect(Point p1, Point q1, Point p2, Point q2)

{

// Find the four orientations needed for general and

// special cases

**int** o1 = orientation(p1, q1, p2);

**int** o2 = orientation(p1, q1, q2);

**int** o3 = orientation(p2, q2, p1);

**int** o4 = orientation(p2, q2, q1);

// General case

**if** (o1 != o2 && o3 != o4)

**return true**;

// Special Cases

// p1, q1 and p2 are colinear and p2 lies on segment p1q1

**if** (o1 == 0 && onSegment(p1, p2, q1)) **return true**;

// p1, q1 and p2 are colinear and q2 lies on segment p1q1

**if** (o2 == 0 && onSegment(p1, q2, q1)) **return true**;

// p2, q2 and p1 are colinear and p1 lies on segment p2q2

**if** (o3 == 0 && onSegment(p2, p1, q2)) **return true**;

// p2, q2 and q1 are colinear and q1 lies on segment p2q2

**if** (o4 == 0 && onSegment(p2, q1, q2)) **return true**;

**return false**; // Doesn't fall in any of the above cases

}

// Driver program to test above functions

**int** main()

{

**struct** Point p1 = {1, 1}, q1 = {10, 1};

**struct** Point p2 = {1, 2}, q2 = {10, 2};

doIntersect(p1, q1, p2, q2)? cout << "Yes\n": cout << "No\n";

p1 = {10, 0}, q1 = {0, 10};

p2 = {0, 0}, q2 = {10, 10};

doIntersect(p1, q1, p2, q2)? cout << "Yes\n": cout << "No\n";

p1 = {-5, -5}, q1 = {0, 0};

p2 = {1, 1}, q2 = {10, 10};

doIntersect(p1, q1, p2, q2)? cout << "Yes\n": cout << "No\n";

**return** 0;

}

**Sources:**

http://www.dcs.gla.ac.uk/~pat/52233/slides/Geometry1x1.pdf

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E.

Leiserson, Ronald L. Rivest

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above

37. How to check if a given point lies inside or outside a polygon?

Given a polygon and a point ‘p’, find if ‘p’ lies inside the polygon or not. The points lying

on the border are considered inside.

We strongly recommend to see the following post first.

How to check if two given line segments intersect?

Following is a simple idea to check whether a point is inside or outside.

**1)** Draw a horizontal line to the right of each point and extend it to infinity

**1)** Count the number of times the line intersects with polygon edges.

**2)** A point is inside the polygon if either count of intersections is odd or

point lies on an edge of polygon. If none of the conditions is true, then

point lies outside.

**How to handle point ‘g’ in the above figure?**

Note that we should returns true if the point lies on the line or same as one of the

vertices of the given polygon. To handle this, after checking if the line from ‘p’ to

extreme intersects, we check whether ‘p’ is colinear with vertices of current line of

polygon. If it is coliear, then we check if the point ‘p’ lies on current side of polygon, if it

lies, we return true, else false.

Following is C++ implementation of the above idea.

// A C++ program to check if a given point lies inside a given polygon

// Refer http://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/

// for explanation of functions onSegment(), orientation() and doIntersect()

#include <iostream>

**using namespace** std;

// Define Infinite (Using INT\_MAX caused overflow problems)

#define INF 10000

**struct** Point

{

**int** x;

**int** y;

};

// Given three colinear points p, q, r, the function checks if

// point q lies on line segment 'pr'

**bool** onSegment(Point p, Point q, Point r)

{

**if** (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&

q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))

**return true**;

**return false**;

}

// To find orientation of ordered triplet (p, q, r).

// The function returns following values

// 0 --> p, q and r are colinear

// 1 --> Clockwise

// 2 --> Counterclockwise

**int** orientation(Point p, Point q, Point r)

{

**int** val = (q.y - p.y) \* (r.x - q.x) -

(q.x - p.x) \* (r.y - q.y);

**if** (val == 0) **return** 0; // colinear

**return** (val > 0)? 1: 2; // clock or counterclock wise

}

// The function that returns true if line segment 'p1q1'

// and 'p2q2' intersect.

**bool** doIntersect(Point p1, Point q1, Point p2, Point q2)

{

// Find the four orientations needed for general and

// special cases

**int** o1 = orientation(p1, q1, p2);

**int** o2 = orientation(p1, q1, q2);

**int** o3 = orientation(p2, q2, p1);

**int** o4 = orientation(p2, q2, q1);

// General case

**if** (o1 != o2 && o3 != o4)

**return true**;

// Special Cases

// Special Cases

// p1, q1 and p2 are colinear and p2 lies on segment p1q1

**if** (o1 == 0 && onSegment(p1, p2, q1)) **return true**;

// p1, q1 and p2 are colinear and q2 lies on segment p1q1

**if** (o2 == 0 && onSegment(p1, q2, q1)) **return true**;

// p2, q2 and p1 are colinear and p1 lies on segment p2q2

**if** (o3 == 0 && onSegment(p2, p1, q2)) **return true**;

// p2, q2 and q1 are colinear and q1 lies on segment p2q2

**if** (o4 == 0 && onSegment(p2, q1, q2)) **return true**;

**return false**; // Doesn't fall in any of the above cases

}

// Returns true if the point p lies inside the polygon[] with n vertices

**bool** isInside(Point polygon[], **int** n, Point p)

{

// There must be at least 3 vertices in polygon[]

**if** (n < 3) **return false**;

// Create a point for line segment from p to infinite

Point extreme = {INF, p.y};

// Count intersections of the above line with sides of polygon

**int** count = 0, i = 0;

**do**

{

**int** next = (i+1)%n;

// Check if the line segment from 'p' to 'extreme' intersects

// with the line segment from 'polygon[i]' to 'polygon[next]'

**if** (doIntersect(polygon[i], polygon[next], p, extreme))

{

// If the point 'p' is colinear with line segment 'i-next',

// then check if it lies on segment. If it lies, return true,

// otherwise false

**if** (orientation(polygon[i], p, polygon[next]) == 0)

**return** onSegment(polygon[i], p, polygon[next]);

count++;

}

i = next;

} **while** (i != 0);

// Return true if count is odd, false otherwise

**return** count&1; // Same as (count%2 == 1)

}

// Driver program to test above functions

**int** main()

{

Point polygon1[] = {{0, 0}, {10, 0}, {10, 10}, {0, 10}};

**int** n = **sizeof**(polygon1)/**sizeof**(polygon1[0]);

Point p = {20, 20};

isInside(polygon1, n, p)? cout << "Yes \n": cout << "No \n";

p = {5, 5};

isInside(polygon1, n, p)? cout << "Yes \n": cout << "No \n";

Point polygon2[] = {{0, 0}, {5, 5}, {5, 0}};

p = {3, 3};

n = **sizeof**(polygon2)/**sizeof**(polygon2[0]);

Output:

No

Yes

Yes

Yes

No

No

**Time Complexity:** O(n) where n is the number of vertices in the given polygon.

**Source:**

http://www.dcs.gla.ac.uk/~pat/52233/slides/Geometry1x1.pdf

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above

38. Convex Hull | Set 1 (Jarvis’s Algorithm or Wrapping)

Given a set of points in the plane. the convex hull of the set is the smallest convex

polygon that contains all the points of it.

We strongly recommend to see the following post first.

How to check if two given line segments intersect?

The idea of Jarvis’s Algorithm is simple, we start from the leftmost point (or point with

n = **sizeof**(polygon2)/**sizeof**(polygon2[0]);

isInside(polygon2, n, p)? cout << "Yes \n": cout << "No \n";

p = {5, 1};

isInside(polygon2, n, p)? cout << "Yes \n": cout << "No \n";

p = {8, 1};

isInside(polygon2, n, p)? cout << "Yes \n": cout << "No \n";

Point polygon3[] = {{0, 0}, {10, 0}, {10, 10}, {0, 10}};

p = {-1,10};

n = **sizeof**(polygon3)/**sizeof**(polygon3[0]);

isInside(polygon3, n, p)? cout << "Yes \n": cout << "No \n";

**return** 0;

}

minimum x coordinate value) and we keep wrapping points in counterclockwise direction.

The big question is, given a point p as current point, how to find the next point in output?

The idea is to use orientation() here. Next point is selected as the point that beats all

other points at counterclockwise orientation, i.e., next point is q if for any other point r,

we have “orientation(p, r, q) = counterclockwise”. Following is the detailed algorithm.

**1)** Initialize p as leftmost point.

**2)** Do following while we don’t come back to the first (or leftmost) point.

…..**a)** The next point q is the point such that the triplet (p, q, r) is counterclockwise for

any other point r.

…..**b)** next[p] = q (Store q as next of p in the output convex hull).

…..**c)** p = q (Set p as q for next iteration).

// A C++ program to find convex hull of a set of points

// Refer http://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/

// for explanation of orientation()

#include <iostream>

**using namespace** std;

// Define Infinite (Using INT\_MAX caused overflow problems)

#define INF 10000

**struct** Point

{

**int** x;

**int** y;

};

// To find orientation of ordered triplet (p, q, r).

// The function returns following values

// 0 --> p, q and r are colinear

// 1 --> Clockwise

// 2 --> Counterclockwise

**int** orientation(Point p, Point q, Point r)

{

**int** val = (q.y - p.y) \* (r.x - q.x) -

(q.x - p.x) \* (r.y - q.y);

**if** (val == 0) **return** 0; // colinear

**return** (val > 0)? 1: 2; // clock or counterclock wise

}

// Prints convex hull of a set of n points.

**void** convexHull(Point points[], **int** n)

{

// There must be at least 3 points

**if** (n < 3) **return**;

// Initialize Result

**int** next[n];

**for** (**int** i = 0; i < n; i++)

next[i] = -1;

// Find the leftmost point

**int** l = 0;

**for** (**int** i = 1; i < n; i++)

**if** (points[i].x < points[l].x)

l = i;

**Output:** The output is points of the convex hull.

(0, 3)

(3, 0)

(0, 0)

(3, 3)

**Time Complexity:** For every point on the hull we examine all the other points to

determine the next point. Time complexity is where n is number of input points

and m is number of output or hull points (m <= n). In worst case, time complexity is O(n

2). The worst case occurs when all the points are on the hull (m = n)

We will soon be discussing other algorithms for finding convex hulls.

**Sources:**

http://www.cs.uiuc.edu/~jeffe/teaching/373/notes/x05-convexhull.pdf

http://www.dcs.gla.ac.uk/~pat/52233/slides/Hull1x1.pdf

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above

// Start from leftmost point, keep moving counterclockwise

// until reach the start point again

**int** p = l, q;

**do**

{

// Search for a point 'q' such that orientation(p, i, q) is

// counterclockwise for all points 'i'

q = (p+1)%n;

**for** (**int** i = 0; i < n; i++)

**if** (orientation(points[p], points[i], points[q]) == 2)

q = i;

next[p] = q; // Add q to result as a next point of p

p = q; // Set p as q for next iteration

} **while** (p != l);

// Print Result

**for** (**int** i = 0; i < n; i++)

{

**if** (next[i] != -1)

cout << "(" << points[i].x << ", " << points[i].y << ")\n";

}

}

// Driver program to test above functions

**int** main()

{

Point points[] = {{0, 3}, {2, 2}, {1, 1}, {2, 1},

{3, 0}, {0, 0}, {3, 3}};

**int** n = **sizeof**(points)/**sizeof**(points[0]);

convexHull(points, n);

**return** 0;

}

39. Convex Hull | Set 2 (Graham Scan)

Given a set of points in the plane. the convex hull of the set is the smallest convex

polygon that contains all the points of it.

We strongly recommend to see the following post first.

How to check if two given line segments intersect?

We have discussed Jarvis’s Algorithm for Convex Hull. Worst case time complexity of

Jarvis’s Algorithm is O(n^2). Using Graham’s scan algorithm, we can find Convex Hull in

O(nLogn) time. Following is Graham’s algorithm

Let points[0..n-1] be the input array.

**1)** Find the bottom-most point by comparing y coordinate of all points. If there are two

points with same y value, then the point with smaller x coordinate value is considered.

Put the bottom-most point at first position.

**2)** Consider the remaining n-1 points and sort them by polor angle in counterclockwise

order around points[0]. If polor angle of two points is same, then put the nearest point

first.

**3)** Create an empty stack ‘S’ and push points[0], points[1] and points[2] to S.

**4)** Process remaining n-3 points one by one. Do following for every point ‘points[i]’

**4.1)** Keep removing points from stack while orientation of following 3 points is not

counterclockwise (or they don’t make a left turn).

a) Point next to top in stack

b) Point at the top of stack

c) points[i]

**4.2)** Push points[i] to S

**5)** Print contents of S

The above algorithm can be divided in two phases.

**Phase 1 (Sort points):** We first find the bottom-most point. The idea is to pre-process

points be sorting them with respect to the bottom-most point. Once the points are

sorted, they form a simple closed path (See following diagram).

What should be the sorting criteria? computation of actual angles would be inefficient

since trigonometric functions are not simple to evaluate. The idea is to use the

orientation to compare angles without actually computing them (See the compare()

function below)

**Phase 2 (Accept or Reject Points):** Once we have the closed path, the next step is to

traverse the path and remove concave points on this path. How to decide which point to

remove and which to keep? Again, orientation helps here. The first two points in sorted

array are always part of Convex Hull. For remaining points, we keep track of recent three

points, and find the angle formed by them. Let the three points be prev(p), curr(c) and

next(n). If orientation of these points (considering them in same order) is not

counterclockwise, we discard c, otherwise we keep it. Following diagram shows step by

step process of this phase (Source of these diagrams is Ref 2).

Following is C++ implementation of the above algorithm.

// A C++ program to find convex hull of a set of points

// Refer http://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/

// for explanation of orientation()

#include <iostream>

#include <stack>

#include <stdlib.h>

**using namespace** std;

**struct** Point

{

**int** x;

**int** y;

};

// A globle point needed for sorting points with reference to the first point

// Used in compare function of qsort()

Point p0;

// A utility function to find next to top in a stack

Point nextToTop(stack<Point> &S)

{

Point p = S.top();

S.pop();

Point res = S.top();

S.push(p);

**return** res;

}

// A utility function to swap two points

**int** swap(Point &p1, Point &p2)

{

Point temp = p1;

p1 = p2;

p2 = temp;

}

// A utility function to return square of distance between p1 and p2

**int** dist(Point p1, Point p2)

{

**return** (p1.x - p2.x)\*(p1.x - p2.x) + (p1.y - p2.y)\*(p1.y - p2.y);

}

// To find orientation of ordered triplet (p, q, r).

// The function returns following values

// 0 --> p, q and r are colinear

// 1 --> Clockwise

// 2 --> Counterclockwise

**int** orientation(Point p, Point q, Point r)

{

**int** val = (q.y - p.y) \* (r.x - q.x) -

(q.x - p.x) \* (r.y - q.y);

**if** (val == 0) **return** 0; // colinear

**return** (val > 0)? 1: 2; // clock or counterclock wise

}

// A function used by library function qsort() to sort an array of

// points with respect to the first point

**int** compare(**const void** \*vp1, **const void** \*vp2)

{

Point \*p1 = (Point \*)vp1;

Point \*p2 = (Point \*)vp2;

// Find orientation

**int** o = orientation(p0, \*p1, \*p2);

**if** (o == 0)

**return** (dist(p0, \*p2) >= dist(p0, \*p1))? -1 : 1;

**return** (o == 2)? -1: 1;

}

// Prints convex hull of a set of n points.

**void** convexHull(Point points[], **int** n)

{

// Find the bottommost point

**int** ymin = points[0].y, min = 0;

**for** (**int** i = 1; i < n; i++)

{

**int** y = points[i].y;

Output:

(0, 3)

(4, 4)

(3, 1)

(0, 0)

**Time Complexity:** Let n be the number of input points. The algorithm takes O(nLogn)

time if we use a O(nLogn) sorting algorithm.

// Pick the bottom-most or chose the left most point in case of tie

**if** ((y < ymin) || (ymin == y && points[i].x < points[min].x))

ymin = points[i].y, min = i;

}

// Place the bottom-most point at first position

swap(points[0], points[min]);

// Sort n-1 points with respect to the first point. A point p1 comes

// before p2 in sorted ouput if p2 has larger polar angle (in

// counterclockwise direction) than p1

p0 = points[0];

**qsort**(&points[1], n-1, **sizeof**(Point), compare);

// Create an empty stack and push first three points to it.

stack<Point> S;

S.push(points[0]);

S.push(points[1]);

S.push(points[2]);

// Process remaining n-3 points

**for** (**int** i = 3; i < n; i++)

{

// Keep removing top while the angle formed by points next-to-top,

// top, and points[i] makes a non-left turn

**while** (orientation(nextToTop(S), S.top(), points[i]) != 2)

S.pop();

S.push(points[i]);

}

// Now stack has the output points, print contents of stack

**while** (!S.empty())

{

Point p = S.top();

cout << "(" << p.x << ", " << p.y <<")" << endl;

S.pop();

}

}

// Driver program to test above functions

**int** main()

{

Point points[] = {{0, 3}, {1, 1}, {2, 2}, {4, 4},

{0, 0}, {1, 2}, {3, 1}, {3, 3}};

**int** n = **sizeof**(points)/**sizeof**(points[0]);

convexHull(points, n);

**return** 0;

}

The first step (finding the bottom-most point) takes O(n) time. The second step (sorting

points) takes O(nLogn) time. In third step, every element is pushed and popped at most

one time. So the third step to process points one by one takes O(n) time, assuming that

the stack operations take O(1) time. Overall complexity is O(n) + O(nLogn) + O(n) which

is O(nLogn)

**References:**

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E.

Leiserson, Ronald L. Rivest

http://www.dcs.gla.ac.uk/~pat/52233/slides/Hull1x1.pdf

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above

40. How to check if a given number is Fibonacci number?

Given a number ‘n’, how to check if n is a Fibonacci number.

A simple way is to generate Fibonacci numbers until the generated number is greater

than or equal to ‘n’. Following is an interesting property about Fibonacci numbers that

can also be used to check if a given number is Fibonacci or not.

*A number is Fibonacci if and only if one or both of (5\*n2 + 4) or (5\*n2 – 4) is a perfect*

*square* (Source: Wiki). Following is a simple program based on this concept.

Output:

1 is a Fibonacci Number

2 is a Fibonacci Number

3 is a Fibonacci Number

4 is a not Fibonacci Number

5 is a Fibonacci Number

6 is a not Fibonacci Number

7 is a not Fibonacci Number

8 is a Fibonacci Number

9 is a not Fibonacci Number

10 is a not Fibonacci Number

This article is contributed by **Abhay Rathi**. Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above

41. Russian Peasant Multiplication

Given two integers, write a function to multiply them without using multiplication operator.

// C++ program to check if x is a perfect square

#include <iostream>

#include <math.h>

**using namespace** std;

// A utility function that returns true if x is perfect square

**bool** isPerfectSquare(**int** x)

{

**int** s = **sqrt**(x);

**return** (s\*s == x);

}

// Returns true if n is a Fibinacci Number, else false

**bool** isFibonacci(**int** n)

{

// n is Fibinacci if one of 5\*n\*n + 4 or 5\*n\*n - 4 or both

// is a perferct square

**return** isPerfectSquare(5\*n\*n + 4) ||

isPerfectSquare(5\*n\*n - 4);

}

// A utility function to test above functions

**int** main()

{

**for** (**int** i = 1; i <= 10; i++)

isFibonacci(i)? cout << i << " is a Fibonacci Number \n":

cout << i << " is a not Fibonacci Number \n" ;

**return** 0;

}

There are many other ways to multiply two numbers (For example, see this). One

interesting method is the Russian peasant algorithm. The idea is to double the first

number and halve the second number repeatedly till the second number doesn’t become

1. In the process, whenever the second number become odd, we add the first number to

result (result is initialized as 0)

The following is simple algorithm.

Let the two given numbers be 'a' and 'b'

1) Initialize result 'res' as 0.

2) Do following while 'b' is greater than 0

a) If 'b' is odd, add 'a' to 'res'

b) Double 'a' and halve 'b'

3) Return 'res'.

Output:

18

240

**How does this work?**

The value of a\*b is same as (a\*2)\*(b/2) if b is even, otherwise the value is same as

((a\*2)\*(b/2) + a). In the while loop, we keep multiplying ‘a’ with 2 and keep dividing ‘b’ by

2. If ‘b’ becomes odd in loop, we add ‘a’ to ‘res’. When value of ‘b’ becomes 1, the value

of ‘res’ + ‘a’, gives us the result.

#include <iostream>

**using namespace** std;

// A method to multiply two numbers using Russian Peasant method

unsigned **int** russianPeasant(unsigned **int** a, unsigned **int** b)

{

**int** res = 0; // initialize result

// While second number doesn't become 1

**while** (b > 0)

{

// If second number becomes odd, add the first number to result

**if** (b & 1)

res = res + a;

// Double the first number and halve the second number

a = a << 1;

b = b >> 1;

}

**return** res;

}

// Driver program to test above function

**int** main()

{

cout << russianPeasant(18, 1) << endl;

cout << russianPeasant(20, 12) << endl;

**return** 0;

}

Note that when ‘b’ is a power of 2, the ‘res’ would remain 0 and ‘a’ would have the

multiplication. See the reference for more information.

**Reference:**

http://mathforum.org/dr.math/faq/faq.peasant.html

This article is compiled by **Shalki Agarwal**. Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above

42. Program for nth Catalan Number

Catalan numbers are a sequence of natural numbers that occurs in many interesting

counting problems like following.

**1)** Count the number of expressions containing n pairs of parentheses which are

correctly matched. For n = 3, possible expressions are ((())), ()(()), ()()(), (())(), (()()).

**2)** Count the number of possible Binary Search Trees with n keys (See this)

**3)** Count the number of full binary trees (A rooted binary tree is full if every vertex has

either two children or no children) with n+1 leaves.

See this for more applications.

The first few Catalan numbers for n = 0, 1, 2, 3, … are **1, 1, 2, 5, 14, 42, 132, 429, 1430,**

**4862, …**

**Recursive Solution**

Catalan numbers satisfy the following recursive formula.

Following is C++ implementation of above recursive formula.

Output :

1 1 2 5 14 42 132 429 1430 4862

Time complexity of above implementation is equivalent to nth catalan number.

The value of nth catalan number is exponential that makes the time complexity

exponential.

**Dynamic Programming Solution**

We can observe that the above recursive implementation does a lot of repeated work

(we can the same by drawing recursion tree). Since there are overlapping subproblems,

we can use dynamic programming for this. Following is a Dynamic programming based

implementation in C++.

#include<iostream>

**using namespace** std;

// A recursive function to find nth catalan number

unsigned **long int** catalan(unsigned **int** n)

{

// Base case

**if** (n <= 1) **return** 1;

// catalan(n) is sum of catalan(i)\*catalan(n-i-1)

unsigned **long int** res = 0;

**for** (**int** i=0; i<n; i++)

res += catalan(i)\*catalan(n-i-1);

**return** res;

}

// Driver program to test above function

**int** main()

{

**for** (**int** i=0; i<10; i++)

cout << catalan(i) << " ";

**return** 0;

}

Output:

1 1 2 5 14 42 132 429 1430 4862

Time Complexity: Time complexity of above implementation is O(n2)

**Using Binomial Coefficient**

We can also use the below formula to find nth catalan number in O(n) time.

We have discussed a O(n) approach to find binomial coefficient nCr.

#include<iostream>

**using namespace** std;

// A dynamic programming based function to find nth

// Catalan number

unsigned **long int** catalanDP(unsigned **int** n)

{

// Table to store results of subproblems

unsigned **long int** catalan[n+1];

// Initialize first two values in table

catalan[0] = catalan[1] = 1;

// Fill entries in catalan[] using recursive formula

**for** (**int** i=2; i<=n; i++)

{

catalan[i] = 0;

**for** (**int** j=0; j<i; j++)

catalan[i] += catalan[j] \* catalan[i-j-1];

}

// Return last entry

**return** catalan[n];

}

// Driver program to test above function

**int** main()

{

**for** (**int** i = 0; i < 10; i++)

cout << catalanDP(i) << " ";

**return** 0;

}

Output:

1 1 2 5 14 42 132 429 1430 4862

Time Complexity: Time complexity of above implementation is O(n).

We can also use below formula to find nth catalan number in O(n) time.

.

**References:**

http://en.wikipedia.org/wiki/Catalan\_number

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above

#include<iostream>

**using namespace** std;

// Returns value of Binomial Coefficient C(n, k)

unsigned **long int** binomialCoeff(unsigned **int** n, unsigned **int** k)

{

unsigned **long int** res = 1;

// Since C(n, k) = C(n, n-k)

**if** (k > n - k)

k = n - k;

// Calculate value of [n\*(n-1)\*---\*(n-k+1)] / [k\*(k-1)\*---\*1]

**for** (**int** i = 0; i < k; ++i)

{

res \*= (n - i);

res /= (i + 1);

}

**return** res;

}

// A Binomial coefficient based function to find nth catalan

// number in O(n) time

unsigned **long int** catalan(unsigned **int** n)

{

// Calculate value of 2nCn

unsigned **long int** c = binomialCoeff(2\*n, n);

// return 2nCn/(n+1)

**return** c/(n+1);

}

// Driver program to test above functions

**int** main()

{

**for** (**int** i = 0; i < 10; i++)

cout << catalan(i) << " ";

**return** 0;

}

43. Count trailing zeroes in factorial of a number

Given an integer n, write a function that returns count of trailing zeroes in n!.

**Examples:**

Input: n = 5

Output: 1

Factorial of 5 is 20 which has one trailing 0.

Input: n = 20

Output: 4

Factorial of 20 is 2432902008176640000 which has

4 trailing zeroes.

Input: n = 100

Output: 24

A simple method is to first calculate factorial of n, then count trailing 0s in the result (We

can count trailing 0s by repeatedly dividing the factorial by 10 till the remainder is 0).

The above method can cause overflow for a slightly bigger numbers as factorial of a

number is a big number (See factorial of 20 given in above examples). The idea is to

consider prime factors of a factorial n. A trailing zero is always produced by prime

factors 2 and 5. If we can count the number of 5s and 2s, our task is done. Consider the

following examples.

**n = 5:** There is one 5 and 3 2s in prime factors of 5! (2 \* 2 \* 2 \* 3 \* 5). So count of

trailing 0s is 1.

**n = 11:** There are two 5s and three 2s in prime factors of 11! (2 8 \* 34 \* 52 \* 7). So

count of trailing 0s is 2.

We can easily observe that the number of 2s in prime factors is always more than or

equal to the number of 5s. So if we count 5s in prime factors, we are done. *How to*

*count total number of 5s in prime factors of n!?* A simple way is to calculate floor(n/5).

For example, 7! has one 5, 10! has two 5s. It is done yet, there is one more thing to

consider. Numbers like 25, 125, etc have more than one 5. For example if we consider

28!, we get one extra 5 and number of 0s become 6. Handling this is simple, first divide

n by 5 and remove all single 5s, then divide by 25 to remove extra 5s and so on.

Following is the summarized formula for counting trailing 0s.

Trailing 0s in n! = Count of 5s in prime factors of n!

= floor(n/5) + floor(n/25) + floor(n/125) + ....

Following is C++ program based on above formula.

Output:

Count of trailing 0s in 100! is 24

This article is contributed by **Rahul Jain**. Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above

44. Horner’s Method for Polynomial Evaluation

Given a polynomial of the form cnxn + cn-1xn-1 + cn-2xn-2 + … + c1x + c0 and a value of

x, find the value of polynomial for a given value of x. Here cn, cn-1, .. are integers (may

be negative) and n is a positive integer.

Input is in the form of an array say *poly[]* where poly[0] represents coefficient for xn and

poly[1] represents coefficient for xn-1 and so on.

Examples:

// Evaluate value of 2x3 - 6x2 + 2x - 1 for x = 3

Input: poly[] = {2, -6, 2, -1}, x = 3

Output: 5

// C++ program to count trailing 0s in n!

#include <iostream>

**using namespace** std;

// Function to return trailing 0s in factorial of n

**int** findTrailingZeros(**int** n)

{

// Initialize result

**int** count = 0;

// Keep dividing n by powers of 5 and update count

**for** (**int** i=5; n/i>=1; i \*= 5)

count += n/i;

**return** count;

}

// Driver program to test above function

**int** main()

{

**int** n = 100;

cout << "Count of trailing 0s in " << 100

<< "! is " << findTrailingZeros(n);

**return** 0;

}

// Evaluate value of 2x3 + 3x + 1 for x = 2

Input: poly[] = {2, 0, 3, 1}, x = 2

Output: 23

A naive way to evaluate a polynomial is to one by one evaluate all terms. First calculate

xn, multiply the value with cn, repeat the same steps for other terms and return the sum.

Time complexity of this approach is O(n2) if we use a simple loop for evaluation of xn.

Time complexity can be improved to O(nLogn) if we use O(Logn) approach for

evaluation of xn.

**Horner’s method** can be used to evaluate polynomial in O(n) time. To understand the

method, let us consider the example of 2x3 – 6x2 + 2x – 1. The polynomial can be

evaluated as ((2x – 6)x + 2)x – 1. The idea is to initialize result as coefficient of xn which

is 2 in this case, repeatedly multiply result with x and add next coefficient to result. Finally

return result.

Following is C++ implementation of Horner’s Method.

Output:

Value of polynomial is 5

Time Complexity: O(n)

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above

#include <iostream>

**using namespace** std;

// returns value of poly[0]x(n-1) + poly[1]x(n-2) + .. + poly[n-1]

**int** horner(**int** poly[], **int** n, **int** x)

{

**int** result = poly[0]; // Initialize result

// Evaluate value of polynomial using Horner's method

**for** (**int** i=1; i<n; i++)

result = result\*x + poly[i];

**return** result;

}

// Driver program to test above function.

**int** main()

{

// Let us evaluate value of 2x3 - 6x2 + 2x - 1 for x = 3

**int** poly[] = {2, -6, 2, -1};

**int** x = 3;

**int** n = **sizeof**(poly)/**sizeof**(poly[0]);

cout << "Value of polynomial is " << horner(poly, n, x);

**return** 0;

}

45. Write a function that generates one of 3 numbers according to

given probabilities

You are given a function rand(a, b) which generates equiprobable random numbers

between [a, b] inclusive. Generate 3 numbers x, y, z with probability P(x), P(y), P(z) such

that P(x) + P(y) + P(z) = 1 using the given rand(a,b) function.

The idea is to utilize the equiprobable feature of the rand(a,b) provided. ***Let the given***

***probabilities be in percentage form, for example P(x)=40%, P(y)=25%, P(z)=35%.***.

Following are the detailed steps.

**1)** Generate a random number between 1 and 100. Since they are equiprobable, the

probability of each number appearing is 1/100.

**2)** Following are some important points to note about generated random number ‘r’.

a) ‘r’ is smaller than or equal to P(x) with probability P(x)/100.

b) ‘r’ is greater than P(x) and smaller than or equal P(x) + P(y) with P(y)/100.

c) ‘r’ is greater than P(x) + P(y) and smaller than or equal 100 (or P(x) + P(y) + P(z)) with

probability P(z)/100.

This function will solve the purpose of generating 3 numbers with given three

probabilities.

This article is contributed by **Harsh Agarwal**. Please write comments if you find

anything incorrect, or you want to share more information about the topic discussed

// This function generates 'x' with probability px/100, 'y' with

// probability py/100 and 'z' with probability pz/100:

// Assumption: px + py + pz = 100 where px, py and pz lie

// between 0 to 100

**int** random(**int** x, **int** y, **int** z, **int** px, **int** py, **int** pz)

{

// Generate a number from 1 to 100

**int** r = **rand**(1, 100);

// r is smaller than px with probability px/100

**if** (r <= px)

**return** x;

// r is greater than px and smaller than or equal to px+py

// with probability py/100

**if** (r <= (px+py))

**return** y;

// r is greater than px+py and smaller than or equal to 100

// with probability pz/100

**else**

**return** z;

}

above

46. Find the smallest number whose digits multiply to a given

number n

Given a number ‘n’, find the smallest number ‘p’ such that if we multiply all digits of ‘p’,

we get ‘n’. The result ‘p’ should have minimum two digits.

Examples:

Input: n = 36

Output: p = 49

// Note that 4\*9 = 36 and 49 is the smallest such number

Input: n = 100

Output: p = 455

// Note that 4\*5\*5 = 100 and 455 is the smallest such number

Input: n = 1

Output:p = 11

// Note that 1\*1 = 1

Input: n = 13

Output: Not Possible

For a given n, following are the two cases to be considered.

**Case 1: n < 10** When n is smaller than n, the output is always n+10. For example for n =

7, output is 17. For n = 9, output is 19.

**Case 2: n >= 10** Find all factors of n which are between 2 and 9 (both inclusive). The

idea is to start searching from 9 so that the number of digits in result are minimized. For

example 9 is preferred over 33 and 8 is preferred over 24.

Store all found factors in an array. The array would contain digits in non-increasing

order, so finally print the array in reverse order.

Following is C implementation of above concept.

Output:

#include<stdio.h>

// Maximum number of digits in output

#define MAX 50

// prints the smallest number whose digits multiply to n

**void** findSmallest(**int** n)

{

**int** i, j=0;

**int** res[MAX]; // To sore digits of result in reverse order

// Case 1: If number is smaller than 10

**if** (n < 10)

{

**printf**("%d", n+10);

**return**;

}

// Case 2: Start with 9 and try every possible digit

**for** (i=9; i>1; i--)

{

// If current digit divides n, then store all

// occurrences of current digit in res

**while** (n%i == 0)

{

n = n/i;

res[j] = i;

j++;

}

}

// If n could not be broken in form of digits (prime factors of n

// are greater than 9)

**if** (n > 10)

{

**printf**("Not possible");

**return**;

}

// Print the result array in reverse order

**for** (i=j-1; i>=0; i--)

**printf**("%d", res[i]);

}

// Driver program to test above function

**int** main()

{

findSmallest(7);

**printf**("\n");

findSmallest(36);

**printf**("\n");

findSmallest(13);

**printf**("\n");

findSmallest(100);

**return** 0;

}

17

49

Not possible

455

This article is contributed by **Ashish Bansal**. Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above

47. Calculate the angle between hour hand and minute hand

This problem is know as Clock angle problem where we need to find angle between

hands of an analog clock at .

Examples:

Input: h = 12:00, m = 30.00

Output: 165 degree

Input: h = 3.00, m = 30.00

Output: 75 degree

The idea is to take 12:00 (h = 12, m = 0) as a reference. Following are detailed steps.

**1)** Calculate the angle made by hour hand with respect to 12:00 in h hours and m

minutes.

**2)** Calculate the angle made by minute hand with respect to 12:00 in h hours and m

minutes.

**3)** The difference between two angles is the angle between two hands.

**How to calculate the two angles with respect to 12:00?**

The minute hand moves 360 degree in 60 minute(or 6 degree in one minute) and hour

hand moves 360 degree in 12 hours(or 0.5 degree in 1 minute). In h hours and m

minutes, the minute hand would move (h\*60 + m)\*6 and hour hand would move (h\*60 +

m)\*0.5.

Output:

90

75

**Exercise:** Find all times when hour and minute hands get superimposed.

This article is contributed by **Ashish Bansal**. Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above

48. Count Possible Decodings of a given Digit Sequence

Let 1 represent ‘A’, 2 represents ‘B’, etc. Given a digit sequence, count the number of

possible decodings of the given digit sequence.

// C program to find angle between hour and minute hands

#include <stdio.h>

#include <stdlib.h>

// Utility function to find minimum of two integers

**int** min(**int** x, **int** y) { **return** (x < y)? x: y; }

**int** calcAngle(**double** h, **double** m)

{

// validate the input

**if** (h <0 || m < 0 || h >12 || m > 60)

**printf**("Wrong input");

**if** (h == 12) h = 0;

**if** (m == 60) m = 0;

// Calculate the angles moved by hour and minute hands

// with reference to 12:00

**int** hour\_angle = 0.5 \* (h\*60 + m);

**int** minute\_angle = 6\*m;

// Find the difference between two angles

**int** angle = **abs**(hour\_angle - minute\_angle);

// Return the smaller angle of two possible angles

angle = min(360-angle, angle);

**return** angle;

}

// Driver program to test above function

**int** main()

{

**printf**("%d \n", calcAngle(9, 60));

**printf**("%d \n", calcAngle(3, 30));

**return** 0;

}

Examples:

Input: digits[] = "121"

Output: 3

// The possible decodings are "ABA", "AU", "LA"

Input: digits[] = "1234"

Output: 3

// The possible decodings are "ABCD", "LCD", "AWD"

An empty digit sequence is considered to have one decoding. It may be assumed that

the input contains valid digits from 0 to 9 and there are no leading 0’s, no extra trailing

0’s and no two or more consecutive 0’s.

**We strongly recommend to minimize the browser and try this yourself first.**

This problem is recursive and can be broken in sub-problems. We start from end of the

given digit sequence. We initialize the total count of decodings as 0. We recur for two

subproblems.

1) If the last digit is non-zero, recur for remaining (n-1) digits and add the result to total

count.

2) If the last two digits form a valid character (or smaller than 27), recur for remaining (n-

2) digits and add the result to total count.

Following is C++ implementation of the above approach.

Output:

Count is 3

The time complexity of above the code is exponential. If we take a closer look at the

above program, we can observe that the recursive solution is similar to Fibonacci

Numbers. Therefore, we can optimize the above solution to work in O(n) time using

Dynamic Programming. Following is C++ implementation for the same.

// A naive recursive C++ implementation to count number of decodings

// that can be formed from a given digit sequence

#include <iostream>

#include <cstring>

**using namespace** std;

// Given a digit sequence of length n, returns count of possible

// decodings by replacing 1 with A, 2 woth B, ... 26 with Z

**int** countDecoding(**char** \*digits, **int** n)

{

// base cases

**if** (n == 0 || n == 1)

**return** 1;

**int** count = 0; // Initialize count

// If the last digit is not 0, then last digit must add to

// the number of words

**if** (digits[n-1] > '0')

count = countDecoding(digits, n-1);

// If the last two digits form a number smaller than or equal to 26,

// then consider last two digits and recur

**if** (digits[n-2] < '2' || (digits[n-2] == '2' && digits[n-1] < '7') )

count += countDecoding(digits, n-2);

**return** count;

}

// Driver program to test above function

**int** main()

{

**char** digits[] = "1234";

**int** n = **strlen**(digits);

cout << "Count is " << countDecoding(digits, n);

**return** 0;

}

Output:

Count is 3

Time Complexity of the above solution is O(n) and it requires O(n) auxiliary space. We

can reduce auxiliary space to O(1) by using space optimized version discussed in the

Fibonacci Number Post.

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above

49. Find next greater number with same set of digits

Given a number n, find the smallest number that has same set of digits as n and is

// A Dynamic Programming based C++ implementation to count decodings

#include <iostream>

#include <cstring>

**using namespace** std;

// A Dynamic Programming based function to count decodings

**int** countDecodingDP(**char** \*digits, **int** n)

{

**int** count[n+1]; // A table to store results of subproblems

count[0] = 1;

count[1] = 1;

**for** (**int** i = 2; i <= n; i++)

{

count[i] = 0;

// If the last digit is not 0, then last digit must add to

// the number of words

**if** (digits[i-1] > '0')

count[i] = count[i-1];

// If second last digit is smaller than 2 and last digit is

// smaller than 7, then last two digits form a valid character

**if** (digits[i-2] < '2' || (digits[i-2] == '2' && digits[i-1] < '7'

count[i] += count[i-2];

}

**return** count[n];

}

// Driver program to test above function

**int** main()

{

**char** digits[] = "1234";

**int** n = **strlen**(digits);

cout << "Count is " << countDecodingDP(digits, n);

**return** 0;

}

greater than n. If x is the greatest possible number with its set of digits, then print “not

possible”.

Examples:

For simplicity of implementation, we have considered input number as a string.

Input: n = "218765"

Output: "251678"

Input: n = "1234"

Output: "1243"

Input: n = "4321"

Output: "Not Possible"

Input: n = "534976"

Output: "536479"

**We strongly recommend to minimize the browser and try this yourself first.**

Following are few observations about the next greater number.

1) If all digits sorted in descending order, then output is always “Not Possible”. For

example, 4321.

2) If all digits are sorted in ascending order, then we need to swap last two digits. For

example, 1234.

3) For other cases, we need to process the number from rightmost side (why? because

we need to find the smallest of all greater numbers)

You can now try developing an algorithm yourself.

Following is the algorithm for finding the next greater number.

**I)** Traverse the given number from rightmost digit, keep traversing till you find a digit

which is smaller than the previously traversed digit. For example, if the input number is

“534976”, we stop at **4** because 4 is smaller than next digit 9. If we do not find such a

digit, then output is “Not Possible”.

**II)** Now search the right side of above found digit ‘d’ for the smallest digit greater than

‘d’. For “53**4**976″, the right side of 4 contains “976”. The smallest digit greater than 4 is

**6**.

**III)** Swap the above found two digits, we get 53**6**97**4** in above example.

**IV)** Now sort all digits from position next to ‘d’ to the end of number. The number that we

get after sorting is the output. For above example, we sort digits in bold 536**974**. We get

“536**479**” which is the next greater number for input 534976.

Following is C++ implementation of above approach.

// C++ program to find the smallest number which greater than a given number

// and has same set of digits as given number

#include <iostream>

#include <cstring>

#include <algorithm>

**using namespace** std;

// Utility function to swap two digits

**void** swap(**char** \*a, **char** \*b)

{

**char** temp = \*a;

\*a = \*b;

\*b = temp;

}

// Given a number as a char array number[], this function finds the

// next greater number. It modifies the same array to store the result

**void** findNext(**char** number[], **int** n)

{

**int** i, j;

// I) Start from the right most digit and find the first digit that is

// smaller than the digit next to it.

**for** (i = n-1; i > 0; i--)

**if** (number[i] > number[i-1])

**break**;

// If no such digit is found, then all digits are in descending order

// means there cannot be a greater number with same set of digits

**if** (i==0)

{

cout << "Next number is not possible";

**return**;

}

// II) Find the smallest digit on right side of (i-1)'th digit that is

// greater than number[i-1]

**int** x = number[i-1], smallest = i;

**for** (j = i+1; j < n; j++)

**if** (number[j] > x && number[j] < number[smallest])

smallest = j;

// III) Swap the above found smallest digit with number[i-1]

swap(&number[smallest], &number[i-1]);

// IV) Sort the digits after (i-1) in ascending order

sort(number + i, number + n);

cout << "Next number with same set of digits is " << number;

**return**;

}

// Driver program to test above function

**int** main()

{

**char** digits[] = "534976";

**int** n = **strlen**(digits);

findNext(digits, n);

**return** 0;

}

Output:

Next number with same set of digits is 536479

The above implementation can be optimized in following ways.

1) We can use binary search in step II instead of linear search.

2) In step IV, instead of doing simple sort, we can apply some clever technique to do it

in linear time. Hint: We know that all digits are linearly sorted in reverse order except one

digit which was swapped.

With above optimizations, we can say that the time complexity of this method is O(n).

This article is contributed by **Rahul Jain**. Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above

50. Print squares of first n natural numbers without using \*, / and -

Given a natural number ‘n’, print squares of first n natural numbers without using \*, / and -

.

Input: n = 5

Output: 0 1 4 9 16

Input: n = 6

Output: 0 1 4 9 16 25

**We strongly recommend to minimize the browser and try this yourself first.**

**Method 1:** The idea is to calculate next square using previous square value. Consider

the following relation between square of x and (x-1). We know square of (x-1) is (x-1)2 –

2\*x + 1. We can write x2 as

x2 = (x-1)2 + 2\*x - 1

x2 = (x-1)2 + x + (x - 1)

When writing an iterative program, we can keep track of previous value of x and add the

current and previous values of x to current value of square. This way we don’t even use

the ‘-‘ operator.

Output:

0 1 4 9 16

**Method 2:** Sum of first n odd numbers are squares of natural numbers from 1 to n. For

example 1, 1+3, 1+3+5, 1+3+5+7, 1+3+5+7+9, ….

Following is C++ program based on above concept. Thanks to Aadithya Umashanker

and raviteja for suggesting this method.

// C++ program to print squares of first 'n' natural numbers

// wothout using \*, / and -

#include<iostream>

**using namespace** std;

**void** printSquares(**int** n)

{

// Initialize 'square' and previous value of 'x'

**int** square = 0, prev\_x = 0;

// Calculate and print squares

**for** (**int** x = 0; x < n; x++)

{

// Update value of square using previous value

square = (square + x + prev\_x);

// Print square and update prev for next iteration

cout << square << " ";

prev\_x = x;

}

}

// Driver program to test above function

**int** main()

{

**int** n = 5;

printSquares(n);

}

Output:

0 1 4 9 16

This article is contributed by **Sachin**. Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above

51. Find n’th number in a number system with only 3 and 4

Given a number system with only 3 and 4. Find the nth number in the number system.

First few numbers in the number system are: 3, 4, 33, 34, 43, 44, 333, 334, 343, 344,

433, 434, 443, 444, 3333, 3334, 3343, 3344, 3433, 3434, 3443, 3444, …

Source: Zoho Interview

**We strongly recommend to minimize the browser and try this yourself first.**

We can generate all numbers with i digits using the numbers with (i-1) digits. The idea is

to first add a ‘3’ as prefix in all numbers with (i-1) digit, then add a ‘4’. For example, the

numbers with 2 digits are 33, 34, 43 and 44. The numbers with 3 digits are 333, 334,

343, 344, 433, 434, 443 and 444 which can be generated by first adding a 3 as prefix,

// C++ program to print squares of first 'n' natural numbers

// wothout using \*, / and -

#include<iostream>

**using namespace** std;

**void** printSquares(**int** n)

{

// Initialize 'square' and first odd number

**int** square = 0, odd = 1;

// Calculate and print squares

**for** (**int** x = 0; x < n; x++)

{

// Print square

cout << square << " ";

// Update 'square' and 'odd'

square = square + odd;

odd = odd + 2;

}

}

// Driver program to test above function

**int** main()

{

**int** n = 5;

printSquares(n);

}

then 4.

Following are detailed steps.

1) Create an array 'arr[]' of strings size n+1.

2) Initialize arr[0] as empty string. (Number with 0 digits)

3) Do following while array size is smaller than or equal to n

.....a) Generate numbers by adding a 3 as prefix to the numbers generated

in previous iteration. Add these numbers to arr[]

.....a) Generate numbers by adding a 4 as prefix to the numbers generated

in previous iteration. Add these numbers to arr[]

Thanks to kaushik Lele for suggesting this idea in a comment here. Following is C++

implementation for the same.

Output:

3

4

33

34

43

44

333

334

343

// C++ program to find n'th number in a number system with only 3 and 4

#include <iostream>

**using namespace** std;

// Function to find n'th number in a number system with only 3 and 4

**void** find(**int** n)

{

// An array of strings to store first n numbers. arr[i] stores i'th number

string arr[n+1];

arr[0] = ""; // arr[0] stores the empty string (String with 0 digits)

// size indicates number of current elements in arr[]. m indicates

// number of elements added to arr[] in previous iteration.

**int** size = 1, m = 1;

// Every iteration of following loop generates and adds 2\*m numbers to

// arr[] using the m numbers generated in previous iteration.

**while** (size <= n)

{

// Consider all numbers added in previous iteration, add a prefix

// "3" to them and add new numbers to arr[]

**for** (**int** i=0; i<m && (size+i)<=n; i++)

arr[size + i] = "3" + arr[size - m + i];

// Add prefix "4" to numbers of previous iteration and add new

// numbers to arr[]

**for** (**int** i=0; i<m && (size + m + i)<=n; i++)

arr[size + m + i] = "4" + arr[size - m + i];

// Update no. of elements added in previous iteration

m = m<<1; // Or m = m\*2;

// Update size

size = size + m;

}

cout << arr[n] << endl;

}

// Driver program to test above functions

**int** main()

{

**for** (**int** i = 1; i < 16; i++)

find(i);

**return** 0;

}

344

433

434

443

444

3333

This article is contributed by **Raman**. Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above.

52. Count Distinct Non-Negative Integer Pairs (x, y) that Satisfy the

Inequality x\*x + y\*y < n

Given a positive number n, count all distinct Non-Negative Integer pairs (x, y) that satisfy

the inequality x\*x + y\*y < n.

Examples:

Input: n = 5

Output: 6

The pairs are (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (0, 2)

Input: n = 6

Output: 8

The pairs are (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (0, 2),

(1, 2), (2, 1)

A **Simple Solution** is to run two loops. The outer loop goes for all possible values of x

(from 0 to √n). The inner loops picks all possible values of y for current value of x

(picked by outer loop). Following is C++ implementation of simple solution.

Output:

Total Number of distinct Non-Negative pairs is 8

An upper bound for time complexity of the above solution is O(n). The outer loop runs

√n times. The inner loop runs at most √n times.

Using an **Efficient Solution**, we can find the count in O(√n) time. The idea is to first find

the count of all y values corresponding the 0 value of x. Let count of distinct y values be

yCount. We can find yCount by running a loop and comparing yCount\*yCount with n.

After we have initial yCount, we can one by one increase value of x and find the next

value of yCount by reducing yCount.

#include <iostream>

**using namespace** std;

// This function counts number of pairs (x, y) that satisfy

// the inequality x\*x + y\*y < n.

**int** countSolutions(**int** n)

{

**int** res = 0;

**for** (**int** x = 0; x\*x < n; x++)

**for** (**int** y = 0; x\*x + y\*y < n; y++)

res++;

**return** res;

}

// Driver program to test above function

**int** main()

{

cout << "Total Number of distinct Non-Negative pairs is "

<< countSolutions(6) << endl;

**return** 0;

}

Output:

Total Number of distinct Non-Negative pairs is 8

**Time Complexity** of the above solution seems more but if we take a closer look, we can

see that it is O(√n). In every step inside the inner loop, value of yCount is decremented

by 1. The value yCount can decrement at most O(√n) times as yCount is count y values

for x = 0. In the outer loop, the value of x is incremented. The value of x can also

increment at most O(√n) times as the last x is for yCount equals to 1.

This article is contributed by **Sachin Gupta**. Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above.

// An efficient C program to find different (x, y) pairs that

// satisfy x\*x + y\*y < n.

#include <iostream>

**using namespace** std;

// This function counts number of pairs (x, y) that satisfy

// the inequality x\*x + y\*y < n.

**int** countSolutions(**int** n)

{

**int** x = 0, yCount, res = 0;

// Find the count of different y values for x = 0.

**for** (yCount = 0; yCount\*yCount < n; yCount++) ;

// One by one increase value of x, and find yCount for

// current x. If yCount becomes 0, then we have reached

// maximum possible value of x.

**while** (yCount != 0)

{

// Add yCount (count of different possible values of y

// for current x) to result

res += yCount;

// Increment x

x++;

// Update yCount for current x. Keep reducing yCount while

// the inequality is not satisfied.

**while** (yCount != 0 && (x\*x + (yCount-1)\*(yCount-1) >= n))

yCount--;

}

**return** res;

}

// Driver program to test above function

**int** main()

{

cout << "Total Number of distinct Non-Negative pairs is "

<< countSolutions(6) << endl;

**return** 0;

}

53. Birthday Paradox

*How many people must be there in a room to make the probability 100% that two*

*people in the room have same birthday?*

Answer: 367 (since there are 366 possible birthdays, including February 29).

The above question was simple. Try the below question yourself.

***How many people must be there in a room to make the probability 50% that two***

***people in the room have same birthday?***

Answer: 23

The number is surprisingly very low. In fact, we need only 70 people to make the

probability 99.9 %.

Let us discuss the generalized formula.

**What is the probability that two persons among n have same birthday?**

Let the probability that two people in a room with n have same birthday be P(same).

P(Same) can be easily evaluated in terms of P(different) where P(different) is the

probability that all of them have different birthday.

P(same) = 1 – P(different)

P(different) can be written as 1 x (364/365) x (363/365) x (362/365) x …. x (1 – (n-

1)/365)

*How did we get the above expression?*

Persons from first to last can get birthdays in following order for all birthdays to be

distinct:

The first person can have any birthday among 365

The second person should have a birthday which is not same as first person

The third person should have a birthday which is not same as first two persons.

…………….

……………

The n’th person should have a birthday which is not same as any of the earlier

considered (n-1) persons.

**Approximation of above expression**

The above expression can be approximated using Taylor’s Series.

provides a first-order approximation for ex for x << 1:

To apply this approximation to the first expression derived for p(different), set x = -a /

365\ . Thus,

(1)

The above expression derived for p(different) can be written as

1 x (1 – 1/365) x (1 – 2/365) x (1 – 3/365) x …. x (1 – (n-1)/365)

By putting the value of as , we get following.

Therefore,

p(same) = 1- p(different)

An even coarser approximation is given by

p(same)

By taking Log on both sides, we get the reverse formula.

Using the above approximate formula, we can approximate number of people for a

given probability. For example the following C++ function find() returns the smallest n for

which the probability is greater than the given p.

**C++ Implementation of approximate formula.**

The following is C++ program to approximate number of people for a given probability.

Output:

30

**Source:**

http://en.wikipedia.org/wiki/Birthday\_problem

**Applications:**

// C++ program to approximate number of people in Birthday Paradox

// problem

#include <cmath>

#include <iostream>

**using namespace** std;

// Returns approximate number of people for a given probability

**int** find(**double** p)

{

**return ceil**(**sqrt**(2\*365\***log**(1/(1-p))));

}

**int** main()

{

cout << find(0.70);

}

1) Birthday Paradox is generally discussed with hashing to show importance of collision

handling even for a small set of keys.

2) Birthday Attack

This article is contributed by **Shubham**. Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above.

54. How to check if an instance of 8 puzzle is solvable?

**What is 8 puzzle?**

Given a 3×3 board with 8 tiles (every tile has one number from 1 to 8) and one empty

space. The objective is to place the numbers on tiles in order using the empty space.

We can slide four adjacent (left, right, above and below) tiles into the empty space.

**How to find if given state is solvable?**

Following are two examples, the first example can reach goal state by a series of slides.

The second example cannot.

Following is simple rule to check if a 8 puzzle is solvable.

*It is not possible to solve an instance of 8 puzzle if number of inversions is odd in the*

*input state.* In the examples given in above figure, the first example has 10 inversions,

therefore solvable. The second example has 11 inversions, therefore unsolvable.

**What is inversion?**

A pair of tiles form an inversion if the the values on tiles are in reverse order of their

appearance in goal state. For example, the following instance of 8 puzzle has two

inversions, (8, 6) and (8, 7).

1 2 3

4 \_ 5

8 6 7

Following is a simple C++ program to check whether a given instance of 8 puzzle is

solvable or not. The idea is simple, we count inversions in the given 8 puzzle.

Output:

Solvable

Note that the above implementation uses simple algorithm for inversion count. It is done

this way for simplicity. The code can be optimized to O(nLogn) using the merge sort

based algorithm for inversion count.

**How does this work?**

The idea is based on the fact the parity of inversions remains same after a set of

moves, i.e., if the inversion count is odd in initial stage, then it remain odd after any

sequence of moves and if the inversion count is even, then it remains even after any

// C++ program to check if a given instance of 8 puzzle is solvable or not

#include <iostream>

**using namespace** std;

// A utility function to count inversions in given array 'arr[]'

**int** getInvCount(**int** arr[])

{

**int** inv\_count = 0;

**for** (**int** i = 0; i < 9 - 1; i++)

**for** (**int** j = i+1; j < 9; j++)

// Value 0 is used for empty space

**if** (arr[j] && arr[i] && arr[i] > arr[j])

inv\_count++;

**return** inv\_count;

}

// This function returns true if given 8 puzzle is solvable.

**bool** isSolvable(**int** puzzle[3][3])

{

// Count inversions in given 8 puzzle

**int** invCount = getInvCount((**int** \*)puzzle);

// return true if inversion count is even.

**return** (invCount%2 == 0);

}

/\* Driver progra to test above functions \*/

**int** main(**int** argv, **int**\*\* args)

{

**int** puzzle[3][3] = {{1, 8, 2},

{0, 4, 3}, // Value 0 is used for empty space

{7, 6, 5}};

isSolvable(puzzle)? cout << "Solvable":

cout << "Not Solvable";

**return** 0;

}

sequence of moves. In the goal state, there are 0 inversions. So we can reach goal

state only from a state which has even inversion count.

How parity of inversion count is invariant?

When we slide a tile, we either make a row move (moving a left or right tile into the blank

space), or make a column move (moving a up or down tile to the blank space).

**a)** A row move doesn’t change the inversion count. See following example

1 2 3 Row Move 1 2 3

4 \_ 5 ----------> \_ 4 5

8 6 7 8 6 7

Inversion count remains 2 after the move

1 2 3 Row Move 1 2 3

4 \_ 5 ----------> 4 5 \_

8 6 7 8 6 7

Inversion count remains 2 after the move

**b)** A column move does one of the following three.

…..(i) Increases inversion count by 2. See following example.

1 2 3 Column Move 1 \_ 3

4 \_ 5 -----------> 4 2 5

8 6 7 8 6 7

Inversion count increases by 2 (changes from 2 to 4)

…..(ii) Decreases inversion count by 2

1 3 4 Column Move 1 3 4

5 \_ 6 ------------> 5 2 6

7 2 8 7 \_ 8

Inversion count decreases by 2 (changes from 5 to 3)

…..(iii) Keeps the inversion count same.

1 2 3 Column Move 1 2 3

4 \_ 5 ------------> 4 6 5

7 6 8 7 \_ 8

Inversion count remains 1 after the move

So if a move either increases/decreases inversion count by 2, or keeps the inversion

count same, then it is not possible to change parity of a state by any sequence of

row/column moves.

**Exercise:** How to check if a given instance of 15 puzzle is solvable or not. In a 15

puzzle, we have 4×4 board where 15 tiles have a number and one empty space. Note

that the above simple rules of inversion count don’t directly work for 15 puzzle, the rules

need to be modified for 15 puzzle.

This article is contributed by Ishan. Please write comments if you find anything incorrect,

or you want to share more information about the topic discussed above

55. Factorial of a large number

**How to compute factorial of 100 using a C/C++ program?**

Factorial of 100 has 158 digits. It is not possible to store these many digits even if we

use long long int. Following is a simple solution where we use an array to store individual

digits of the result. The idea is to use basic mathematics for multiplication.

The following is detailed algorithm for finding factorial.

***factorial(n)***

1) Create an array ‘res[]’ of MAX size where MAX is number of maximum digits in

output.

2) Initialize value stored in ‘res[]’ as 1 and initialize ‘res\_size’ (size of ‘res[]’) as 1.

3) Do following for all numbers from x = 2 to n.

……a) Multiply x with res[] and update res[] and res\_size to store the multiplication

result.

***How to multiply a number ‘x’ with the number stored in res[]?***

The idea is to use simple school mathematics. We one by one multiply x with every digit

of res[]. The important point to note here is digits are multiplied from rightmost digit to

leftmost digit. If we store digits in same order in res[], then it becomes difficult to update

res[] without extra space. That is why res[] is maintained in reverse way, i.e., digits from

right to left are stored.

***multiply(res[], x)***

1) Initialize carry as 0.

2) Do following for i = 0 to res\_size – 1

….a) Find value of res[i] \* x + carry. Let this value be prod.

….b) Update res[i] by storing last digit of prod in it.

….c) Update carry by storing remaining digits in carry.

3) Put all digits of carry in res[] and increase res\_size by number of digits in carry.

**Example to show working of multiply(res[], x)**

A number 5189 is stored in res[] as following.

res[] = {9, 8, 1, 5}

x = 10

Initialize carry = 0;

i = 0, prod = res[0]\*x + carry = 9\*10 + 0 = 90.

res[0] = 0, carry = 9

i = 1, prod = res[1]\*x + carry = 8\*10 + 9 = 89

res[1] = 9, carry = 8

i = 2, prod = res[2]\*x + carry = 1\*10 + 8 = 18

res[2] = 8, carry = 1

i = 3, prod = res[3]\*x + carry = 5\*10 + 1 = 51

res[3] = 1, carry = 5

res[4] = carry = 5

res[] = {0, 9, 8, 1, 5}

Below is C++ implementation of above algorithm.

Output:

// C++ program to compute factorial of big numbers

#include<iostream>

**using namespace** std;

// Maximum number of digits in output

#define MAX 500

**int** multiply(**int** x, **int** res[], **int** res\_size)

// This function finds factorial of large numbers and prints them

**void** factorial(**int** n)

{

**int** res[MAX];

// Initialize result

res[0] = 1;

**int** res\_size = 1;

// Apply simple factorial formula n! = 1 \* 2 \* 3 \* 4...\*n

**for** (**int** x=2; x<=n; x++)

res\_size = multiply(x, res, res\_size);

cout << "Factorial of given number is \n";

**for** (**int** i=res\_size-1; i>=0; i--)

cout << res[i];

}

// This function multiplies x with the number represented by res[].

// res\_size is size of res[] or number of digits in the number represented

// by res[]. This function uses simple school mathematics for multiplication.

// This function may value of res\_size and returns the new value of res\_size

**int** multiply(**int** x, **int** res[], **int** res\_size)

{

**int** carry = 0; // Initialize carry

// One by one multiply n with individual digits of res[]

**for** (**int** i=0; i<res\_size; i++)

{

**int** prod = res[i] \* x + carry;

res[i] = prod % 10; // Store last digit of 'prod' in res[]

carry = prod/10; // Put rest in carry

}

// Put carry in res and increase result size

**while** (carry)

{

res[res\_size] = carry%10;

carry = carry/10;

res\_size++;

}

**return** res\_size;

}

// Driver program

**int** main()

{

factorial(100);

**return** 0;

}

Factorial of given number is

9332621544394415268169923885626670049071596826438162146859296389

5217599993229915608941463976156518286253697920827223758251185210

916864000000000000000000000000

The above approach can be optimized in many ways. We will soon be discussing

optimized solution for same.

This article is contributed by **Harshit Agrawal**. Please write comments if you find

anything incorrect, or you want to share more information about the topic discussed

above

56. Find length of period in decimal value of 1/n

Given a positive integer n, find the period in decimal value of 1/n. Period in decimal

value is number of digits (somewhere after decimal point) that keep repeating.

Examples:

Input: n = 3

Output: 1

The value of 1/3 is 0.**3**33333...

Input: n = 7

Output: 6

The value of 1/7 is 0.**142857**142857142857.....

Input: n = 210

Output: 6

The value of 1/210 is 0.00**476190**47619048.....

Let us first discuss a simpler problem of finding individual digits in value of 1/n.

**How to find individual digits in value of 1/n?**

Let us take an example to understand the process. For example for n = 7. The first digit

can be obtained by doing 10/7. Second digit can be obtained by 30/7 (3 is remainder in

previous division). Third digit can be obtained by 20/7 (2 is remainder of previous

division). So the idea is to get the first digit, then keep taking value of (remainder \* 10)/n

as next digit and keep updating remainder as (remainder \* 10) % 10. The complete

program is discussed here.

**How to find the period?**

The period of 1/n is equal to the period in sequence of remainders used in the above

process. This can be easily proved from the fact that digits are directly derived from

remainders.

One interesting fact about sequence of remainders is, all terns in period of this

remainder sequence are distinct. The reason for this is simple, if a remainder repeats,

then it’s beginning of new period.

The following is C++ implementation of above idea.

Output:

1

6

// C++ program to find length of period of 1/n

#include <iostream>

#include <map>

**using namespace** std;

// Function to find length of period in 1/n

**int** getPeriod(**int** n)

{

// Create a map to store mapping from remainder

// its position

map<**int**, **int**> mymap;

map<**int**, **int**>::iterator it;

// Initialize remainder and position of remainder

**int** rem = 1, i = 1;

// Keep finding remainders till a repeating remainder

// is found

**while** (**true**)

{

// Find next remainder

rem = (10\*rem) % n;

// If remainder exists in mymap, then the difference

// between current and previous position is length of

// period

it = mymap.find(rem);

**if** (it != mymap.end())

**return** (i - it->second);

// If doesn't exist, then add 'i' to mymap

mymap[rem] = i;

i++;

}

// This code should never be reached

**return** INT\_MAX;

}

// Driver program to test above function

**int** main()

{

cout << getPeriod(3) << endl;

cout << getPeriod(7) << endl;

**return** 0;

}

We can avoid the use of map or hash using the following fact. For a number n, there can

be at most n distinct remainders. Also, the period may not begin from the first remainder

as some initial remainders may be non-repetitive (not part of any period). So to make

sure that a remainder from a period is picked, start from the (n+1)th remainder and keep

looking for its next occurrence. The distance between (n+1)’th remainder and its next

occurrence is the length of the period.

Output:

1

6

**Reference:**

Algorithms And Programming: Problems And Solutions by Alexander Shen

This article is contributed by **Sachin**. Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above.

// C++ program to find length of period of 1/n without

// using map or hash

#include <iostream>

**using namespace** std;

// Function to find length of period in 1/n

**int** getPeriod(**int** n)

{

// Find the (n+1)th remainder after decimal point

// in value of 1/n

**int** rem = 1; // Initialize remainder

**for** (**int** i = 1; i <= n+1; i++)

rem = (10\*rem) % n;

// Store (n+1)th remainder

**int** d = rem;

// Count the number of remainders before next

// occurrence of (n+1)'th remainder 'd'

**int** count = 0;

**do** {

rem = (10\*rem) % n;

count++;

} **while**(rem != d);

**return** count;

}

// Driver program to test above function

**int** main()

{

cout << getPeriod(3) << endl;

cout << getPeriod(7) << endl;

**return** 0;

}

57. Greedy Algorithm for Egyptian Fraction

Every positive fraction can be represented as sum of unique unit fractions. A fraction is

unit fraction if numerator is 1 and denominator is a positive integer, for example 1/3 is a

unit fraction. Such a representation is called Egyptial Fraction as it was used by ancient

Egyptians.

Following are few examples:

Egyptian Fraction Representation of 2/3 is 1/2 + 1/6

Egyptian Fraction Representation of 6/14 is 1/3 + 1/11 + 1/231

Egyptian Fraction Representation of 12/13 is 1/2 + 1/3 + 1/12 + 1/156

We can generate Egyptian Fractions using Greedy Algorithm. For a given number of the

form ‘nr/dr’ where dr > nr, first find the greatest possible unit fraction, then recur for the

remaining part. For example, consider 6/14, we first find ceiling of 14/6, i.e., 3. So the

first unit fraction becomes 1/3, then recur for (6/14 – 1/3) i.e., 4/42.

Below is C++ implementation of above idea.

Output:

Egyptian Fraction Representation of 6/14 is

1/3 + 1/11 + 1/231

The Greedy algorithm works because a fraction is always reduced to a form where

// C++ program to print a fraction in Egyptian Form using Greedy

// Algorithm

#include <iostream>

**using namespace** std;

**void** printEgyptian(**int** nr, **int** dr)

{

// If either numerator or denominator is 0

**if** (dr == 0 || nr == 0)

**return**;

// If numerator divides denominator, then simple division

// makes the fraction in 1/n form

**if** (dr%nr == 0)

{

cout << "1/" << dr/nr;

**return**;

}

// If denominator divides numerator, then the given number

// is not fraction

**if** (nr%dr == 0)

{

cout << nr/dr;

**return**;

}

// If numerator is more than denominator

**if** (nr > dr)

{

cout << nr/dr << " + ";

printEgyptian(nr%dr, dr);

**return**;

}

// We reach here dr > nr and dr%nr is non-zero

// Find ceiling of dr/nr and print it as first

// fraction

**int** n = dr/nr + 1;

cout << "1/" << n << " + ";

// Recur for remaining part

printEgyptian(nr\*n-dr, dr\*n);

}

// Driver Program

**int** main()

{

**int** nr = 6, dr = 14;

cout << "Egyptian Fraction Representation of "

<< nr << "/" << dr << " is\n ";

printEgyptian(nr, dr);

**return** 0;

}

denominator is greater than numerator and numerator doesn’t divide denominator. For

such reduced forms, the highlighted recursive call is made for reduced numerator. So

the recursive calls keep on reducing the numerator till it reaches 1.

References:

http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fractions/egyptian.html

This article is contributed by **Shubham**. Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above.

58. Write an iterative O(Log y) function for pow(x, y)

Given an integer x and a positive number y, write a function that computes xy under

following conditions.

a) Time complexity of the function should be O(Log y)

b) Extra Space is O(1)

Examples:

Input: x = 3, y = 5

Output: 243

Input: x = 2, y = 5

Output: 32

**We strongly recommend to minimize your browser and try this yourself first.**

We have discussed recursive O(Log y) solution for power. The recursive solutions are

generally not preferred as they require space on call stack and they involve function call

overhead.

Following is C function to compute xy.

Output:

Power is 243

This article is contributed by **Udit Gupta**. Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above

#include <stdio.h>

/\* Iterative Function to calculate x raised to the power y in O(logy) \*/

**int** power(**int** x, unsigned **int** y)

{

// Initialize result

**int** res = 1;

**while** (y > 0)

{

// If y is even, simply do x square

**if** (y%2 == 0)

{

y = y/2;

x = x\*x;

}

// Else multiply x with result. Note that this else

// is always executed in the end when y becomes 1

**else**

{

y--;

res = res\*x;

}

}

**return** res;

}

// Driver program to test above functions

**int** main()

{

**int** x = 3;

unsigned **int** y = 5;

**printf**("Power is %d", power(x, y));

**return** 0;

}