BOOTHS ALGO

def conversion(a): #returns a binary string of a number which has bits equal to count

q=""

current\_n = len(a)

temp = count- current\_n

if (current\_n != count):

q = "0"\*temp + a

return q

def add(x,y): #fun to add two binary number strings

max\_len = max(len(x),len(y))

result = ''

carry = 0

for i in range(max\_len-1, -1, -1):

r = carry

if x[i] == '1':

r += 1

if y[i] == '1':

r += 1

if r % 2 == 1:

result = "1" + result

else:

result = "0" + result

if r<2:

carry =0

else:

carry= 1

return result

def twoc(a):

l = list(a)

for i in range(len(l)):

if l[i] == "1" :

l[i] = "0"

else: l[i] ="1"

b = "0"\*(len(l)-1) + "1"

return add("".join(l),b)

def right\_shift(ac,q,q1):

a = ac[0]

for i in range(1,len(ac)):

a+=ac[i-1]

b = ac[-1]

for j in range(1,len(q)):

b+=q[j-1]

c = q[-1]

return a,b,c

#Taking input

x = int(input("enter x:"))

y = int(input("enter y:")) #taking x and y decimal numbers as input for x \* y

a = bin(x).replace("0b", "")

b = bin(y).replace("0b", "")

negative\_a=0

negative\_b=0

if (a[0]=="-"):

a = a.replace("-","")

negative\_a =1

if (b[0]=="-"):

b = b.replace("-","")

negative\_b =1

print("binary value of a:"+ a)

print("binary value of b:"+ b)

if (len(a)>len(b)):

count = len(a) + 1

else:

count = len(b) + 1

count1 = count

firstP = conversion(a) #contains the positive representation of the multiplicand

secondP = conversion(b) #contains the positive representation of the multiplier

firstN = twoc(firstP) #contains 2's complement of the multiplicand

secondN = twoc(secondP) #contains 2's complement of the multiplier

#BOOTH ALGO IMPLEMENTATION

if negative\_a ==0:

M = firstP #M is the multiplicand and M2 contains its 2's complement

M2 = firstN

else:

M = firstN

M2 = firstP

if negative\_b ==0:

Q = secondP #Q is the multiplier

else:

Q = secondN

print("M:",M)

print("-M:",M2)

print("Q:",Q)

AC = conversion("0")

Q1 = "0" # one bit for comparision

print("The table for the booth's algorithm is as follow:")

print("\n")

print("Count" +"\t" + "AC" +"\t" + "Q" +"\t" + "Q1" +"\t" + "Operation")

print("\n")

print(str(count) +"\t" + AC +"\t"+ Q +"\t" + Q1 +"\t"+ "initial")

print("\n")

while (count>0):

compare = Q[-1] + Q1

if compare[0]==compare[-1]:

AC, Q, Q1 =right\_shift(AC,Q,Q1)

Op = "right shift"

elif compare =="10":

AC = add(AC,M2)

AC, Q, Q1 =right\_shift(AC,Q,Q1)

Op = "AC=AC-M and right shift"

elif compare == "01":

AC = add(AC,M)

AC, Q, Q1 =right\_shift(AC,Q,Q1)

Op = "AC=AC+M and right shift"

print(str(count) +"\t"+ AC +"\t" + Q +"\t" + Q1 +"\t" + Op)

print("\n")

count = count-1

answer = AC+Q

if negative\_a==negative\_b:

ans\_d = str(int(answer,2))

else:

ans\_d = "-" + str(int(twoc(answer),2))

print("The product in binary is:" + answer)

print("Decimal conversion:" + ans\_d)

NON-Restoring ALGO

def shift\_left(s):

s = s[1:]

s = s + "0"

return s

def complement(s):

d = {'0': '1', '1': '0'}

e = ''.join(d[x] for x in s)

l = len(e)

sum = bin(int(e, 2) + int("1", 2))

sum = sum[2:]

return sum.zfill(l)

def add\_zero\_to\_string(x):

x = list(x)

x[-1] = "0"

return "".join(x)

def add\_one\_to\_string(x):

x = list(x)

x[-1] = "1"

return "".join(x)

q = int(input("Enter the dividend: "))

m = int(input("Enter the divisor: "))

Q = bin(q)[2:] # final Q

M = bin(m)[2:] # final M

if len(M) > len(Q):

Q = Q.zfill(len(M))

else:

M = M.zfill(len(Q))

print(f"Initial Values: A: {'0' \* len(Q)}, Q: {Q}, M: {M}\n")

l = len(Q)

count = l

Mc = complement(M)

a = "0"

for i in range(l - 1):

a = a + "0"

print("Steps of Division:")

while count > 0:

s = a + Q

value = shift\_left(s)

a = value[0:l]

Q = value[l:]

a = bin(int(a, 2) + int(Mc, 2))

if len(a[2:]) == l + 1:

a = a[3:]

else:

a = a[2:]

if a[0] == "1":

a = bin(int(a, 2) + int(M, 2))

if len(a[2:]) == l + 1:

a = a[3:]

else:

a = a[2:]

Q = add\_zero\_to\_string(Q)

print(f"Step {l - count}: Left Shift and Subtract: A: {a}")

print(f"A: {a} Q: {Q} Unsuccessful")

else:

Q = add\_one\_to\_string(Q)

print(f"Step {l - count}: Left Shift and Subtract: A: {a}")

print(f"A: {a} Q: {Q} Successful")

count -= 1

print(f"A: {a} Q: {Q} No Restoration\n")

print("\nFinal Results:")

# print("Remainder in Binary Form :", a)

print("Remainder in Decimal Form :", int(a, 2))

# print("Quotient in Binary Form :", Q)

print("Quotient in Decimal Form :", int(Q, 2))

RESTORING

def left\_shift(accumulator, quotient):

return accumulator[1:] + quotient[0], quotient[1:] + "0"

def right\_shift(accumulator, quotient):

return "0" + accumulator[:-1], accumulator[-1] + quotient[:-1]

def add\_binary(x, y):

max\_len = max(len(x), len(y))

x = x.zfill(max\_len)

y = y.zfill(max\_len)

result = ''

carry = 0

for i in range(max\_len - 1, -1, -1):

r = carry

r += 1 if x[i] == '1' else 0

r += 1 if y[i] == '1' else 0

result = ('1' if r % 2 == 1 else '0') + result

carry = 0 if r < 2 else 1

if carry != 0:

result = '1' + result

return result.zfill(max\_len)

def complement(x):

return ''.join('1' if bit == '0' else '0' for bit in x)

def restore\_division(dividend, divisor):

if len(dividend) < len(divisor):

raise ValueError("Dividend must be greater than or equal to the divisor")

dividend = dividend.lstrip('0') # Remove leading zeros

divisor = divisor.lstrip('0') # Remove leading zeros

accumulator = "0" \* len(divisor)

quotient = dividend[:len(divisor)]

steps = []

for i in range(len(dividend) - len(divisor) + 1):

if accumulator[0] == "1":

accumulator = add\_binary(accumulator, divisor)

steps.append(f"Step {i}: Subtract and Shift: A: {accumulator}, Q: {quotient}, 1")

else:

accumulator = add\_binary(accumulator, complement(divisor))

steps.append(f"Step {i}: Add and Shift: A: {accumulator}, Q: {quotient}, 0")

accumulator, quotient = left\_shift(accumulator, quotient)

return steps

def main():

dividend = input("Enter the dividend (binary): ")

divisor = input("Enter the divisor (binary): ")

steps = restore\_division(dividend, divisor)

print("\nRestoring Division Steps:")

for step in steps:

print(step)

if \_\_name\_\_ == "\_\_main\_\_":

main()

FIFO,LRU,OPTIMAL

def fifo(pages, capacity):

memory = []

page\_faults = 0

replacements = []

for page in pages:

if page not in memory:

if len(memory) == capacity:

memory.pop(0)

memory.append(page)

page\_faults += 1

replacements.append("Miss")

else:

replacements.append("Hit")

return page\_faults, replacements

def lru(pages, capacity):

memory = []

page\_faults = 0

replacements = []

for page in pages:

if page in memory:

memory.remove(page)

else:

if len(memory) == capacity:

memory.pop(0)

page\_faults += 1

replacements.append("Miss")

memory.append(page)

replacements.append("Hit")

return page\_faults, replacements

def optimal(pages, capacity):

memory = []

page\_faults = 0

replacements = []

for page in pages:

if page not in memory:

if len(memory) == capacity:

future\_occurrences = {}

for p in memory:

try:

future\_occurrences[p] = pages.index(p, pages.index(page) + 1)

except ValueError:

future\_occurrences[p] = float('inf')

page\_to\_remove = max(future\_occurrences, key=future\_occurrences.get)

memory.remove(page\_to\_remove)

memory.append(page)

page\_faults += 1

replacements.append("Miss")

else:

replacements.append("Hit")

return page\_faults, replacements

# Test the algorithms

pages = [7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2]

capacity = 3

print("FIFO:")

page\_faults\_fifo, replacements\_fifo = fifo(pages, capacity)

print("Page Faults:", page\_faults\_fifo)

print("Replacements:", replacements\_fifo)

print("\nLRU:")

page\_faults\_lru, replacements\_lru = lru(pages, capacity)

print("Page Faults:", page\_faults\_lru)

print("Replacements:", replacements\_lru)

print("\nOptimal:")

page\_faults\_optimal, replacements\_optimal = optimal(pages, capacity)

print("Page Faults:", page\_faults\_optimal)

print("Replacements:", replacements\_optimal)

ADD and SUB

Mov ax,[1000h]

mov bx,[1002h]

mov cl,00h

add ax,bx

mov [1004h],ax

jnc jump

inc cl

jump:

mov [1006h],cl

hlt

ASCENDING DESC

mov si,1100h

mov cl,[si]

dec cl

repeat:

mov si,1100h

mov ch,[si]

dec ch

inc si

repcom:

mov al,[si]

inc si

cmp al,[si]

jc ahead

xchg al,[si]

xchg al,[si-1]

ahead:

dec ch

jnz repcom

dec cl

jnz repeat

hlt

BLOCK to another block

data segment

seg1 db 1h,2h,3h

ends

extra segment

seg2 db ?

ends

code segment

start:

mov ax,data

mov ds,ax

mov ax,extra

mov es,ax

lea si,seg1

lea di,seg2

mov cx,03h

x:mov ah,ds:[si]

mov es:[di],ah

inc si

inc di

dec cx

jnz x

int 3

ends

end start

MIN MAX

data segment

arr db 5,3,7,1,9,6,8,4

len dw $-arr

min db ?

max db ?

data ends

code segment

assume:ds:data cs:code

start:

mov ax,data

mov ds,ax

lea si,arr

mov al,arr[si]

mov min,al

mov max,al

mov cx,len

repeat:

mov al,arr[si]

cmp min, al

jl checkmax

mov min,al

checkmax:

cmp max,al

jg done

mov max,al

done:

inc si

loop repeat

mov ah,4ch

int 21h

code ends

end start

FACTORIAL

fact macro F

up:

mul F

dec F

jnz up

endm

data segment

num dw 05h

result dw ?

ends

stack segment

dw 128 dup<0>

ends

code segment

start:

mov ax,data

mov ds,ax

mov cx,num

mov ax,0001h

fact num

mov result,ax

ends

INTERRUPTS

data segment

msg db "Enter a character$"

data ends

code segment

assume cs:code,ds:data

start:

mov ax,data

mov ds,ax

lea dx,msg

mov ah,09h

int 21h

mov ah,01

int 21h

mov dl,al

mov ah,01

int 21h

mov ah,4ch

int 21h

code ends

end star