CS 101 Computer Programming and Utilization

Practice Problems

Param Rathour
https://paramrathour.github.io/CS101

Autumn Semester 2021-22

Last update: 2022-02-17 13:42:16+05:30

Disclaimer

These are **optional** problems. As these problems are pretty involving, my advice to you would be to first solve exercises given in slides, lab optional questions and get comfortable with the course content. The taught methods will suffice to solve these problems. (You are free to use 'other' stuff but not recommended)

Good Programming Practices

- Clearly writing documentation explaining what the program does, how to use it, what quantities it takes as input, and what quantities it returns as output.
- Using appropriate variable/function names.
- Extensive internal comments explaining how the program works.
- Complete error handling with informative error messages.
  For example, if $a = b = 0$, then the $\gcd(a, b)$ routine should return the error message "$\gcd(0, 0)$ is undefined" instead of going into an infinite loop or returning a "division by zero" error.

Contents

§0. Practice Problems 0 - Introduction

0.1. Hilbert Curve

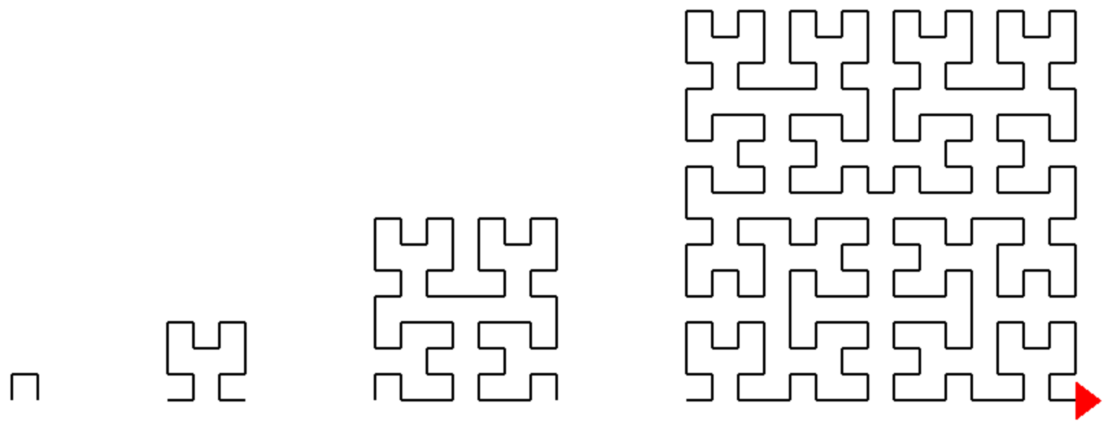


Figure 1: Hilbert Curve

Problem Statement:
Take an integer as input and draw the corresponding iteration of this fractal using turtleSim
You may think along these lines

Step 1 Find a simple pattern in these iterations

Step 2 Think how can you implement this pattern in an efficient way (here think in the number of lines of code you have to write. **Word of caution**: this is just one of the possible definitions of efficient code)

Step 3 Do you think that you need something that will implement/shorten your code?
How will it look like? (it's a feature)

Feel free to discuss your thoughts on this.

Note. *For people comfortable with the basics of C++, this shouldn't be difficult. You may try this*

Fun Videos

Hilbert's Curve: Is infinite math useful?
Recursive PowerPoint Presentations [Gone Fractal!]

Book Chapters for Graphics

Additional chapters of the book on Simplecpp graphics demonstrating its power
(It is just a list, you are not expected to understand/study things, CS101 is for a reason :P)

Chapter 1 Turtle graphics

Chapter 5 Coordinate based graphics, shapes besides turtles

Chapter 15.2.3 Polygons

Chapter 19 Gravitational simulation

Chapter 20 Events, Frames, Snake game

Chapter 24.2 Layout of math formulae

Chapter 26 Composite class

Chapter 28 Airport simulation

§1. Practice Problems 1 - Warmup

1.1. Harshad number

A positive integer is a Harshad Number if it's divisible by sum of its digits when written in some base n.

Problem Statement: For simplicity, we take $n = 10$
For each of n given integers $(k_1, k_2, \ldots, k_n)$, output "yes" if it is a Harshad Number else output "no".

Input Format
n (number of integers)
$k_1, k_2, \ldots, k_n$
Output Format
Output (each on a newline)
Sample Input
```
8
1 7 15 24 196 1729 8529671 12751220
```
Sample Output
```
yes
yes
no
yes
no
yes
no
yes
```

1.2. Palindromic number

A non-negative integer is a Palindromic number if it remains the same when it's digits are reversed.

Problem Statement: For simplicity, we take $n = 10$
For each of n given integers $(k_1, k_2, \ldots, k_n)$, output "yes" if it is a Palindromic Number else output "no".

Input Format
n (number of integers)
$k_1, k_2, \ldots, k_n$
Output Format
Output (each on a newline)
Sample Input
```
12
1 7 15 22 196 666 1212 96096 8801088 9256713 40040004 123454321
```
Sample Output
```
yes
yes
no
yes
no
yes
no
no
yes
no
no
yes
```

1.3. Base Conversion

(a) **Problem Statement:**
Convert n positive binary numbers $(k_1, k_2, \ldots, k_n)$ to decimal.

Input Format
n
$k_1, k_2, \ldots, k_n$
Output Format
Output truncated till 8 decimal places (each on a newline)
Sample Input
```
9
1 111 110001 101010111 100101100001 1.00011001 11.001001 110.01 10110.01110101
```
Sample Output
```
1
7
49
343
2401
1.09765625
3.140625
6.25
22.45703125
```

(b) **Problem Statement:**
Convert n positive decimal numbers $(k_1, k_2, \ldots, k_n)$ to binary.

Input Format
n
$k_1, k_2, \ldots, k_n$
Output Format
Output truncated till 8 decimal places (each on a newline)
Sample Input
```
9
1 7 49 343 2401 1.1 3.1415 6.25 22.459
```
Sample Output
```
1
111
110001
101010111
100101100001
1.00011001
11.00100100
110.01
10110.01110101
```

§2. Practice Problems 2 - Loops

2.1. Pisano Period

You probably heard about Fibonacci Numbers!

The Fibonacci numbers are the numbers in the integer sequence: (defined by the recurrence relation)

$$\begin{aligned} &F_0 = 0 \\ &F_1 = 1 \\ &F_n = F_{n-1} + F_{n-2} \quad n \in \mathbb{Z} \quad \text{(They can be extended to negative numbers)} \end{aligned} \tag{1}$$

For any integer n, the sequence of Fibonacci numbers F_i taken modulo n is periodic.

The Pisano period, denoted $\pi(n)$, is the length of the period of this sequence.

For example, the sequence of Fibonacci numbers modulo 3 begins:

$$0, 1, 1, 2, 0, 2, 2, 1, 0, 1, 1, 2, 0, 2, 2, 1, 0, 1, 1, 2, 0, 2, 2, 1, 0, \ldots \text{(A082115)}$$

This sequence has period 8, so $\pi(3) = 8$. (Basically, the remainder when these numbers are divided by n is a repeating sequence. You have to find the length of sequence)

Problem Statement:

(a) Find Pisano period of n numbers $k_1, k_2, \ldots, k_n$

Input Format
n
$k_1, k_2, \ldots, k_n$
Output Format
$\pi(k_i)$ (each on a newline)
Sample Input
3
3 10 25
Sample Output
8
60
100

(b) For n numbers $k_1, k_2, \ldots, k_n$, find $\max(\pi(i))$ for $i = 1, 2, \ldots, k$ and corresponding i
If there are 2 (or more) such i's, output smallest of them

Input Format
n
$k_1, k_2, \ldots, k_n$
Output Format
k $\pi(k)$ (each pair on a newline)
Here k is smallest possible integer satisfying $\pi(k) = \max(\pi(i))$ for possible i
Sample Input
5
20 40 60 80 100
Sample Output
10 60
30 120
50 300
50 300
98 336

2.2. π

$$\frac{\pi}{2} = \sum_{k=0}^{\infty} \frac{k!}{(2k+1)!!} = \sum_{k=0}^{\infty} \frac{2^k k!^2}{(2k+1)!} \tag{2}$$

Note. *$n!!$ is called double factorial. $n!! \neq (n!)!$.*

Problem Statement:
Calculate π using first $k_i + 1$ terms [1] of Equation (2) for n different natural numbers $k_1, k_2, \dots, k_n$.
Give your answers correct to 10 decimal places

Input Format
n
$k_1, k_2, \dots, k_n$
Output Format
Calculated π for k_i (each on a newline)
Sample Input
```
3
10 20 30
```
Sample Output
```
3.1411060216
3.1415922987
3.1415926533
```

Fun Video

7 Factorials You Probably Didn't Know

2.3. Simpson's Rule

A method for numerical integration

$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{3}\left(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \cdots + 4f(x_{n-1}) + f(x_n)\right) \tag{3}$$

Note. *Simpson's rule can only be applied when an odd number of ordinates is chosen.*

Problem Statement:
Solve Equation (2.3) giving the answers correct to 7 decimal places (Use 101 ordinates)

$$\int_{0.5}^{1} \frac{\sin\theta}{\theta}\,\mathrm{d}\theta$$

Correct Answer $=$ 0.4529756

2.4. Special Pythagorean Triplet

A Pythagorean Triplet is a set of positive integers a, b, c $(a \leq b \leq c)$ such that

$$a^2 + b^2 = c^2 \tag{4}$$

Problem Statement:
There exists exactly one Pythagorean triplet for which $a + b + c = 1000$. Find abc

Correct Answer $=$ 31875000

[1] i.e calculate π till $\frac{k_i!}{(2k_i+1)!!}$ term

2.5. Modular Exponentiation

Consider the problem of calculating $x^y \pmod{k}$ (i.e. the remainder when x^y is divided by k).

A naive approach is to keep multiplying by x (and take $\pmod{k}$) until we reach x^y.[2]

$$x \pmod{k} \to x^2 \pmod{k} \to x^3 \pmod{k} \to x^4 \pmod{k} \to \cdots \to x^y \pmod{k}$$

We can use a much faster method which involves *repeated squaring* of $x \pmod{k}$

$$x \pmod{k} \to x^2 \pmod{k} \to x^4 \pmod{k} \to x^8 \pmod{k} \to \cdots \to x^{2^{\lfloor \log y \rfloor}} \pmod{k} \tag{5}$$

The idea is to multiply some of the above numbers and get $x^y \pmod{k}$.

This is achieved by choosing all powers that have 1 in binary representation of y.
For example,

$$x^{25} = x^{11001_2} = x^{10000_2} \cdot x^{1000_2} \cdot x^{1_2} = x^{16} \cdot x^8 \cdot x^1$$

Which gives,

$$x^{25} \pmod{k} = ((x^{16} \pmod{k}) \cdot (x^8 \pmod{k}) \cdot (x^1 \pmod{k})) \pmod{k}$$

(a) **Problem Statement:**
Calculate $x^y \pmod{k}$ using the above method for n (x, y, k) triples. Take $k = 10^9 + 7$. why this number?

Input Format
n
$x_1, y_1, \quad x_2, y_2, \ldots, x_n, y_n$
Output Format
$x^y \pmod{k}$ for x_i, y_i (each on a newline)
Sample Input

```
5
3 4   2 8   123 123   129612095 411099530   241615980 487174929
```

Sample Output

```
81
256
921450052
276067146
838400234
```

Note. *Before proceeding to next task, verify your program on more testcases from here.*

(b) **Problem Statement:**
Calculate $x^{y^z} \pmod{k}$ using the above method for n (x, y, z, k) quadruples. Take $k = 10^9 + 7$.

Input Format
n
$x_1, y_1, z_1, \quad x_2, y_2, z_2, \ldots, x_n, y_n, z_n$
Output Format
$x^{y^z} \pmod{k}$ for x_i, y_i, z_i (each on a newline)
Sample Input

```
5
3 7 1   15 2 2   3 4 5   427077162 725488735 969284582   690776228 346821890 923815306
```

Sample Output

```
2187
50625
763327764
464425025
534369328
```

Note. *Verify your program on more testcases from here.*

[2]this works because $(a \cdot b) \pmod{m} = ((a \pmod{m}) \cdot (b \pmod{m})) \pmod{m}$

§3. Practice Problems 3 - Functions

3.1. Collatz Conjecture

Consider the following operation on an arbitrary positive integer:

- If the number is even, divide it by two.
- If the number is odd, triple it and add one.

This operation can be defined using the function f as follows:

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{cases} \tag{6}$$

We can form a sequence $\{a_i\}$ by applying f repeatedly i.e., begin with given n and take $f(n)$ as the next input to f

$$a_i = \begin{cases} f(a_{i-1}) & i > 0 \\ n & i = 0 \end{cases} \tag{7}$$

Collatz conjecture states that this procedure will eventually reach 1, for every positive integer.

(a) **Problem Statement:**
Your task is print all term of this sequence till 1, given input $a_0 = n$ $(n < 10^6)$[3].

Input Format
n
Output Format
a_i's (Elements of $\{a_i\}$ starting from n till 1)
Sample Input
```
27
```
Sample Output
```
27 82 41 124 62 31 94 47 142 71 214 107 322 161 484 242 121 364 182 91 274 137 412 206 103 310 155
466 233 700 350 175 526 263 790 395 1186 593 1780 890 445 1336 668 334 167 502 251 754 377 1132
566 283 850 425 1276 638 319 958 479 1438 719 2158 1079 3238 1619 4858 2429 7288 3644 1822 911
2734 1367 4102 2051 6154 3077 9232 4616 2308 1154 577 1732 866 433 1300 650 325 976 488 244 122
61 184 92 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
```

Note. *Before proceeding to next task, verify your program on more testcases from here.*

(b) **Problem Statement:**
Your task is to return the number of operations required to reach 1 for arbitrary number of inputs.

Input Format
Arbitrary number of testcases (each space separated)   Stop when input is negative
Output Format
Count of operations for each number (each on a newline)
Sample Input
```
1 3 7 9 27 871 77031 -1
```
Sample Output
```
0
7
16
19
111
178
350
```

[3] As of 2020, the conjecture has been checked by computer for all starting values up to $2^{68} \approx 2.95 \times 10^{20}$, so sequence from n will reach 1

3.2. Horner's method

An algorithm for polynomial evaluation

$$\begin{aligned} \mathrm{P}(x) &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n \\ &= a_0 + x\Big(a_1 + x\Big(a_2 + x\big(a_3 + \cdots + x(a_{n-1} + x\,a_n)\cdots\big)\Big)\Big) \end{aligned} \tag{8}$$

This allows the evaluation of a polynomial of degree n with only n multiplications and n additions. This is optimal, since there are polynomials of degree n that cannot be evaluated with fewer arithmetic operations

Problem Statement:
Write a function Horner() takes inputs degree of input x, polynomial n & coefficients of P,and returns $\mathrm{P}(x)$
For k testcases with polynomials $P_1, P_2, \ldots, P_n$, degree $n_1, n_2, \ldots, n_k$ and variable $x_1, x_2, \ldots, x_n$ find $P_i(x_i)$ for each i in $\{1, 2, \ldots, n\}$

Input Format
k (number of testcases)
x, n_i (degree) and P_i ($n_i + 1$ integral coefficients in order $(a_n, a_{n-1}, \ldots, a_1, a_0)$ i.e. i^{th} index is $(n-i)^{\text{th}}$ power)
Output Format
$\mathrm{Horner}()$ (each result space seperated)
Sample Input

```
5
1   0   1
2   1   2 -3
2   2   7 -8 15
3   3   4 -3 -1 2
5   6   27 9 48 47 19 10 21
```

Sample Output

```
1   1   27   80   486421
```

3.3. Euler's Totient Function

Euler's totient function ($\varphi(n)$) is the number of positive integers $\leq n$ that are co-prime to n.
Problem Statement:
Define a function that calculates $\varphi(n)$ for given n

Input Format
n
$k_1, k_2, \ldots, k_n$
Output Format
Output $\varphi(k_i)$ (each on a newline)
Sample Input

```
10
1 4 8 20 44 69 97 120 2520 12345678910
```

Sample Output

```
1
2
4
8
20
44
96
32
576
4938271560
```

3.4. Farey Sequence

This sequence has all rational numbers in range $[0/1 \text{ to } 1/1]$ sorted *in increasing order* such that the denominators are less than or equal to n and all numbers are in *reduced forms* i.e., 2/4 does not belong to this sequence as it can be reduced to 1/2.

Problem Statement:
Define a function that generates the Farey Sequence for corresponding n

Input Format
n
Output Format
Corresponding numbers in sequence in p/q format
Sample Input
7
Sample Output
0/1 1/7 1/6 1/5 1/4 2/7 1/3 2/5 3/7 1/2 4/7 3/5 2/3 5/7 3/4 4/5 5/6 6/7 1/1

Can you find efficient solution?

Fun Video

Funny Fractions and Ford Circles

3.5. Calendar

Write a function that calculates the day of the week for any particular date in the past or future.
Consider Gregorian calendar (AD)

Task 1: As a programming exercise, try the naive approach:
Starting from 1 Jan 0001 (Saturday) and calculate day after day till you reach the given date
Use `switch-case` statement

Task 2: Try to make more efficent algorithm (reduce completion time) than Task 1
Implement it, and discuss your approach with me.

Also check for invalid dates (Write another function for this). If dates are invalid, output -**1**

Input Format
n
Followed by n dates in **Date Month Year** format
Output Format
Day of the Week
Sample Input
5
19 2 1627
29 2 1700
15 4 1707
22 12 1887
23 6 1912
Sample Output
Monday
-1
Friday
Thursday
Sunday

Note. *Use your Task 1 program to check Task 2 implementation.*

Fun Video

The Doomsday Algorithm

§4. Practice Problems 4 - Recursion

Practice Problem 4 are inspired from the following video do watch till 6:10 to get clear understanding of recursion 5 Simple Steps for Solving Any Recursive Problem

- What's the simplest possible input?
- Play around with examples and visualize!
- Relate hard cases to simpler cases
- Generalize the pattern
- Write code by combining recursive pattern with base case

4.1. Ackermann function

Write a recursive function $\mathrm{A}()$ that takes two inputs n and m and outputs the number $\mathrm{A}(m,n)$ where $\mathrm{A}(m,n)$ is defined as

$$\mathrm{A}(0,n) = n+1 \tag{9}$$
$$\mathrm{A}(m+1,0) = \mathrm{A}(m,1) \tag{10}$$
$$\mathrm{A}(m+1,n+1) = \mathrm{A}(m,\mathrm{A}(m+1,n)) \tag{11}$$

Problem Statement:
For k pairs $(m_1,n_1),(m_2,n_2),\ldots,(m_k,n_k)$, find $\mathrm{A}(m,n)$ for such m,n

Input Format
k
m_i n_i (each pair on a newline)
Output Format
$\mathrm{A}(m_i,n_i)$ (each result on a newline)
Sample Input
```
7
0
0 4
1 3
2 2
3 4
4 0
4 1
```
Sample Output
```
1
5
5
7
125
13
65533
```

4.2. Hereditary Representation

Here, a natural number n_b (in base b) is represented only using symbols 0 to $(b-1)$ as exponents of b.

To generate this representation, find the usual base representation of the number and then represent its exponents also in the usual base representation. Keep repeating this until there is no exponenet $> b$. For example,

$$\begin{aligned} 666_2 &= 2^1 + 2^3 + 2^4 + 2^7 + 2^9 \\ &= 2^1 + 2^{2^0+2^1} + 2^{2^2} + 2^{2^0+2^1+2^2} + 2^{2^0+2^3} \\ &= 2^1 + 2^{2^0+2^1} + 2^{2^{2^1}} + 2^{2^0+2^1+2^{2^1}} + 2^{2^0+2^{2^0+2^1}} \end{aligned} \tag{12}$$

Here are some more examples to get familiar,

$$\begin{aligned} 10_2 &= 2^1 + 2^{2^0+2^1} \\ 100_2 &= 2^{2^1} + 2^{2^0+2^{2^1}} + 2^{2^1+2^{2^1}} \\ 3435_3 &= 2\cdot 3^1 + 3^{3^1} + 2\cdot 3^{2\cdot 3^0+3^1} + 3^{2\cdot 3^1} + 3^{3^0+2\cdot 3^1} \end{aligned}$$

Problem Statement:
Write a Recursive Function Hereditary() that takes input the natural number n and base b (≥ 2) and writes its Hereditary Representation using the below conventions:

The general base representation is $n_b = a_0 \cdot b^0 + a_1 \cdot b^1 + \cdots$ where a_i's $\in 0, 1, \ldots, b-1$

- The powers of base representation are in increasing order (first b^0 then b^1 then b^2 and so on)
- Don't simplify the exponents when it becomes between 0 and $b-1$. So, b is represented as `b^{1}` and not as `b^{b^{0}}`.
- Only powers with non-zero coefficients are displayed
- Only coefficients > 1 are displayed
- Use `+`, `*` to denote addition (add space between operands), multiplication respectively and `b^{y}` for b^y

Input Format
k
n_i b_i (each pair on a newline)
Output Format
Hereditary Representation (each result on a newline)
Sample Input
```
8
2 2
10 2
100 2
666 2
3435 3
1234321 2
1234321 3
1234321 5
```
Sample Output
```
2^{1}

2^{1} + 2^{2^{0} + 2^{1}}

2^{2^{1}} + 2^{2^{0} + 2^{2^{1}}} + 2^{2^{1} + 2^{2^{1}}}

2^{1} + 2^{2^{0} + 2^{1}} + 2^{2^{2^{1}}} + 2^{2^{0} + 2^{1} + 2^{2^{1}}} + 2^{2^{0} + 2^{2^{0} + 2^{1}}}

2*3^{1} + 3^{3^{1}} + 2*3^{2*3^{0} + 3^{1}} + 3^{2*3^{1}} + 3^{3^{0} + 2*3^{1}}

2^{0} + 2^{2^{2^{1}}} + 2^{2^{0} + 2^{1} + 2^{2^{1}}} + 2^{2^{2^{0} + 2^{1}}} + 2^{2^{1} + 2^{2^{0} + 2^{1}}}
+ 2^{2^{2^{1}} + 2^{2^{0} + 2^{1}}} + 2^{2^{1} + 2^{2^{1}} + 2^{2^{0} + 2^{1}}} + 2^{2^{0} + 2^{1} + 2^{2^{1}}
+ 2^{2^{0} + 2^{1}}} + 2^{2^{0} + 2^{2^{2^{1}}}} + 2^{2^{2^{1}} + 2^{2^{2^{1}}}}

3^{0} + 2*3^{1} + 3^{2} + 3^{3^{1}} + 3^{3^{0} + 3^{1}} + 3^{2*3^{1}} + 2*3^{2*3^{0} + 2*3^{1}} + 2*3^{3^{2}}
+ 2*3^{3^{0} + 3^{2}} + 2*3^{3^{1} + 3^{2}}

5^{0} + 4*5^{1} + 2*5^{2} + 4*5^{3} + 4*5^{4} + 4*5^{5^{1}} + 3*5^{5^{0} + 5^{1}} + 3*5^{3*5^{0} + 5^{1}}
```

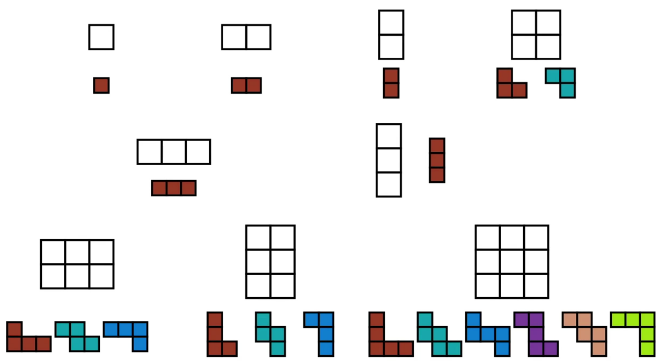
Fun Videos

Solve the problem Kill the Mathematical Hydra using Hereditary Representation How Infinity Explains the Finite | Infinite Series

4.3. GridPaths

Write a recursive function $\mathrm{NumberOfGridPaths}()$ that takes two inputs n and m and outputs the number of unique paths from the top left corner to bottom right corner of a $n \times m$ grid.

Constraints: $n, m \geq 1$ and you can only move down or right 1 unit at a time.
Examples given in Figure 2a
Problem Statement:


(a) Number of Grid Paths for $m, n \in \{1, 2, 3\}$

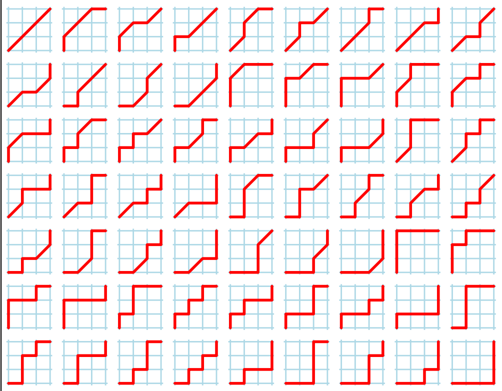

(b) DelannoyNumber(3,3) = 63

Figure 2

For k pairs $(n_1, m_1), (n_2, m_2), \ldots, (n_k, m_k)$, find $\mathrm{NumberOfGridPaths}(n, m)$ for such n, m

Input Format
k
n_i m_i (each pair on a newline)
Output Format
$\mathrm{NumberOfGridPaths}(n_i, m_i)$ (each result on a newline)
Sample Input

```
6
1 1
2 5
3 3
6 3
7 10
17 8
```

Sample Output

```
1
5
6
21
5005
245157
```

Can you find efficient solution?

4.4. Delannoy Numbers

Delannoy numbers describes the number of paths from the southwest corner (0, 0) of a rectangular grid to the northeast corner (m, n), using only single steps north, northeast, or east.
Write a recursive function $\text{DelannoyNumber}()$ to count number of these paths Constraints: $n, m \geq 0$
Examples given in Figure 2b
Problem Statement:
For k pairs $(n_1, m_1), (n_2, m_2), \ldots, (n_k, m_k)$, find $\text{DelannoyNumbers}(n, m)$ for such n, m

Input Format
k
n_i m_i (each pair seperated by space)
Output Format
$\text{DelannoyNumber}(n_i, m_i)$ (each result on a newline)
Sample Input
```
6
1 1   2 5   3 3   6 3   7 10   17 8
```
Sample Output
```
3
61
63
377
433905
62390545
```
Can you find efficient solution?

4.5. Thue-Morse Sequence aka Fair Share Sequence

Thue-Morse Sequence is the infinite binary sequence obtained by starting with 0 and successively appending the Boolean complement of the sequence obtained thus far (called prefixes of the sequence).
First few steps :

- Start with 0
- Append complement of 0, we get 01
- Append complement of 01, we get 0110
- Append complement of 0110, we get 01101001

Problem Statement:
Consider appending complement of a prefix to itself as one iteration
Define a function to take a positive integer n as input then iterate n times to print the first 2^n digits

Input Format
n
Output Format
Corresponding digits in sequence
Sample Input
```
6
```
Sample Output
```
0110100110010110100101100110100110010110011010010110100110010110
```
Again, can you find better solution?

Fun Video
The Fairest Sharing Sequence Ever

4.6. Partitions

(a) Write a recursive function $\mathrm{NumberOfPartitions}()$ that counts the number of ways you can partition n objects using parts up to m

Constraints: $n, m > 0$
Examples given in Figure 3

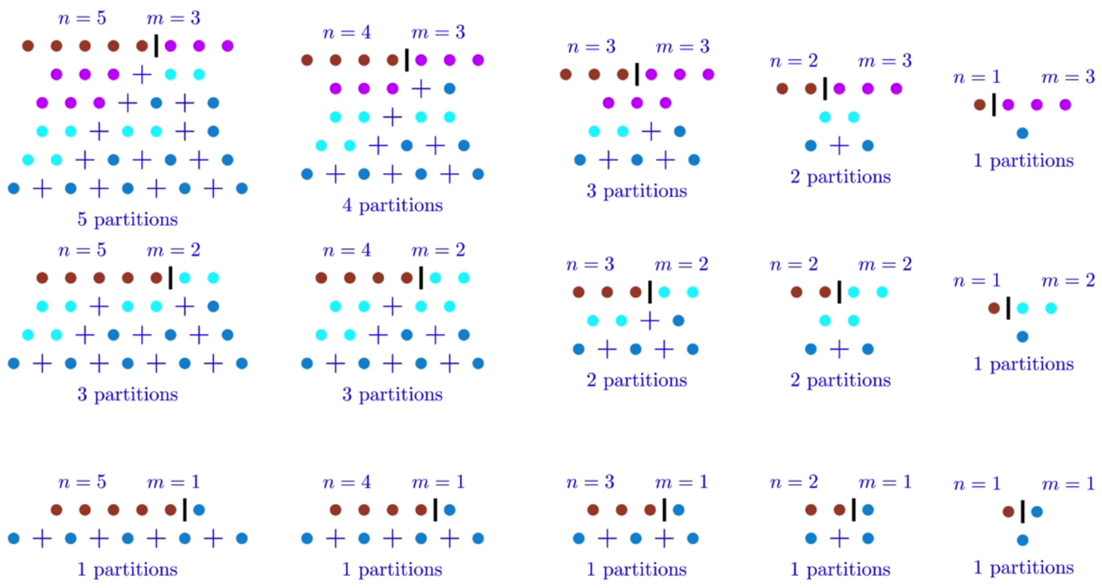


Figure 3: Partition n objects using parts up to m

Can you find efficient solution?
Problem Statement:
For k pairs $(n_1, m_1), (n_2, m_2), \ldots, (n_k, m_k)$, find $\mathrm{NumberOfPartitions}(n, m)$ for such n, m

Input Format
k
n_i m_i (each pair on a newline)
Output Format
$\mathrm{NumberOfPartitions}(n_i, m_i)$ (each result on a newline)
Sample Input

```
7
1 1
2 5
3 3
6 3
7 10
17 8
20 12
```

Sample Output

```
1
2
3
7
15
230
582
```

(b) Number of partitions $\mathrm{P}(n)$ of an integer n is same as $\mathrm{NumberOfPartitions}(n, n)$

Theorem 1 (Pentagonal Number Theorem)**.** *This theorem relates the product and series representations of the Euler function*

$$\prod_{n=1}^{\infty}(1-x^n) = \sum_{k=-\infty}^{\infty}(-1)^k x^{k(3k-1)/2} = 1 + \sum_{k=1}^{\infty}(-1)^k\left(x^{k(3k+1)/2} + x^{k(3k-1)/2}\right) \tag{13}$$

In other words,

$$(1-x)(1-x^2)(1-x^3)\cdots = 1 - x - x^2 + x^5 + x^7 - x^{12} - x^{15} + x^{22} + x^{26} - \cdots \tag{14}$$

The exponents $1, 2, 5, 7, 12, \ldots$ on the right hand side are called (generalized) pentagonal numbers (A001318) and are given by the formula $g_k = k(3k-1)/2$ for $k = 1, -1, 2, -2, 3, -3, \ldots$

The equation (14) implies a recurrence for calculating $\mathrm{P}(n)$, the number of partitions of n:

$$\mathrm{P}(n) = \mathrm{P}(n-1) + \mathrm{P}(n-2) - \mathrm{P}(n-5) - \mathrm{P}(n-7) + \cdots \tag{15}$$

or more formally,

$$\mathrm{P}(n) = \sum_{k\neq 0}(-1)^{k-1}\,\mathrm{P}(n-g_k) \tag{16}$$

Problem Statement:
Write a function $\mathrm{P}()$ which calculates number of partitions using equation (15)

Input Format
Arbitrary number of testcases (each space separated)
Stop when input is negative
Output Format
Number of partitions for each testcase (each on a newline)
Sample Input
```
1 2 4 8 16 32 64 128 -1
```
Sample Output
```
1
2
5
22
231
8349
1741630
4351078600
```

Crazy Video

The hardest What comes next (Euler's pentagonal formula)

§5. Practice Problems 5 - Arrays

5.1. Josephus Problem

Suppose there are n terrorists around a circle facing towards the centre. They are numbered 1 to n along clockwise direction. Initially, terrorist 1 has the sword. Now, the terrorist with sword kills the nearest alive terrorist to its left and passes the sword to 2nd nearest alive terrorist to its left. The process repeats. Basically, every second terrorist is killed until only one survives. Then the last terrorist is killed.

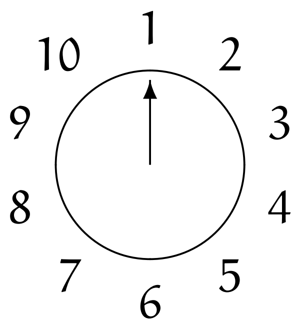


Figure 4: Example arrangement of 10 terrorists

(a) **Problem Statement:**
For a given n, and starting position 1, print the terrorists in the order they are killed

Input Format
n (number of testcases)
k_i for $i = 1, 2, \ldots, n$
Output Format
Order of killing (each result on a newline)
Sample Input
8
1 2 4 7 19 20 42 64 97
Sample Output
1

2 1

2 4 3 1

2 4 6 1 5 3 7

2 4 6 8 10 12 14 16 18 1 5 9 13 17 3 11 19 15 7

2 4 6 8 10 12 14 16 18 20 3 7 11 15 19 5 13 1 17 9

2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 3 7 11 15 19 23 27 31 35 39 1 9 17 25 33 41 13 29 5 37 21

2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 3 7 11 15 19 23 27 31 35 39 43 47 51 55 59 63 5 13 21 29 37 45 53 61 9 25 41 57 17 49 33 1

2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97 7 15 23 31 39 47 55 63 71 79 87 95 11 27 43 59 75 91 19 51 83 35 3 67

Note. *Before proceeding to next task, verify your program on more testcases from here.*

(b) **Problem Statement:**
For a given n, and starting position 1, print the last terrorist killed

Input Format
n (number of testcases)
k_i for $i = 1, 2, \ldots, n$
Output Format
Last Person Killed (each result on a newline)
Sample Input

```
10
1 2 4 7 19 20 64 97 120 2520
```

Sample Output

```
1
1
1
7
7
9
1
67
113
945
```

Note. *Find generalised Josephus problems here and here.*

5.2. Large Factorials

Problem Statement:
Compute factorial of any number $n \leq 96$

Note. `long long int` *can not store more than 20 digits. Digits in testcases* ≤ 150

Input Format
n (number of testcases)
k_i for $i = 1, 2, \ldots, n$
Output Format
$\text{factorial}(k_i)$ (each result on a newline)
Sample Input

```
6
21 33 57 69 77 93
```

Sample Output

```
51090942171709440000

8683317618811886495518194401280000000

40526919504877216755680601905432322134980384796226602145184481280000000000000

171122452428141311372468338881272839092270544893520369393648040923257279754140647424000000000000000

1451830920282858696340707840863082849837403792242083588467815746880619913491564200800652078612480000000
000000000000

1156772507081641574759205162306240436214753229576413535186142281213246807121467315215203289516844845303
8389962893870780907520000000000000000000000
```

5.3. Remove Duplicates

Problem Statement:
Input an integer array and output all unique elements of that array (for repeated elements keep only first occurence)

Input Format
k (number of testcases)
n_i (size of array) followed by n_i elements ($<= 20$) of array (each testcase on a newline)
Output Format
Each result on a newline
Sample Input

```
2
15   1 4 9 10 18 1 4 9 16 17 6 10 11 13 17
20   1 3 4 8 16 1 11 15 17 20 1 3 14 15 18 4 5 17 19 20
```

Sample Output

```
1 4 9 10 18 16 17 6 11 13
1 3 4 8 16 11 15 17 20 14 18 5 19
```

5.4. Maximum Element

Problem Statement:
Given an array of integers which is initially increasing and then decreasing, find the maximum value in the array.
A naive approach is of $\Theta(n)$ time complexity, try to think an algorithm which runs in $\Theta(\log n)$ time (assume strictly increasing and strictly decreasing for this case)

Input Format
k (number of testcases)
n_i (size of array) followed by n_i distinct elements of array (each testcase on a newline)
Output Format
Max element of each array (each result on a newline)
Sample Input

```
2
8   1 4 9 10 18 17 13 11
9   8 13 25 87 167 235 454 512 32
```

Sample Output

```
18
512
```

5.5. Majority Element

Problem Statement:
Write a function which takes an array and prints the majority element (if it exists), otherwise prints -1.
A majority element in an array $A[\,]$ of size n is an element that appears more than $n/2$ times
$\Theta(n^2)$ time complexity is easy, try to think an algorithm which runs in $\Theta(n \log n)$ time. Can you do it in $\Theta(n)$?

Input Format
k (number of testcases)
n_i (size of array) followed by n_i elements of array (each testcase on a newline)
Output Format
Max element of each array (each result on a newline)
Sample Input

```
2
9   3 3 4 2 4 4 2 4 4
8   3 3 4 2 4 4 2 4
```

Sample Output

```
4
-1
```

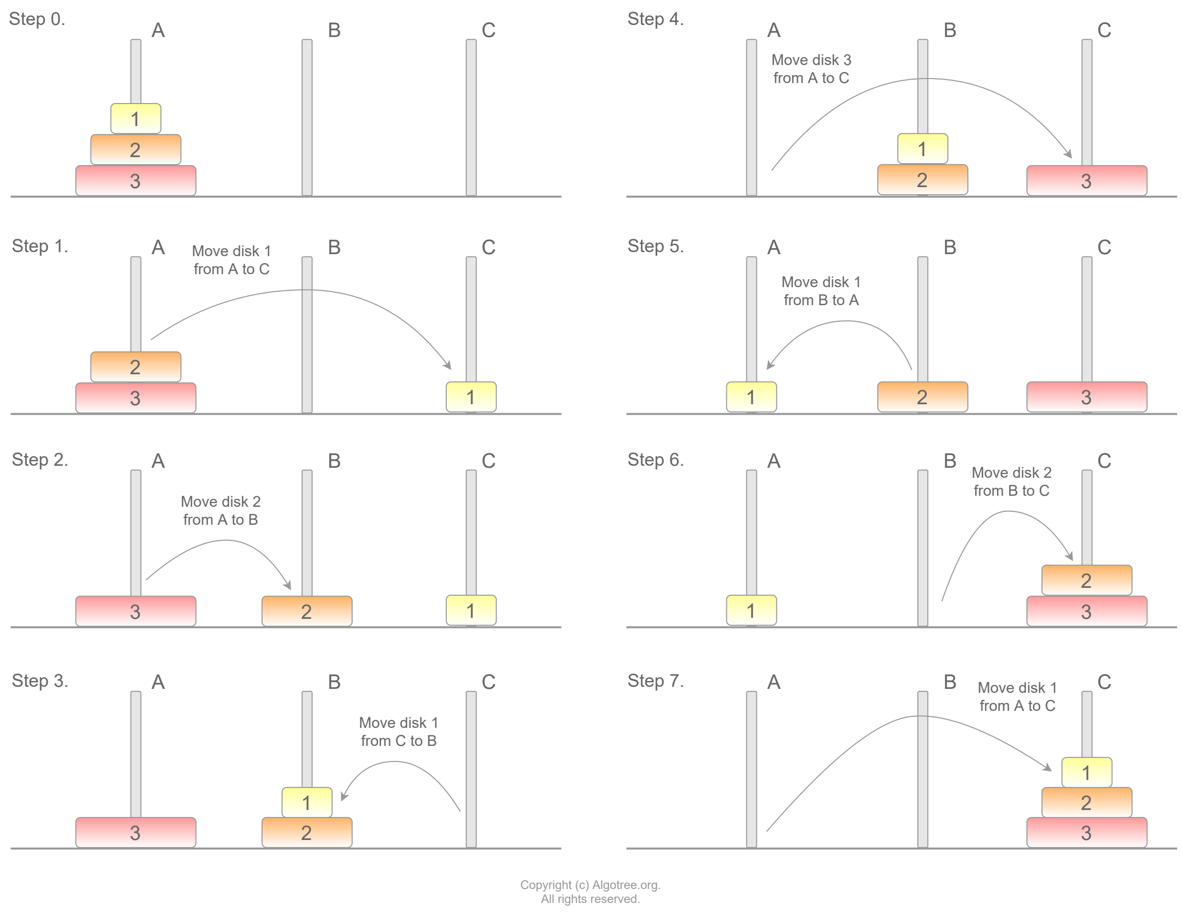
§6. Practice Problems 6 - More Recursion?

6.1. Tower of Hanoi

Tower of Hanoi is a mathematical puzzle with three rods (A,B,C) and n disks on left rod (A) with in decreasing order of their radius from top to bottom . The objective of the puzzle is to move the entire stack of disks to the rightmost rod (C), obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- No disk may be placed on top of a smaller disk.

A example for $n = 3$ is given below



You can also watch the first 2 videos on next page to know more about the problem.
Problem Statement:
For a given n, output the sequence of steps to be taken in the format
`Disk <disk-number> from <rod-name> to <rod-name>`

Task 1: Solve the problem using recursion

Task 2: Solve the problem without using recursion (i.e. only iteration)

Input Format
n (number of testcases)
k_i for $i = 1, 2, \ldots, n$
Output Format
corresponding steps till completetion (each step of each k_i on a newline)
Sample Input

```
4
1 2 3 4
```

Sample Output

```
Disk 1 from A to B

Disk 1 from A to C
Disk 2 from A to B
Disk 1 from C to B

Disk 1 from A to B
Disk 2 from A to C
Disk 1 from B to C
Disk 3 from A to B
Disk 1 from C to A
Disk 2 from C to B
Disk 1 from A to B

Disk 1 from A to C
Disk 2 from A to B
Disk 1 from C to B
Disk 3 from A to C
Disk 1 from B to A
Disk 2 from B to C
Disk 1 from A to C
Disk 4 from A to B
Disk 1 from C to B
Disk 2 from C to A
Disk 1 from B to A
Disk 3 from C to B
Disk 1 from A to C
Disk 2 from A to B
Disk 1 from C to B
```

Crazy Videos

Binary, Hanoi and Sierpinski, part 1

Binary, Hanoi and Sierpinski, part 2

Towers of Hanoi: A Complete Recursive Visualization

6.2. Egyptian Fraction

A fraction is unit fraction if numerator is 1 and denominator is a positive integer, for example 1/3.
Every positive fraction can be represented as sum of unique unit fractions.
Such a representation is called Egyptian Fraction
Problem Statement:
For a given fraction, find its Egyptian Fraction Decomposition

Input Format
n (number of testcases)
N D (numerator and denomination of given fraction)
Output Format
corresponding decomposition, denominators is ascending order (each result on a newline)
Sample Input

```
9
2 5   3 8   7 12   5 23   14 19   52 19   5 31   160 31   6 6
```

Sample Output

```
1/3 1/15
1/3 1/24
1/2 1/12
1/5 1/58 1/6670
1/2 1/5 1/28 1/887 1/2359420
2 + 1/2 + 1/5 + 1/28 + 1/887 + 1/2359420
1/7 + 1/55 + 1/3979 + 1/23744683 + 1/1127619917796295
5 + 1/7 + 1/55 + 1/3979 + 1/23744683 + 1/1127619917796295
1/1
```

6.3. QuickSort

An efficient sorting algorithm.
It is a divide and conquer algorithm like merge sort discused in class.
It first divides the input array into two smaller sub-arrays: the low elements and the high elements. It then recursively sorts the sub-arrays. The steps are:

- Pick an element, called a pivot, from the array.
- Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
- Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

The base case of the recursion is arrays of size zero or one, which are in order by definition, so they never need to be sorted.

The pivot selection and partitioning steps can be done in several different ways; the choice of specific implementation schemes greatly affects the algorithm's performance.
Problem Statement:
Your task is to implement QuickSort. For this problem take last element of array as pivot (Lomuto partition scheme). A complete runthrough is provided in Figure 5.

Note. *You are not provided with the size of array for this problem.*

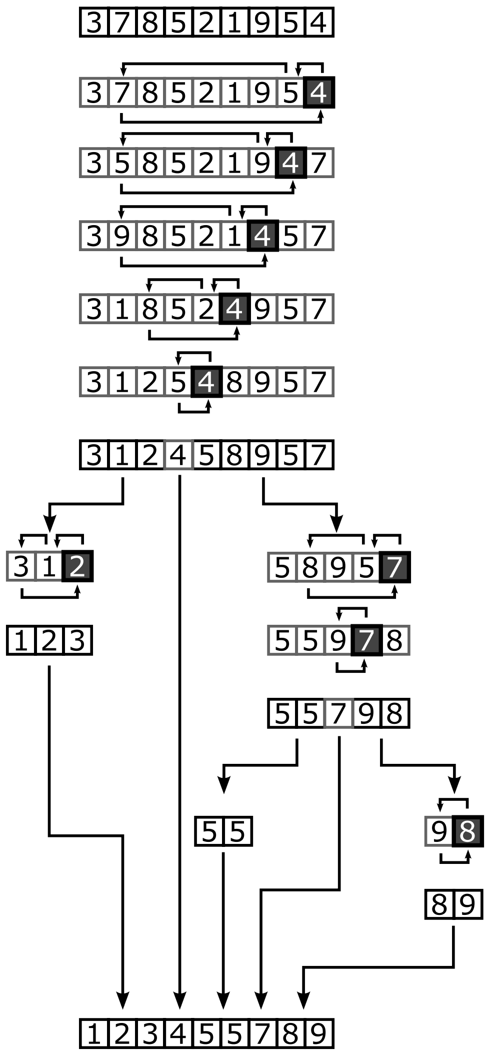


Figure 5: Quick Sort

Input Format
k (number of testcases)
Elements of array (each array on a newline)
Output Format
Sorted array for (each testcase on a newline)
Sample Input
```
2
86 56 24 26 55 73 77 100 53 20 52 59 74 43 19 21 74 51 44 79 76 15 54 62 6 43 42 5 28 84
17 9 10 6 6 12 5 16 18 1 14 11 6 12 14 12 13 10 12 3 2 16 16 14 11 12 7
```
Sample Output
```
5 6 15 19 20 21 24 26 28 42 43 43 44 51 52 53 54 55 56 59 62 73 74 74 76 77 79 84 86 100
1 2 3 5 6 6 6 7 9 10 10 11 11 12 12 12 12 12 13 14 14 14 16 16 16 17 18
```

6.4. Currency Sums

India's currency consists of 1,2,5,10,20,50,100,200,500,2000.

Problem Statement:

Write a function to get number of different ways n can be made using any number of given coins/notes.

Input Format

$n, d_1, d_2, \ldots, d_n$ (number of available coins followed by their denominations)

$t, k_1, k_2, \ldots, k_t$ (number of testcases followed by change denomination)

Output Format

Correct answer (each on a newline)

Sample Input

```
10 1 2 5 10 20 50 100 200 500 2000
10 1 2 5 10 20 50 100 200 500 2000
```

Sample Output

```
1
2
4
11
41
451
4563
73682
6295435
27984272287
```

6.5. Dyck Words

A Dyck word is a string consisting of n X's and n Y's such that no initial segment of the string has more Y's than X's.

For example, $n = 3$: XXXYYY XYXXYY XYXYXY XXYYXY XXYXYY

Problem Statement:

Find number of possible Dyck words for a given n modulo $10^9 + 7$. why this number?

Input Format

n (number of testcases)

n space separated integers

Output Format

Number of ways; don't forget to take modulo! (each result on a newline)

Sample Input

```
8
1 2 3 5 10 20 100 1000
```

Sample Output

```
1
2
5
42
16796
564120378
558488487
110961515
```

§7. Practice Problems 7 - Classes

Here are last set of practice problems. Hopefully you liked these problems and learnt something new in them. The last set are not standard programming problems, Before looking towards efficient techniques I have listed, think on yourself. Ofcourse these may not be "most efficient" (I hope you are comfortable with this by now :D).

Things as simple as polynomial multiplication, division, composition, root finding, factoring and many more are still under research, better/different approaches keep coming. I have linked some of them which you can go through and relatively simpler to implement. But do explore them yourself just a simple search will uncover many research papers and expositions. Take it as a exercise to go through the one you feel most interested in.
You will get a flavour of how things works in academia and also familiarity in reading published work.

El Psy Kongroo

7.1. Polynomials

In Lab-11 optional problem 2 (Autumn 2020-21), you have made a simple polynomial class which does addition, subtraction, multiplication, differentiation, evaluation and printing the coefficients.

Here you have to make the same (+ some more operations) using efficient techniques.

You are free to decide input-output format but make a documentation for it. Some required features:

- Should work for any degree polynomial
- Member functions should return required polynomial(s) and operand polynomials should not get updated during operations
- Use operator overloading whenever possible

Operations required: (with efficient techniques resources also given for them) If you think there is better approach, feel free to discuss :)

- Evaluation (Horner's Algorithm)
- Polynomial Interpolation (Neville's algorithm)
- Addition
- Subtraction
- Multiplication (Using Fast Fourier Transform Video, FFT info, blog)
- Division (Remember synthetic division?, better approach)
- Composition (Based on Ranged Horner's Algorithm)
- Find all roots (Graeffe's Method, Durand–Kerner method, Aberth method, Real-root isolation helps)
  You may need to use complex number library developed in Lab-6 optional problem 2
- Differentiation
- Integration

Fast Fourier Transform Algorithms with Applications
Problem Statement:
Try 5.2 and 6.4 after implementing this

7.2. Linear Algebra

Here you have to make the operations for Matrices using efficient techniques. If done properly, this will also help you in your future endeavors. :)

You are free to decide input-output format but make a documentation for it. Some required features:

- Should work for any rectangular matrix
- Member functions should return required matrix and operand matrices should not get updated during operations
- Use operator overloading whenever possible

Operations required: (with efficient techniques resources also given for them) If you think there is better approach, feel free to discuss :)

- Transpose
- Addition
- Subtraction
- Multiplication (Using Strassen's Algorithm)
- Frobenius Inner Product
- Outer Product
- Determinant
- Adjoint
- Matrix Inverse (Using Gaussian Elimination)
- Moore–Penrose inverse aka PseudoInverse
- Diagonalisation
- LU Decomposition
- QR Decomposition
- Singular Value Decomposition
- Givens rotation
- Householder transformation
- Eigenvalue Decomposition using QR Algorithm

7.3. Symbolic Computation aka Computer algebra

Symbolic Computation is evaluation of expressions without using numerical values directly, it directly operates on the symbols. The system which does this is called a Computer algebra system (CAS) (See Wolfram Alpha)

Example:
$3/9$ is $1/3$ for a CAS but $0.333\ldots$ to some finite number of decimals numerically. There is a loss of precision

Calculating $f = (x+y)^2$ gives $g = x^2 + 2xy + y^2$ but in C++ we need a value for x and y to calculate this expression. If we change values then we need to use f again but with CAS we can use simplified form g.
If this is still not clear try to calculate integral of a function using C++. We have to use numerical techniques but we can use CAS to calculate antiderivative (if it exists) and use that for further analysis.

Motivation

- Manipulate and calculate with symbols without needing to initially assign each symbol a numerical value.
- Answers are exact (infinite precision), as unlike floating point numbers, errors due to rounding are not introduced during calculations
- Solves for analytical expressions. By outputting an algebraic expression, the program can show relationships in a situation that are not apparent when the 'answer' is only a string of digits.
- The strength of most mathematical techniques lies in their generality which relies on the use of symbols rather than specific values

The greater computational power of computer algebra systems means they require more memory than most numerically based mathematics software. But it's benefits outclass this issue.

Problem Statement:
Look up examples of CAS systems and their working

Think how are they implemented?

Implement a class which supports $+, -, /, *$ roots and exponentiation over rational numbers, surds, complex numbers and user variables. (you may want include more operations such as logarithms, trig functions, and numbers like π, e)

Calculate $\sum_{i=1}^{n} i^n$ (my dream :p) using Faulhaber polynomials symbolically using 7.2 or otherwise

References

Essay

§8. Advanced Concepts

8.1. Divide and Conquer

In computer science, divide and conquer is an algorithm design paradigm. A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

Basically 3 steps:

- Divide the problem into sub-problems that are similar to the original but smaller in size
- Conquer the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
- Combine the solutions to create a solution to the original problem

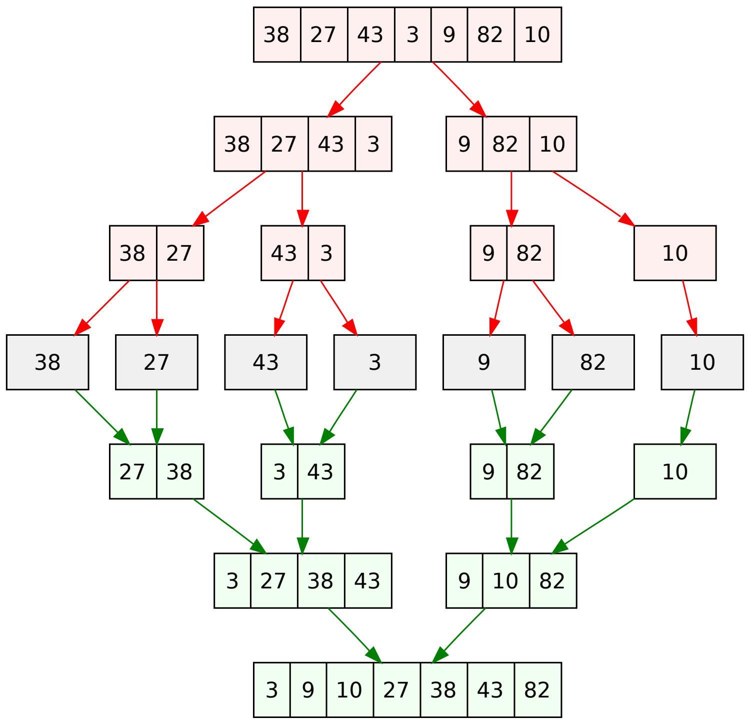


Figure 6: Merge Sort

The divide-and-conquer technique is the basis of efficient algorithms for many problems, such as searching and sorting (e.g., quicksort, merge sort), multiplying large numbers (e.g., the Karatsuba algorithm), finding the closest pair of points, syntactic analysis (e.g., top-down parsers), and computing Discrete Fourier transform (FFT).

Solve questions 5.4, 6.3,7.1 using divide and conquer approach

8.2. Dynamic Programming

Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.
Dynamic Programming Methods

- **Top-down with Memoization** - In this approach, we try to solve the bigger problem by recursively finding the solution to smaller sub-problems. Whenever we solve a sub-problem, we cache its result so that we don't end up solving it repeatedly if it's called multiple times. Instead, we can just return the saved result. This technique of storing the results of already solved subproblems is called Memoization.
- **Bottom-up with Tabulation** - Tabulation is the opposite of the top-down approach and avoids recursion. In this approach, we solve the problem "bottom-up" (i.e. by solving all the related sub-problems first). This is typically done by filling up an n-dimensional table. Based on the results in the table, the solution to the top/original problem is then computed.

Recursive Formula

```
int fib(int n) {
    int ans;
    if(n==0 || n==1) {
        ans = n;
    }
    else {
        ans = fib(n-1)
            + fib(n-2);
    }
    return ans;
}
cout << fib(n);
```

Bottom-up with Tabulation

```
vector<int> f(n+1);
f[0] = 0;
f[1] = 1;
for (int i=2; i<=n; ++i) {
    f[i] = f[i-1] + f[i-2];
}
cout<<f[n];
```

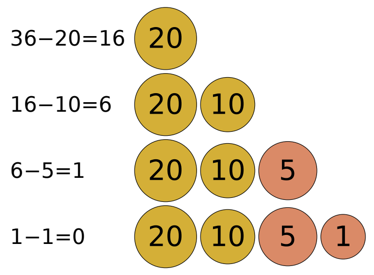
Top-down with Memoization

```
vector<int> f(n+1, -1);
f[0] = 0;
f[1] = 1;
int fib(int n) {
    int ans;
    if (f[n]!=-1) {
        ans = f[n];
    }
    else {
        ans = fib(n-1) + fib(n-2);
        f[n] = ans;
    }
    return ans;
}
```

Solve questions 4.3, 4.4, 4.6, 6.4 and 6.5 using dynamic programming

8.3. Greedy Algorithms

A greedy algorithm, as the name suggests, always makes the choice that seems to be the best at that moment. Means, it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution
In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless, a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable time.
An Example
Greedy algorithms to determine minimum number of coins to give while making change. These are the steps a human would take to emulate a greedy algorithm to represent 36 cents using only 1, 5, 10, 20 coins.



The coin of the highest value, less than the remaining change owed, is the local optimum
As a practice solve question 6.2 using greedy approach.

8.4. Backtracking

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution

Types

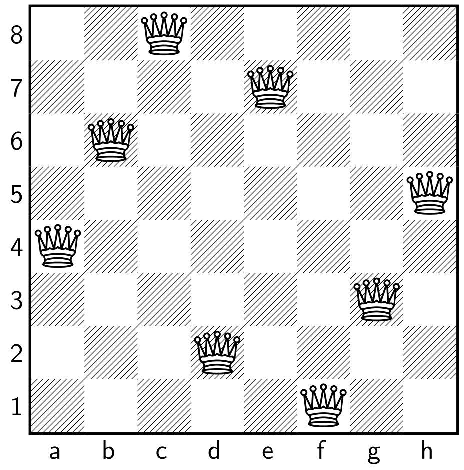
Decision Problem search for a feasible solution

Optimization Problem search for the best solution

Enumeration Problem find all feasible solutions

Backtracking can be applied only for problems which admit the concept of a "partial candidate solution" and a relatively quick test of whether it can possibly be completed to a valid solution. It is useless, for eg., for locating a given value in an unordered table. When it is applicable, however, backtracking is often much faster than brute force enumeration of all complete candidates, since it can eliminate many candidates with a single test.

The classic textbook example of the use of backtracking is the eight queens puzzle, that asks for all arrangements of eight chess queens on a standard chessboard so that no queen attacks any other. In the common backtracking approach, the partial candidates are arrangements of k queens in the first k rows of the board, all in different rows and columns. Any partial solution that contains two mutually attacking queens can be abandoned.



Most of the problems, can be solved using other known algorithms like Dynamic Programming or Greedy Algorithms in logarithmic, linear, linear-logarithmic time complexity in order of input size, and therefore, outshine the backtracking algorithm in every respect (since backtracking algorithms are generally exponential in both time and space). However, a few problems still remain, that only have backtracking algorithms to solve them so far.

References

Divide and Conquer: Wikipedia
5 Simple Steps for Solving Dynamic Programming Problems: Reducible
What is Dynamic Programming?
Coin Sums: Project Euler
Greedy Algorithms: Wikipedia
Backtracking: Wikipedia
Backtracking: Geeks for Geeks

§9. Hints

9.1. Practice Problems 0

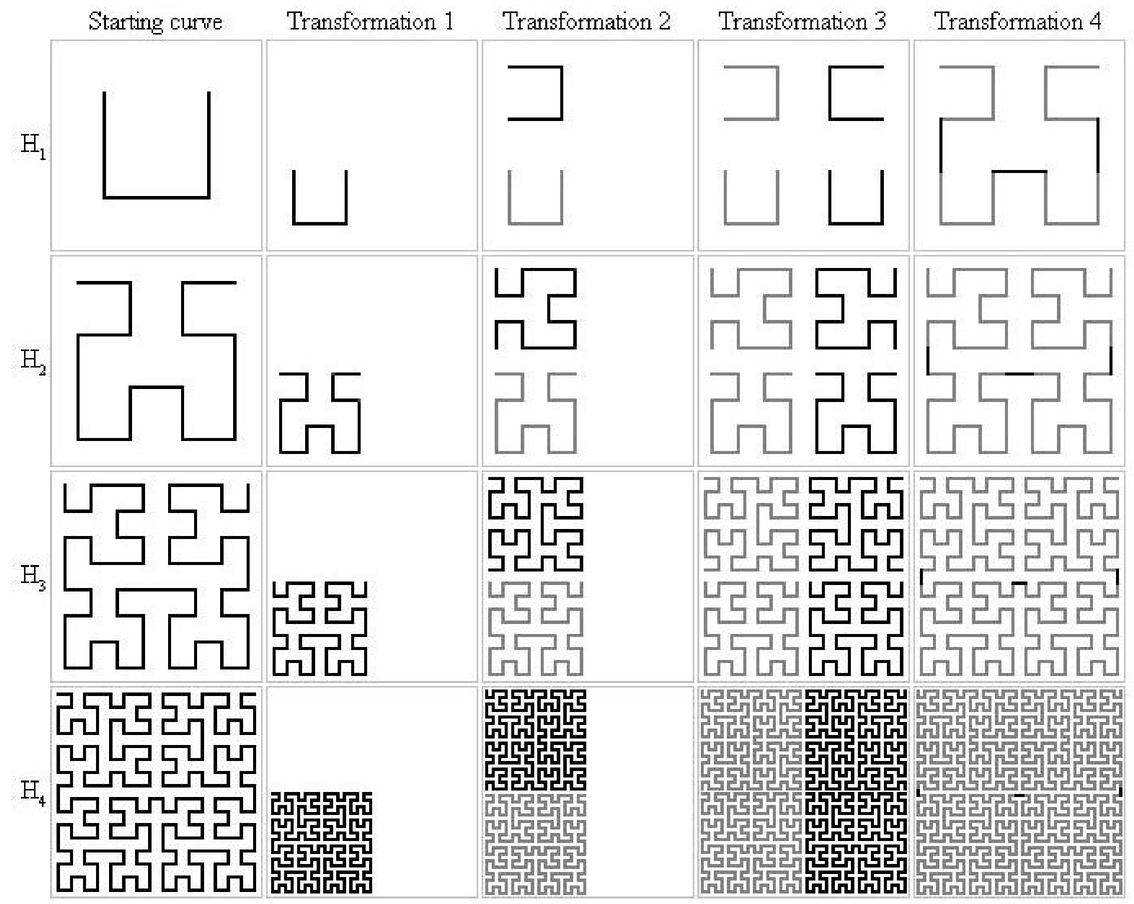


Figure 7: Hilbert Curve Transformations

9.2. Practice Problems 1

1. How to access digits? - % operator, use a variable to store sum.
2. Create the reversed number and check if it is equal to given number.
3. Think about conversion of integer part and fractional part seperately.
   There are multiple methods, one way is using binary representation and sum corresponding powers to 2. The smarter approach uses *doubling* instead of calculating each power of 2.

9.3. Practice Problems 2

1. The sequence repeats when we encounter '0' then '1'.
2. Similar to calculating e (problem given in slides).
3. Figure out the pattern of coefficients of $f(x_i)$ and add them up.
4. $1 \leq a, b, c < 1000$. You can check all combinations of a, b, c which sum to 1000 and are Pythagorean Triplet. There is a better solution also, can you figure it out?
5. a) This is just Problem 1.3 with extra steps :D.
   b) Fermat's Little Theorem.

9.4. Practice Problems 3

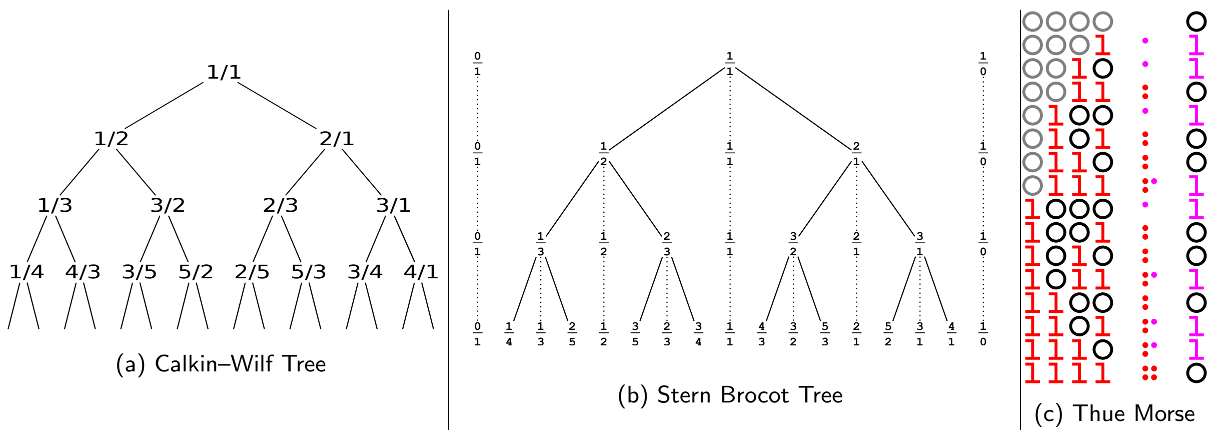
1. Straightforward implementation
2. Straightforward implementation
3. A simple method is to count the integers i's such that $1 \le i \le n$ and $\gcd(i, n) = 1$.
   Better way is to use the Euler's Product Formula

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right) \qquad \text{For all primes } p \le n \tag{17}$$

   So, if $n = p_1^{k_1} p_2^{k_2} \cdots p_r^{k_r}$, where $p_1, p_2, \ldots, p_r$ are the distinct primes dividing n

$$\varphi(n) = p_1^{k_1-1}(p_1-1)\, p_2^{k_2-1}(p_2-1) \cdots p_r^{k_r-1}(p_r-1) \tag{18}$$

4. Similar pattern as in Stern Brocot Tree. Also, exactly same pattern between 3 consecutive terms



(a) Calkin–Wilf Tree (b) Stern Brocot Tree (c) Thue Morse

Figure 8

5. Try skipping days, months, years and centuries :D

   Is there a formula?

9.5. Practice Problems 4

Straightforward implementation of given recursive definition in problems 4.1, 4.2 & 4.6 b)

For 4.3, 4.4 & 4.6 a) try to find a recursive definition using *previous terms*
Observe carefully to find patterns in figures.

For 4.5, how will you calculate n^{th} term? Another approach is shown in Figure 8c.

9.6. Practice Problems 5

1. For a) Straightforward implementation by naive method. But it is possible to solve this without using array too. Think about it.

   For b) The terrorist with the sword when 2^n terrorists are left is the last one to die.
2. How will you store the answer? So, how can you arrive at final answer?
3. A duplicate element occurs more than one time. So, you can count occurence of each element
4. Do you really need to check all elements? Can you skip some of them?
5. Easy to solve in $\Theta(n^2)$ but there is a $\Theta(n)$ solution. You can traverse through all elements in $\Theta(n)$ time. Is there a way to find majority element while traversing?

9.7. Practice Problems 6

1. For recursive approach, check this

   For iterative approach, there is a simple pattern that repeats alternately, can you figure it out?
2. How to calculate first element of decompositon? After subtracting first element what will you do?
3. Divide and Conquer Approach
4. Recursive approach like 4.6. How can you find answer for a n using answers for lower n?
5. Does it feel similar to 4.4?

9.8. Practice Problems 7

No hints here. The goal of this set of problems is to make you explore and understand methods by your own.