

CS 101 Computer Programming and Utilization

Practice Problems

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

Autumn Semester 2020-21

Last update: 2021-02-02 20:32:45+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 $\text{gcd}(a, b)$ routine should return the error message “ $\text{gcd}(0, 0)$ is undefined” instead of going into an infinite loop or returning a “division by zero” error.

Contents

1 Practice Problems 1 - Introduction	3
1.1 Hilbert Curve	3
2 Practice Problems 2 - Loops	4
2.1 Pisano Period	4
2.2 π	5
2.3 Simpson's Rule	5
3 Practice Problems 3 - Functions	6
3.1 Calendar	6
3.2 Farey Sequence	6
3.3 Thue-Morse Sequence aka Fair Share Sequence	7
3.4 Collatz Conjecture	7
4 Practice Problems 4 - Recursion	8
4.1 Ackermann function	8
4.2 GridPaths	9
4.3 Delannoy Numbers	10
4.4 Partitions	11

5 Practice Problems 5 - Arrays	13
5.1 Horner's method	13
5.2 Large Factorials	13
5.3 Remove Duplicates	14
5.4 Maximum Element	14
5.5 Majority Element	14
5.6 Matrix Inversion	15
6 Practice Problems 6	16
6.1 Currency sums	16
6.2 QuickSort	16
6.3 Dyck words	18
6.4 Egyptian Fraction	18
7 Advance Concepts	19
7.1 Dynamic Programming	19
7.2 Greedy Algorithms	19
8 Hints	21
8.1 Practice Problems 1	21
8.2 Practice Problems 2	21
8.3 Practice Problems 3	21

§1. Practice Problems 1 - Introduction

1.1. Hilbert Curve

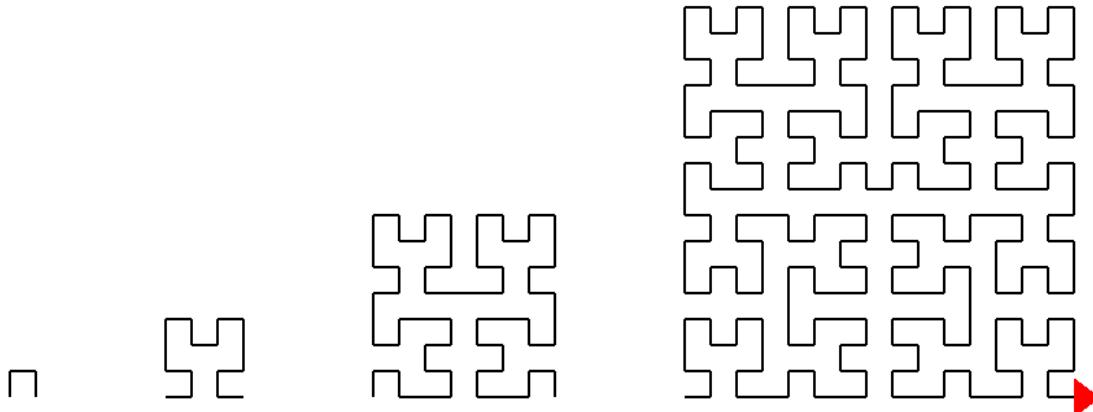


Figure 1: Hilbert Curve

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

§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, ... ([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, \dots, k_n

Input Format

n

k_1, k_2, \dots, 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, \dots, k_n , find $\max(\pi(i))$ for $i = 1, 2, \dots, k$ and corresponding i

If there are 2 (or more) such i 's, output smallest of them

Input Format

n

k_1, k_2, \dots, 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

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

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

Problem Statement:

Calculate π using first $k_i + 1$ terms¹ 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

2.3. Simpson's Rule

A method for numerical integration

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

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

Problem Statement:

Solve Equation (4) giving the answers correct to 7 decimal places (Use 101 ordinates)

$$\int_{0.5}^1 \frac{\sin \theta}{\theta} d\theta \quad (4)$$

Correct Answer = 0.4529756

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

§3. Practice Problems 3 - Functions

3.1. 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 efficient 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.

3.2. 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 $1/2$.

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](#)

3.3. 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](#)

3.4. 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.

Collatz Conjecture states that no matter which positive integer we start with; we always end up with 1.

Problem Statement:

Define a function which performs this operation repeatedly on the result at each step; beginning with a given input n ($n < 10^6$), returns the number of operations required to reach 1²

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

²As 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

§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 A() that takes two inputs n and m and outputs the number $A(m, n)$ where $A(m, n)$ is defined as

$$A(0, n) = n + 1 \quad (5)$$

$$A(m + 1, 0) = A(m, 1) \quad (6)$$

$$A(m + 1, n + 1) = A(m, A(m + 1, n)) \quad (7)$$

Problem Statement:

For k pairs $(m_1, n_1), (m_2, n_2), \dots, (m_k, n_k)$, find $A(m, n)$ for such m, n

Input Format

k

$m_i \ n_i$ (each pair on a newline)

Output Format

$A(m_i, n_i)$ (each result on a newline)

Sample Input

```
7
0 0
0 4
1 3
2 2
3 4
4 0
4 1
```

Sample Output

```
1
5
5
7
125
13
65533
```

4.2. GridPaths

Write a recursive function `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 2

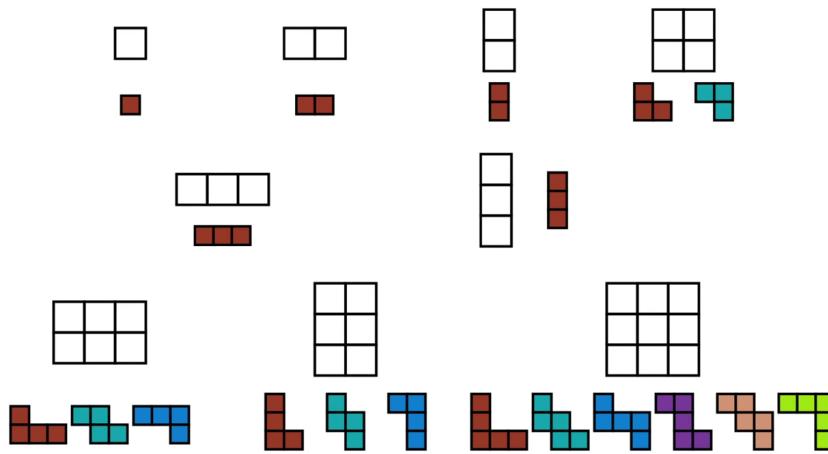


Figure 2: Number of Grid Paths for $m, n \in \{1, 2, 3\}$

Problem Statement:

For k pairs $(n_1, m_1), (n_2, m_2), \dots, (n_k, m_k)$, find `NumberOfGridPaths(n, m)` for such n, m

Input Format

k

$n_i \ m_i$ (each pair on a newline)

Output Format

`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?

4.3. 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 `DelannoyNumber()` to count number of these paths Constraints: $n, m \geq 0$

Examples given in Figure 3

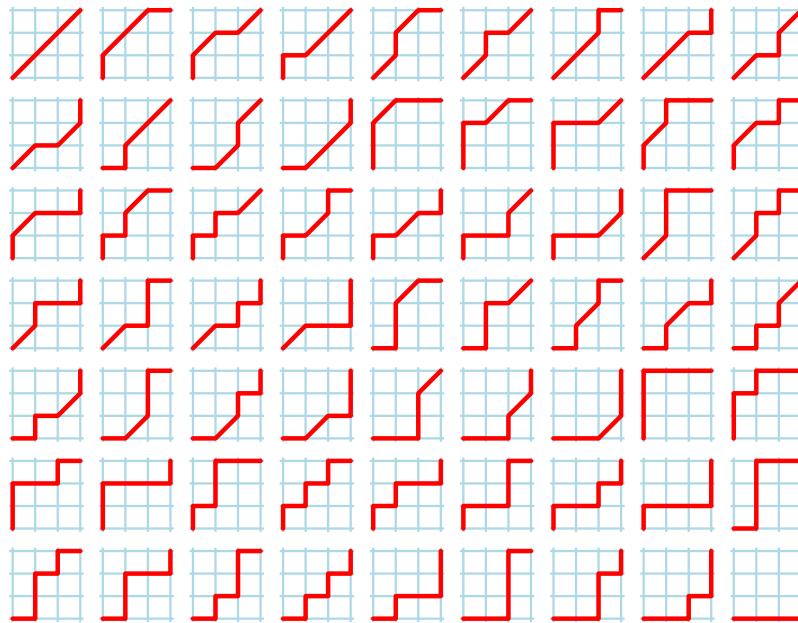


Figure 3: $\text{DelannoyNumber}(3,3) = 63$

Problem Statement:

For k pairs $(n_1, m_1), (n_2, m_2), \dots, (n_k, m_k)$, find $\text{DelannoyNumbers}(n, m)$ for such n, m

Input Format

k

$n_i \ m_i$ (each pair on a newline)

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

245157

62390545

4.4. Partitions

- (a) Write a recursive function 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 4

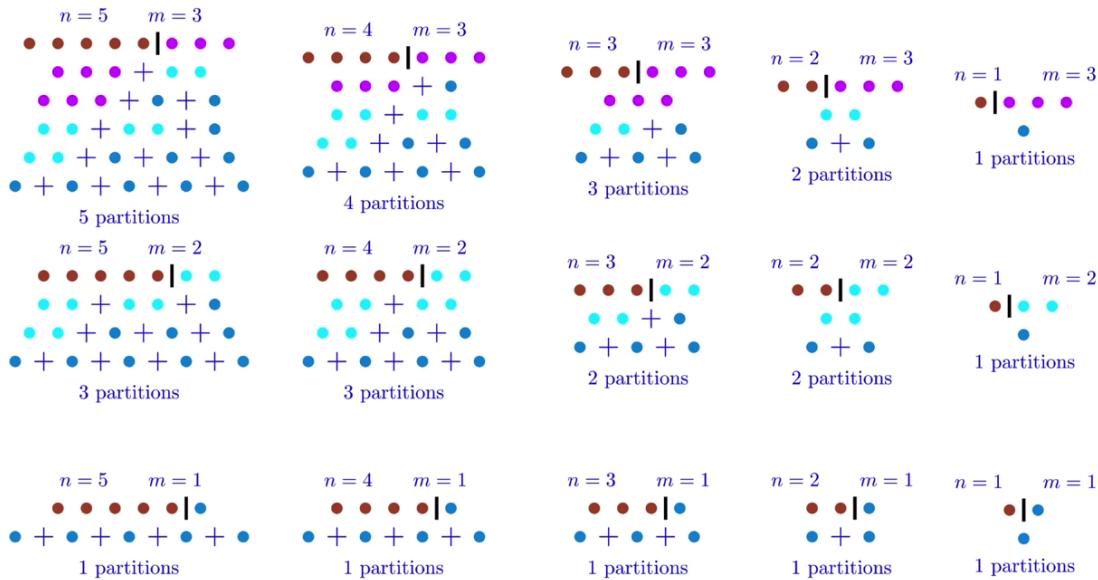


Figure 4: Partition n objects using parts up to m

Can you find efficient solution?

Problem Statement:

For k pairs $(n_1, m_1), (n_2, m_2), \dots, (n_k, m_k)$, find NumberOfPartitions(n, m) for such n, m

Input Format

k

$n_i \ m_i$ (each pair on a newline)

Output Format

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 $P(n)$ of an integer n is same as $\text{NumberOfPartitions}(n, n)$

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

$$\prod_{n=1}^{\infty} (1 - x^n) = \sum_{k=-\infty}^{\infty} (-1)^k x^{k(3k-1)/2} = 1 + \sum_{k=1}^{\infty} (-1)^k \left(x^{k(3k+1)/2} + x^{k(3k-1)/2} \right) \quad (8)$$

In other words,

$$(1 - x)(1 - x^2)(1 - x^3) \cdots = 1 - x - x^2 + x^5 + x^7 - x^{12} - x^{15} + x^{22} + x^{26} - \cdots \quad (9)$$

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

The equation (9) implies a recurrence for calculating $P(n)$, the number of partitions of n :

$$P(n) = P(n-1) + P(n-2) - P(n-5) - P(n-7) + \cdots \quad (10)$$

or more formally,

$$P(n) = \sum_{k \neq 0} (-1)^{k-1} P(n - g_k) \quad (11)$$

Problem Statement:

Write a function $P()$ which calculates number of partitions using equation (10)

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 \(Euler's pentagonal formula\)](#)

Exciting Puzzle

[Towers of Hanoi: A Complete Recursive Visualization](#)

5.3. Remove Duplicates

Problem Statement:

Input an integer array and output all unique elements of that array (for repeated elements keep only first occurrence)

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

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

A naive approach is of $\Theta(n^2)$ time complexity, try to think an algorithm which runs in $\Theta(n \log n)$ time.

Can you do it in $\Theta(n)$ time?

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

5.6. Matrix Inversion

Problem Statement:

Write a function which finds multiplicative inverse of a square matrix (if it exist) using Gaussian Elimination
Each element of your output should contain exactly 2 decimal places

Input Format

k (number of testcases)

n (size of matrix) followed by $n \times n = n^2$ elements of array

Output Format

Inverted Matrix

Sample Input

```
3
2
4 7
2 6
3
2 9 2
3 7 1
1 1 1
5
1 -4 3 -4 -4
-10 4 0 7 -3
-5 4 -5 0 -3
9 -2 -4 -7 1
8 -4 3 -7 5
```

Sample Output

```
0.60 -0.70
-0.20 0.40

-0.43 0.50 0.36
0.14 0.00 -0.29
0.29 -0.50 0.93
```

```
0.78 17.89 -5.56 11.78 5.67
1.69 39.93 -11.70 25.35 12.61
1.04 22.19 -6.74 14.37 7.22
-0.19 -2.93 0.70 -1.85 -1.11
-0.78 -14.89 4.56 -9.78 -4.67
```

§6. Practice Problems 6

6.1. 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

k_1, k_2, \dots, k_n

Output Format

Correct answer (each on a newline)

Sample Input

9

1 2 5 10 20 50 100 200 500 2000

Sample Output

1

4

11

41

451

4563

73682

6295435

27984272287

6.2. QuickSort

An efficient sorting algorithm.

It is a divide and conquer algorithm like merge sort discussed 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.

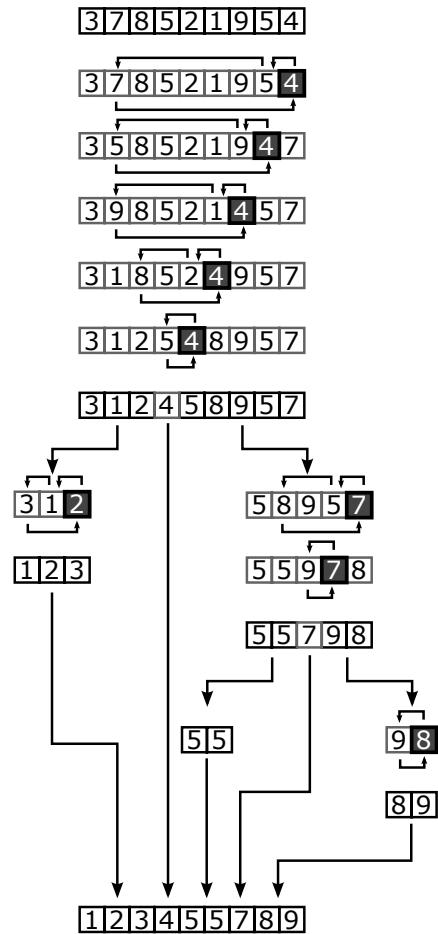


Figure 5: Quick Sort

Input Format

k (number of testcases)

n (size of array) n elements of array

Output Format

Sorted array for (each testcase on a newline)

Sample Input

2

30

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

27

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 13 14 14 14 16 16 16 17 18

6.3. 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.

Some examples for $n = 6$: 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

7

1 2 3 5 10 20 100

Sample Output

1

2

5

42

16796

564120378

558488487

6.4. Egyptian Fraction

A fraction is unit fraction if numerator is 1 and denominator is a positive integer, for example $1/3$ is a unit fraction. 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 denominator of given fraction

Output Format

corresponding decomposition, denominators in ascending order (each result on a newline)

Sample Input

5

2 5 3 8 7 12 5 23 14 19 5 31

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

1/7 1/55 1/3979 1/23744683 1/2024035271

§7. Advance Concepts

7.1. 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

```
1 int fibonacci(int n){  
2     if(n <= 1) {  
3         return n;  
4     }  
5     else {  
6         return fibonacci(n-1)+fibonacci(n-2);  
7     }  
8 }
```

Bottom-up with Tabulation

```
1 int fibonacci[n];  
2 fibonacci[0] = 0;  
3 fibonacci[1] = 1;  
4 for (int i = 2; i < n; ++i)  
5 {  
6     fibonacci[i] = fibonacci[i-1] + fibonacci[i-2];  
7 }  
8 cout<<fibonacci[n];
```

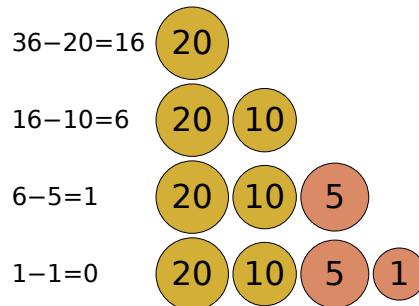
Solve questions 4.2, 4.3, 4.4 of practice problems 4 and 6.1, 6.3 of practice problems 6 using dynamic programming

7.2. 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 6.4 of practice problems 6 using greedy approach

References

[5 Simple Steps for Solving Dynamic Programming Problems](#)

[What is Dynamic Programming](#)

[Coin sums: Project Euler](#)

[Greedy Algorithms: Wikipedia](#)

§8. Hints

8.1. Practice Problems 1

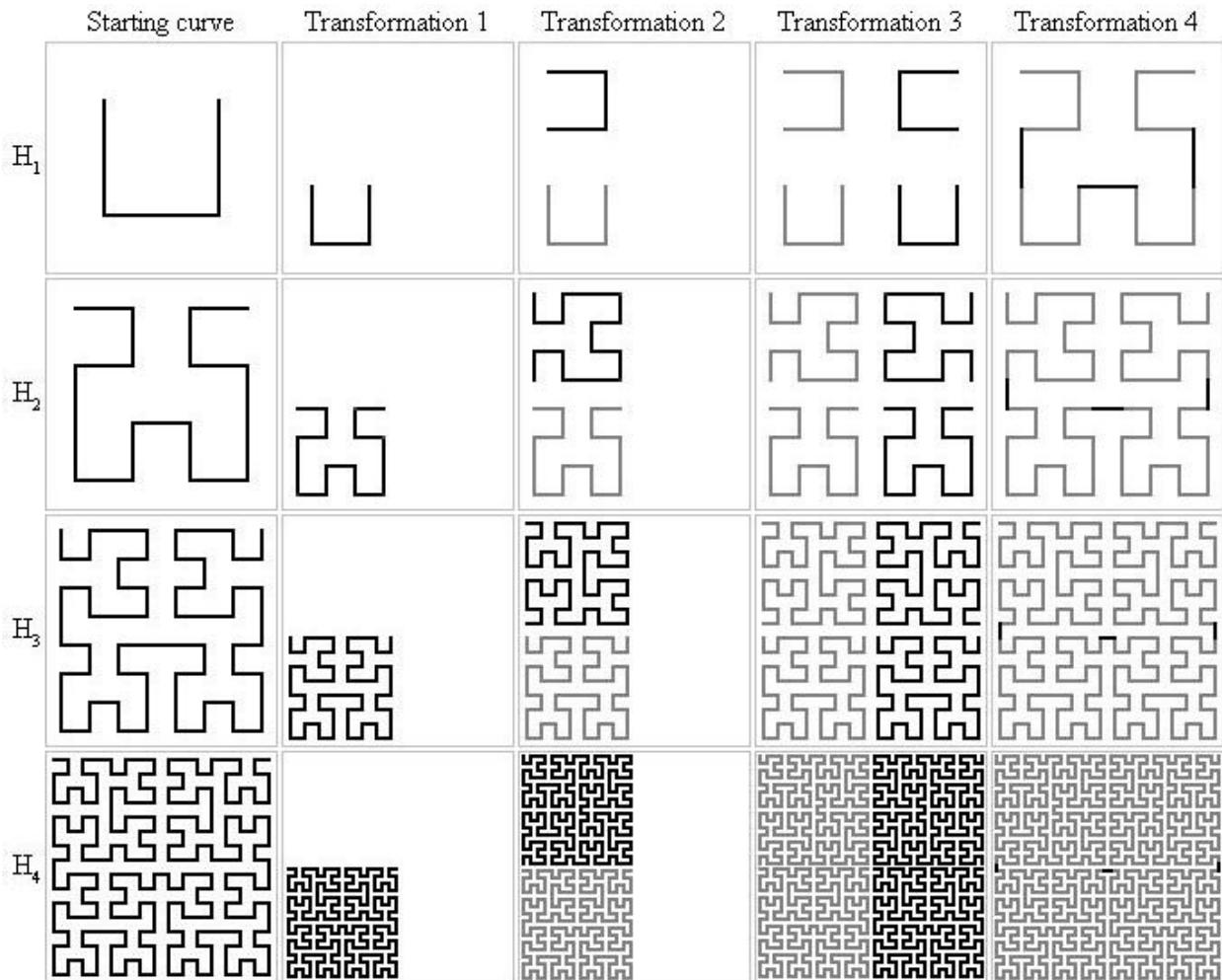


Figure 6: Hilbert Curve Transformations

8.2. Practice Problems 2

1. The sequence repeats when we encounter '0' then '1'
2. Similar to calculating e (problem given in slides)

8.3. Practice Problems 3

1. Try skipping days, months, years and centuries :D
Is there a formula?
2. Similar pattern as in Stern Brocot Tree. Also, exactly same pattern between 3 consecutive terms
3. How will you calculate n^{th} term?

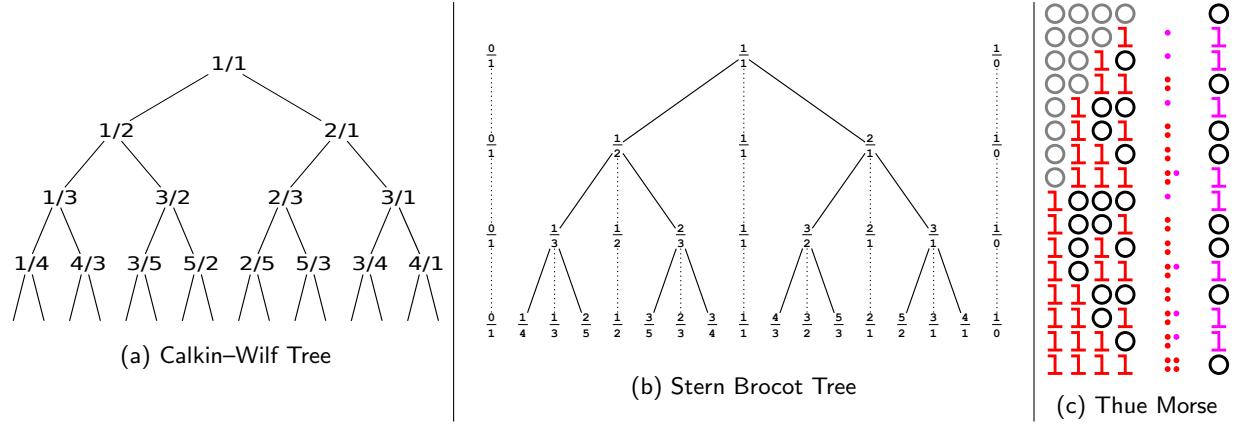


Figure 7