

अन्तर्चक्षुः  
भगवान् महर्षि हिरण्यगर्भ  
**Beacons of Light**

*Edsger Wybe Dijkstra*  
*Richard Phillips Feynman*  
*Leonhard Euler*

**Artifacts**  
*Monographs*

# Monographs

<b>I</b>	<b>Computer Science</b>	<b>1</b>
1	Discipline of Competitive Programming : A Hacker's Perspective	2
2	Elements of Coding : Science of Deriving Correct Programs	3
3	Elements of Coding Linear Algebra : The Nucleus of Artificial Intelligence	4
4	Elements of Software Design Patterns	6
5	Elements of Coding AI	7
6	Elements of Coding DL (Deep Learning)	8
7	Elements of Coding ML : Internals of Machine Learning Library MLPack	9
8	Conceptual BitCoin : Blockchain Coding	10
9	Conceptual Data Science Interviews	11
10	Conceptual Dependency Injection : Unwiring Simplified in C++	12

<b>11 Conceptual Dynamic Programming : Optimal Coding Simplified</b>	<b>13</b>
<b>12 Conceptual Programming Interviews</b>	<b>14</b>
<b>13 Conceptual Machine Learning</b>	<b>15</b>
<b>14 Conceptual Programming of STL Algorithms</b>	<b>16</b>
<b>15 Conceptual Solutions to (CLRS) Introduction to Algorithms</b>	<b>17</b>
<b>16 Conceptual Programming of Algorithms Using Dijkstra's Approach</b>	<b>18</b>
<b>17 Conceptual Solutions to Pattern Recognition and Machine Learning</b>	<b>19</b>
<b>18 Science of Deriving Beautiful Programs</b>	<b>20</b>
<b>19 Modern C++ Ranges : A Revolution in STL</b>	<b>21</b>
<b>20 Elements of C++20</b>	<b>22</b>
<b>21 Solving Problems using Dynamic Programming : A Hacker's Perspective</b>	<b>23</b>
<b>22 Hacking TensorFlow Internals : An Insider's Commentary on A Learning System</b>	<b>37</b>
<b>23 Advanced C++ FAQs Vol 1 &amp; 2</b>	<b>38</b>
<b>24 C++14 FAQs</b>	<b>39</b>
<b>25 The Boost C++ Libraries: Generic Programming</b>	<b>40</b>
<b>26 Generic Algorithms and Data Structures using C++11</b>	<b>41</b>
<b>27 C++11 Standard Library: Usage and Implementation</b>	<b>42</b>

<b>28</b>	<b>Foundation of Algorithms in C++11</b>	<b>43</b>
<b>29</b>	<b>C++11 Algorithms : Using and Extending C++11, Boost and Beyond</b>	<b>44</b>
<b>30</b>	<b>Cracking Programming Interviews : 500 Questions with Solutions</b>	<b>45</b>
<b>31</b>	<b>Top 20 Coding Interview Problems Asked in Google with Solutions</b>	<b>46</b>
<b>32</b>	<b>Top 10 Coding Interview Problems Asked in Google with Solutions</b>	<b>47</b>
<b>II</b>	<b>Physics</b>	<b>48</b>
<b>III</b>	<b>Mathematics</b>	<b>50</b>

**Part I**

**Computer Science**



# Discipline of Competitive Programming : A Hacker's Perspective

$x^2 \triangleq$

Monograph



# **Elements of Coding : Science of Deriving Correct Programs**

# Elements of Coding Linear Algebra : The Nucleus of Artificial Intelligence

## Excerpt from the Chapter Algebraic Concepts

**Concept**  $\mathcal{C}$  is a predicate describing a set of syntactic and semantic requirements on related types ( $\langle T_i \rangle$ ) together with a collection of similar procedures ( $f : T^i \rightarrow T^j$ ) stated in terms of the properties, attributes and type functions ( $F : \mathcal{C}^i \rightarrow \mathcal{C}^j$ ) defined on the types.

$$\therefore \mathcal{C}(\langle T_i \rangle) \triangleq \bigwedge \langle \Psi_j \rangle$$

where  $\triangleq$  stands for *is defined by* and the  $\Psi_j$  represent independent clauses defining the concept.

```
template<class T>
    concept integral = is_integral_v<T>;
```



*If a type  $T$  fulfills all the requirements of a concept  $\mathcal{C}$ , then  $T$  **models**  $\mathcal{C}$ , i.e.  $T \models \mathcal{C}$ .*

$\text{int8\_t}$  and  $\text{uint8\_t} \models \text{integral}$ .

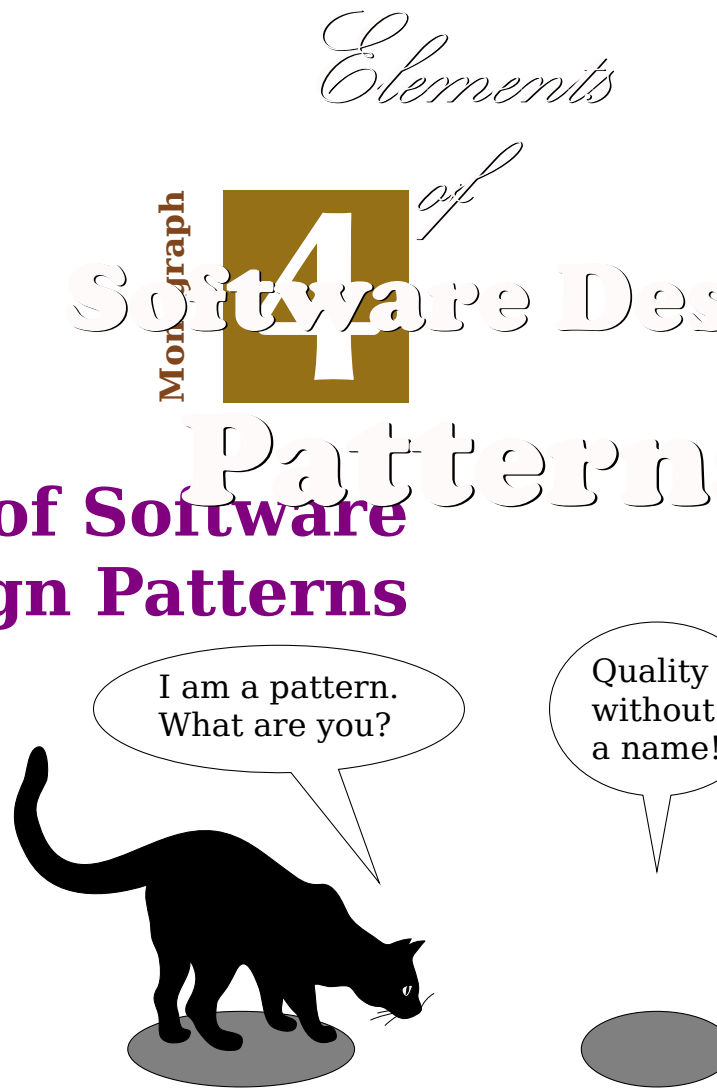
*Concept  $\mathcal{C}^i$  is a **refinement** of concept  $\mathcal{C}^j$  if it subsumes the latter, i.e. if  $\mathcal{C}^i$  is true for a set of types, then  $\mathcal{C}^j$  is also true for the same set.*

In other words,  $\mathcal{C}^i$  **refines**  $\mathcal{C}^j$  ( $\mathcal{C}^i \sqsubset \mathcal{C}^j$ ) by addition of more requirements to  $\mathcal{C}^j$ , i.e.  $\mathcal{C}^j$  **weakens**  $\mathcal{C}^i$  ( $\mathcal{C}^j \sqsupset \mathcal{C}^i$ ).

```
template<class T>
    concept signed_integral = integral<T> && is_signed_v<T>;
        signed_integral  $\sqsubset$  integral
        int8_t  $\models$  signed_integral

template<class T>
    concept unsigned_integral = integral<T> && !signed_integral<T>;
        unsigned_integral  $\sqsubset$  integral
        uint8_t  $\models$  unsigned_integral
```

# Elements of Software Design Patterns



Chandra Shekhar Kumar  
*Ancient Science Publishers*



# Elements of Coding AI



# **Elements of Coding DL (Deep Learning)**



# **Elements of Coding ML : Internals of Machine Learning Library MLPack**



## **Conceptual BitCoin : Blockchain Coding**



# Conceptual Data Science Interviews

Monograph

10

# **Conceptual Dependency Injection : Unwiring Simplified in C++**



Monograph



# **Conceptual Dynamic Programming : Optimal Coding Simplified**

Monograph

12

# Conceptual Programming Interviews

Monograph

13

# Conceptual Machine Learning

Monograph

14

# Conceptual Programming of STL Algorithms

Monograph

15

# Conceptual Solutions to (CLRS) Introduction to Algorithms

# Conceptual Programming of Algorithms Using Dijkstra's Approach

Monograph

17

# **Conceptual Solutions to Pattern Recognition and Machine Learning**

Monograph

18

# Science of Deriving Beautiful Programs



Monograph

19

## **Modern C++ Ranges : A Revolution in STL**



## **Elements of C++20**

## Solving Problems using Dynamic Programming : A Hacker's Perspective

A hacker's approach to a coding problem is beyond the foundational aspect of underlying genetic and computational structures, often termed as  $\pi^\infty$ .

## Solving Problems using Dynamic Programming

$$f_n(k, p) = \begin{cases} 1 & \text{if } k=0 \text{ and } p=0 \\ \sum_{i \in \mathcal{I}(k)} (f_{n-1}(k-1, p-i)) & \text{otherwise} \end{cases}$$

*A Hacker's Perspective*  
 $\pi^\infty$

```
1 function perfectkriya(a)
2   f[0..a] ← {∞}
3   f[0] ← 0
4   for β ∈ [1..a] do
5     for γ ∈ [1..√β] do
6       f[β] ← min(f[β], f[β-γ²]+1)
7   end for
8   return f[a]
9 end function

int firstkriya(int beta, int alpha)
{
  // max no. of Kriyas with beta Pranayams
  int n = std::min(beta, alpha);
  std::vector<int> f(n, 0);
  f[0] = 1;
  for(int p = 1; p <= beta; p++)
  {
    int prev = 0, cur = 0;
    for(int k = 0; k < n; k++)
    {
      cur = f[k];
      f[k] = prev + (k+1 < n ? f[k+1] : 0);
      prev = cur;
    }
  }
  return f[0];
}
```

**Chandra Shekhar Kumar**

*Ancient Science Publishers*

A concept becomes *not difficult* because the *complexities* built into it are clarified. In a bid to reach the *core* of the problem, the concept is split-broken into fragments, *complexities* are exposed and *delicate* points are examined. Then the concept is *recomposed* to make it integral and as a result, this reintegrated concept becomes sufficiently simple and comprehensible.

This helps build a hacker's insight to reveal the internal structure and internal logic of the concepts, algorithms and mathematical theorems.

This book provides a hacker's perspective to solving problems using dynamic programming. Written in an extremely lively form of problems and solutions (including code in modern C++ and pseudo style), this leads to extreme simplification of optimal coding with great emphasis on unconventional and integrated science of dynamic Programming. Though aimed primarily at serious programmers, it imparts the knowledge of deep inter-

nals of underlying concepts and beyond to computer scientists alike.

*Ancient Science Publishers*  
July, 2020. 256 pages

Chandra Shekhar Kumar  
ISBN 9781722497170

*Beautiful (C++) code snippets. Unique yogic exposition to coding.*

*Ancient Science Hackers*

### **Excerpt from the Chapter (Optimal Loot Partition):**

**§ Problem.** *The head of a gang of robbers embarks on distribution of the looted amount  $l(> 0)$ , starting with division into two parts :  $x$  and  $l-x$  for  $0 \leq x \leq l$ . From  $x$  : they get a return of  $u(x)$  such that they are left with a lesser amount  $\alpha x$  :  $0 < \alpha < 1$  and from  $l-x$  : a return of  $v(l-x)$  such that they are left with a lesser amount  $\beta(l-x)$  :  $0 < \beta < 1$ . So the total amount left after the first step of division is  $\alpha x + \beta(l-x)$  and the process continues. Devise the partition strategy to help them maximize the return obtained in a finite  $n$  or infinite number of steps.*  $\diamond$

**§§ Solution.** Let  $y(x)$  denote the return after the first step:

$$\therefore y(x) = u(x) + v(l-x)$$

Assuming  $u$  and  $v$  to be continuous functions, it is trivial to find the maximum of  $y(x)$  over  $x \in [0, l]$  using calculus (or graphical approach) :

$$\frac{dy}{dx} = \frac{d}{dx}u(x) + \frac{d}{dx}v(l-x) = 0 \text{ (for extrema).}$$

Solve for  $x$  and  $y(x)$  is maximum for that  $x$  for which  $\frac{d^2y}{dx^2} < 0$ .

Suppose  $u(x) = x$  and  $v(l-x) = -(l-x)^2$ , then

$$\begin{aligned} y &= x - (l-x)^2 \\ \therefore \frac{dy}{dx} &= 1 + 2(l-x) = 0, \end{aligned}$$

$$\begin{aligned}\therefore x &= l + \frac{1}{2} \\ \frac{d^2y}{dx^2} &= -2 < 0. \\ \therefore y_{max} &= l + \frac{1}{2} - \frac{1}{4} = l + \frac{1}{4}.\end{aligned}$$

After the first step, the initial amount  $l$  is reduced to  $l_1$  (say):

$$\therefore l_1 = \alpha x + \beta(l - x)$$

In the second step,  $l_1$  is partitioned into  $x_1$  (say) and  $(l_1 - x_1)$  for  $0 \leq x_1 \leq l_1$ . Hence, the return from the second step is  $u(x_1) + v(l_1 - x_1)$ . Therefore, the total return after the two steps is:

$$\therefore y(x, x_1) = u(x) + v(l - x) + u(x_1) + v(l_1 - x_1).$$

Maximum of the function  $y(x, x_1)$  over the 2-dimensional space  $(x, x_1)$  yields the maximum return, such that  $x \in [0, l]$  and  $x_1 \in [0, l_1]$ .

Similarly, the total return after  $n$  steps is :

$$\therefore y(x, x_1, x_2, \dots, x_{n-1}) = u(x) + v(l - x) + \sum_{i=1}^{n-1} [u(x_i) + v(l_i - x_i)]. \quad (21.1)$$

Here  $x_i \in [0, l_i]$ .

Using this *enumerative* approach to maximize the  $n$ -dimensional return, the computation procedure soon becomes cumbersome, error-prone and exponential in nature.

Any choice of  $x, x_1, x_2, \dots$  is a *policy*.

The policy maximizing  $y(x, x_1, x_2, \dots)$  is an *optimal policy*.

It can be noted that each step depends on the respective policy only. Hence at the  $(i + 1)^{th}$  step, the corresponding *one-dimensional* choice is made : a choice of  $x_i \in [0, l]$ .

Hence an optimal policy leads to the corresponding maximum return.

Let  $y_n(l)$  denote the maximum total return, given the initial amount  $l$  and  $n$  steps.

$$\therefore y_1(l) = \text{Max}_{x \in [0, l]} [u(x) + v(l - x)].$$

After the first step,  $l$  becomes  $\alpha x + \beta(l - x)$  :

$$\therefore y_2(l) = \text{Max}_{x \in [0, l]} [u(x) + v(l - x) + y_1(\alpha x + \beta(l - x))].$$

This leads to a recurrence relation :

$$\therefore y_n(l) = \text{Max}_{x \in [0, l]} [u(x) + v(l - x) + y_{n-1}(\alpha x + \beta(l - x))]. \quad (21.2)$$

Hence a single  $n$ -dimensional problem is reduced to a sequence of  $n$  one-dimensional problems.

Here, the optimal return depends on the initial amount  $l$  and initial decision of division into the parts  $l$  and  $l - x$  only.

This is possible due to **the Principle of Optimality** :

*An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*

Hence Eq. (21.2) is the required optimal strategy. ■

**Excerpt from the Chapter (Constrained Subsequence):**

## Maximum Sum

**§ Problem.** Given a sequence of  $n \in (-\infty, \infty)$  integers, determine the largest possible sum of the contiguous subsequence.

◇

**§§ Solution.** Let  $f_n(i)$  be the maximum sum of a contiguous subsequence ending at index  $i$ , obtained using an optimal policy and  $n$  steps.

Let  $s_i$  be the value of the element at index  $i$ , i.e.  $s_i$  is used at the  $n^{th}$  step. Then we can use an optimal policy starting with previously accumulated maximum sum of a contiguous subsequence ending at index  $i - 1$ .

Hence the required optimal procedure is

$$\therefore f_n(i) = \text{Max}_{i \in [0, n-1]} [f_{n-1}(i-1) + s_i]$$

At each step (with addition of  $s_i$ ), there are 2 options :

1. leverage the previous accumulated maximum sum if  $f_{n-1}(i-1) + s_i > 0$ , because it is better to continue with a positive running sum or
2. start afresh with a new range (with the starting sum as 0) if  $f_{n-1}(i-1) + s_i < 0$ , because it is better to start with 0 than continuing with a negative running sum.

Also note that:

- If all the elements are negative, then there is no such subsequence, i.e. the required sum is 0.
- If all the elements are positive, then the entire sequence is the required subsequence, i.e. the required sum is the sum of all the elements of the sequence.
- The required subsequence (if any) starts at and ends with a positive value.

Time complexity is  $\mathcal{O}(n)$ . Space complexity is  $\mathcal{O}(1)$ .

```
int maxseq(std::vector<int> & s)
{
    int current_sum = 0;
    int max_sum = 0;

    for(int x : s)
```



### Maximum sum contiguous subsequence : compute sum

```

1: function maxseq( $s[0..n-1]$ )
2:    $currentsum \leftarrow 0$ 
3:    $maxsum \leftarrow 0$ 
4:   for  $x \in s[0..n-1]$  do
5:      $currentsum \leftarrow \mathbf{max}(currentsum + x, 0)$ 
6:      $maxsum \leftarrow \mathbf{max}(maxsum, currentsum)$ 
7:   end for
8:   return  $maxsum$ 
9: end function

{
    current_sum = std::max(current_sum + x, 0);
    max_sum = std::max(max_sum, current_sum);
}
return max_sum;
}

```

■

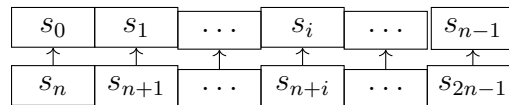
## Circular Sequence

**§ Problem.** Given a circular sequence  $s$  of  $n \in (-\infty, \infty)$  integers, find the maximum possible sum of a non-empty contiguous subsequence of  $s$ . ◇

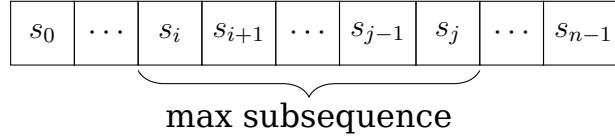
**§§ Solution.** The end of a circular sequence wraps around the start of the sequence itself, i.e.

$$\therefore i \equiv (i + n) \bmod n \quad \forall i \in [0, n)$$

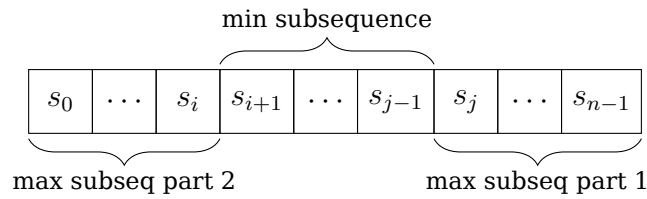
$$\therefore s_i \equiv s_{(i+n) \bmod n} \quad \forall i \in [0, n).$$



For a maximum contiguous subsequence  $[s_i \dots s_j]$ , the solution of Dialogue 21 can be used.



For a maximum contiguous subsequence  $[s_j \cdots s_{n-1}, s_0 \cdots s_i]$ , the left-over part  $[s_{i+1} \cdots s_{j-1}]$  forms a minimum contiguous subsequence.



Summation of the contiguous subsequence  $[s_j \cdots s_{n-1}, s_0 \cdots s_i]$  is

$$\begin{aligned}
 &= s_j + \cdots + s_{n-1} + s_0 + \cdots + s_i \\
 &= s_0 + \cdots + s_{n-1} - [s_{i+1} + \cdots + s_{j-1}]
 \end{aligned}$$

This is maximum when  $[s_{i+1} + \cdots + s_{j-1}]$  is minimum.

$$\therefore \text{Max}[s_j + \cdots + s_{n-1} + s_0 + \cdots + s_i] = \sum_{k=0}^{k=n-1} s_k - \text{Min} \sum_{k=i+1}^{k=j-1} s_k$$

$\therefore$  Maximum sum subsequence = Total sum of the sequence  
 – Minimum sum subsequence

Time complexity is  $\mathcal{O}(n)$ . Space complexity is  $\mathcal{O}(1)$ .

```

int maxsum_circular(std::vector<int> & s)
{
    int current_max = 0, max_sum = std::numeric_limits<int>::min();
    int current_min = 0, min_sum = std::numeric_limits<int>::max();
    int total_sum = 0;

    for(int x : s)
    {
        current_max = std::max(current_max + x, x);
    }
  
```

### Maximum sum circular subsequence

```

1: function maxcircularseq( $s[0..n-1]$ )
2:    $currentmax \leftarrow 0$ 
3:    $maxsum \leftarrow -\infty$ 
4:    $currentmin \leftarrow 0$ 
5:    $minsum \leftarrow \infty$ 
6:    $totalsum \leftarrow 0$ 

7:   for  $x \in s[0..n-1]$  do
8:      $currentmax \leftarrow \mathbf{max}(currentmax + x, x)$ 
9:      $maxsum \leftarrow \mathbf{max}(maxsum, currentmax)$ 

10:     $currentmin \leftarrow \mathbf{min}(currentmin + x, x)$ 
11:     $minsum \leftarrow \mathbf{min}(minsum, currentmin)$ 

12:     $totalsum \leftarrow totalsum + x$ 
13:   end for

14:   if  $totalsum == minsum$  then           ▷ All elements are -ve
15:     return  $maxsum$                      ▷ Value of the least -ve element
16:   else
17:     return  $\mathbf{max}(maxsum, totalsum - minsum)$ 
18:   end if
19: end function

```

```

max_sum = std::max(max_sum, current_max);

```

```

current_min = std::min(current_min + x, x);
min_sum = std::min(min_sum, current_min);

```

```

total_sum += x;

```

```

}
// when all elements are -ve => total_sum == min_sum,
// i.e. total_sum - min_sum becomes 0 => empty subsequence
// but max_sum still holds the value of the least -ve element,
// hence return this singleton than an empty one

```

```
}    return total_sum == min_sum ? max_sum : std::max(max_sum, total_sum);
```

■

## Brief Table of Contents

1. Genesis
  - a) Optimal Loot Partition
    - i. Deterministic
    - ii. Stochastic
  - b) Exam Prep
  - c) Optimal Coin Tossing
  - d) Proving Optimality Principle
2. Computation
  - a) Ascension to Heaven
  - b) Fibonacci Line Search
  - c) Coin Change
  - d) Constrained Subsequence
    - i. Maximum Sum
    - ii. Minimum Sum
    - iii. Circular Sequence
    - iv. Maximum Product
  - e) Stock Trading
  - f) Binary Tree Mall Loot
  - g) Binary Search Tree Generation
  - h) Quantify Yogic Effect
  - i) Path to Heaven
    - i. Stairway
    - ii. Kriya Grid
  - j) Kriya Sequence
  - k) Kriya Catalysis

## List of Algorithms/Programs

1. Minimum Coin Change : Iterative (Bottom-up) Approach
2. Minimum Coin Change : Recursive (Top-down) Approach
3. Minimum Coin Change : Optimal set of coins
4. Coin Change : No of Ways
5. Maximum sum contiguous subsequence : compute sum
6. Maximum sum contiguous subsequence : compute indices
7. Maximum sum non-contiguous subsequence : compute sum
8. Maximum sum non-contiguous subsequence : compute sum : space optimized
9. Minimum sum contiguous subsequence .
10. Min sum contiguous subsequence : Find max of -ve
11. Minimum sum contiguous subsequence : compute indices
12. Maximum sum circular subsequence
13. Minimum sum circular subsequence
14. Maximum product contiguous subsequence : compute product
15. Maximum product contiguous subsequence : compute product : modified
16. Stock Trading : Maximum Profit : One Transaction
17. Maximize Profit : Maximum sum contiguous subsequence
18. Maximize Profit : Buy and Sell Days
19. Stock Trading : Maximum Profit : Two Transactions
20. Stock Trading : Maximum Profit :  $m(< n)$  Transactions
21. Stock Trading : Maximum Profit :  $m(> n)$  or Unlimited Transactions
22. Stock Trading : Maximum Profit :  $m(> n)$  or Unlimited Transactions : Alternative
23. Count Unique BSTs
24. Generate Unique BSTs
25. Quantify Yogic Effect : Drink Air Therapy
26. Quantify Yogic Effect : Khechari Kriya
27. Quantify Yogic Effect : Mool Kriya
28. Quantify Yogic Effect : Tandav Kriya
29. Quantify Yogic Effect : Minimax Kriya Selection .
30. Quantify Yogic Effect : Minimax Kriya Selection : Optimized Computation

31. Quantify Yogic Effect : Trikaldarshi
32. Quantify Yogic Effect : Trikaldarshi : Print Kriya Triangles
33. Staircase to Heaven : Count Distinct Ways
34. Staircase to Heaven : Count Distinct Ways with step-list
35. Staircase to Heaven : Optimal Pranayams
36. Distinct Kriya Grid Paths to Heaven
37. Distinct Kriya Grid Paths to Heaven : Space Optimization
38. Distinct Kriya Grid Paths to Heaven : With Prohibition
39. Distinct Kriya Grid Paths to Heaven : With Prohibition :  
Space Optimization
40. Distinct Kriya Grid Paths to Heaven : With Prohibition :  
Space Optimization : Alternative
41. Kriya Grid Paths to Heaven : Optimal Pranayams
42. Constrained Kriya Grid Paths to Heaven : Optimal Pranayams
43. Constrained Kriya Grid Paths to Heaven : Optimal Pranayams  
: Diff Cols
44. Constrained Kriya Grid Paths to Heaven : Optimal Pranayams  
: Diff Cols : Optimized
45. Optimal Pranayams to reach Heaven
46. Count ways : First Kriya
47. Count ways : First Kriya : Space Optimization
48. Out of Kriya Grid : Count ways
49. Out of Kriya Grid : Count ways : Space Optimization
50. Triangular Kriya Grid : Optimal Pranayams
51. Triangular Kriya Grid : Optimal Pranayams : Alternative
52. Maximal Square Kriya Grid
53. Max Zeroness Kriya Sequences
54. Perfect Kriya
55. Generate Kriya
56. Vanish Kriya
57. Split Kriya
58. Threshold Kriya
59. Threshold Kriya : Space Optimization
60. Rejuvenate Kriya
61. Rejuvenate Kriya : Space Optimization
62.  $\beta$ -Dimensional Kriya
63.  $\beta$ -Dimensional Kriya : Space Optimization
64. Kriya Moves

65. Marking Kriya
66. Marking Kriya : Space Optimization
67. Kriya Selection
68. Kriya Sets : Possible Moves
69. Kriya Sets : Space Optimization
70. Count Distinct Pranayams Sets
71. Partition Kriya : Iso-Pranayams Sets
72. Partition Kriya : Iso-Pranayams Sets : Space Optimization
73. Kriya Probability
74. Combine Kriya
75. Sort Kriya : Optimal Interchanges
76. Sort Kriya : Space Optimization
77. Longest Increasing Subsequence (LIS) of Kriyas
78. Permute Kriyas
79. Length of LCS Kriya
80. LCS Kriya
81. Compute and Print LCS Kriya
82. Compute and Print LCS Kriya : Alternative
83. Compute All The LCS Kriya
84. Length of LCS Kriya : Space Optimization
85. Length of SCS Kriya
86. Reconstruction of SCS Kriya from Optimal Solution
87. Print SCS : Recursive Approach
88. Compute All The SCS Kriya
89. Computation SCS from LCS Kriya
90. SCS Kriya : Alternative Solution from LCS
91. Counting Palindromic Kriya Contiguous Subsequence
92. Longest Palindromic Kriya Contiguous Sub sequences
93. Maximum Length of Palindromic Kriya Subsequence
94. Max Length of Palindromic Kriya Subsequence : Alternative
95. Maximum Length of Palindromic Kriya Subsequence : Space Optimization
96. Max Length of Palindromic Kriya Subsequence : Space Optimization : Alternative
97. Count of Distinct Kriya Subsequences
98. Count of Distinct Kriya Subsequences : Space Optimization

99. Transform Kriya
100. Print Transformation Path
101. Transform Kriya : Space Optimization
102. Print Operations
103. Reconstruct Operations
104. Transform Kriya and Reconstruct Operations
105. Print Operations
106. Reconstruct Operations
107. Transform Kriya and Reconstruct Operations
108. Transform Kriya : Unrestricted Operations
109. Edit Distance : Print Operations with Copy and Finish
110. Edit Distance : Print Custom Operations with Reconstruct Operations
111. Edit Distance : Transform Kriya and Reconstruct Custom Operations
112. Reconstruct and Print Aligned Kriya Sequences
113. Generate Aligned Kriya Sequences
114. Generate & Reconstruct Aligned Kriya Sequences
115. Identical Kriya Sequences
116. Identical Kriya Sequences with Reconstruction
117. Identical Kriya Sequences : Reconstruction (Recursive)
118. Generate Identical Kriya Sequences with Reconstruction (Recursive)
119. Generate Identical Kriya Sequences : Optimal Space
120. Optimal Removed Kriyas
121. Kriya Sequence Generation : Count Ways : Constraints of Favourable Comparisons
122. Preferred Kriya Practice : Count Ways
123. Preferred Kriya Practice : Count Ways : Space Optimization
124. Binary Split Kriyas : Count Ways
125. Organize Kriyas : Ways of Non-adjacent ones
126. Select Kriyas Alternately : Optimal Difference
127. Decode Kriya Sequence from Digits Sequence
128. Sorted Kriya Sequence : Transduction Quotient
129. Cross Kriya Potential
130. Maximum Sum : Linear and Circular Kriya Sequence



Monograph

22

# **Hacking TensorFlow Internals : An Insider's Commentary on A Learning System**

Monograph

23

# Advanced C++ FAQs Vol 1 & 2

Monograph

24

## C++14 FAQs



# **The Boost C++ Libraries: Generic Programming**

Monograph

26

# **Generic Algorithms and Data Structures using C++11**

Monograph

27

# **C++11 Standard Library: Usage and Implementation**

Monograph

28

# Foundation of Algorithms in C++11

# **C++11 Algorithms : Using and Extending C++11, Boost and Beyond**





# **Cracking Programming Interviews : 500 Questions with Solutions**



# **Top 20 Coding Interview Problems Asked in Google with Solutions**



# **Top 10 Coding Interview Problems Asked in Google with Solutions**

# **Part II**

# **Physics**

PHY

**Part III**

**Mathematics**

MATH