ॐ

प्राचीन विज्ञान

ॐ मण्डल ॐ

Ancient Science Publishers

Decode The Myth

अन्तर्चक्षुः

भगवान् महर्षि हिरण्यगर्भ

Beacons of Light

Edsger Wybe Dijkstra

Richard Phillips Feynman

Leonhard Euler

Artifacts

Monographs

ancientsciencepublishers@gmail.com

Catalogue 2022

Sl.	Title	India	Others
1	Conceptual Kinematics: A Companion to I. E. Irodov's Problems in General Physics	₹250	$9.95
2	Conceptual Geometry of Straight Line : A Companion to S. L. Loney's Co-ordinate Geometry	₹350	$19.95
3	Conceptual Trigonometry Part I : A Companion to S. L. Loney's Plane Trigonometry Part I	₹500	$39.95
4	Solutions of the Examples in Higher Algebra : H. S. Hall, S. R. Knight	₹400	$29.95
5	Problems and Solutions in Plane Trigonometry : Isaac Todhunter	₹500	$39.95
6	Concepts, Problems and Solutions in School Calculus : A Dialogue Approach	₹500	$19.95
7	Solving Problems using Dynamic Programming : A Hacker's Perspective	₹500	$29.95
8	Advanced C++ FAQs	₹500	$29.95
9	C++14 FAQs	₹200	$8.95
10	Cracking Programming Interviews : 500 Questions with Solutions	₹500	$29.95
11	Top 20 Coding Interview Problems Asked in Google with Solutions	₹250	$9.95
12	Elements of C++20	₹2000	$39.95

Flat 20% Off, Bulk Discount 30-50%

email : ancientsciencepublishers@gmail.com

Monographs

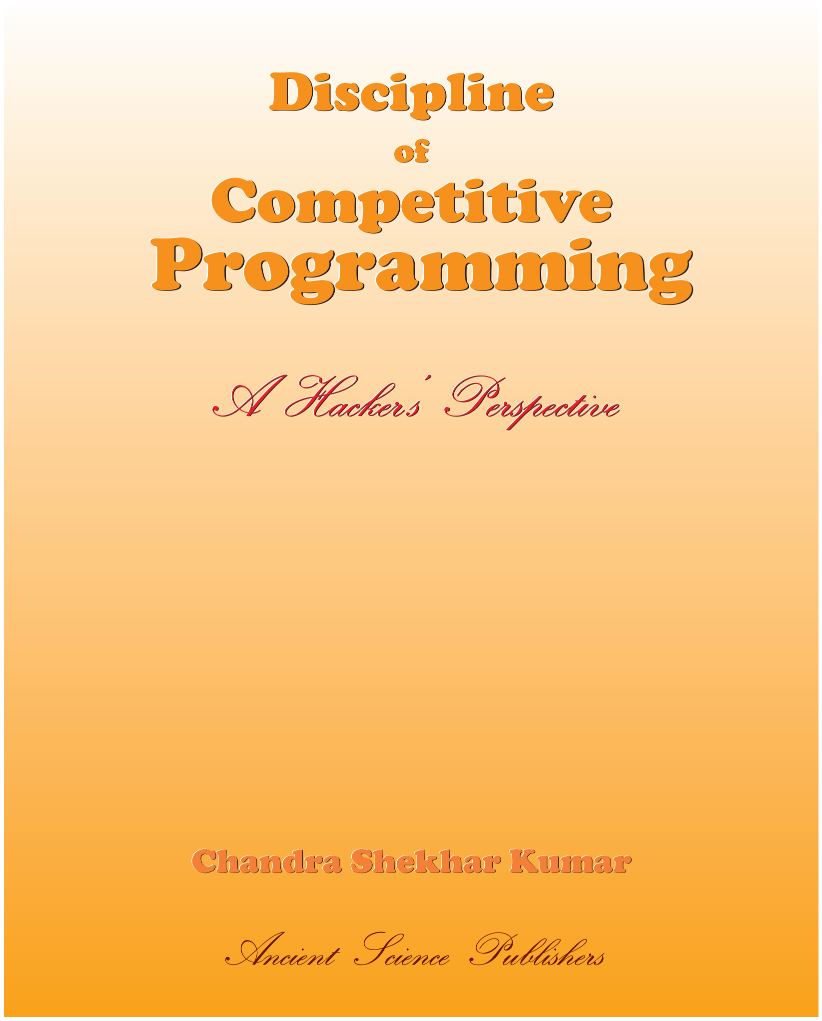
Part I

Computer Science

Monograph

1

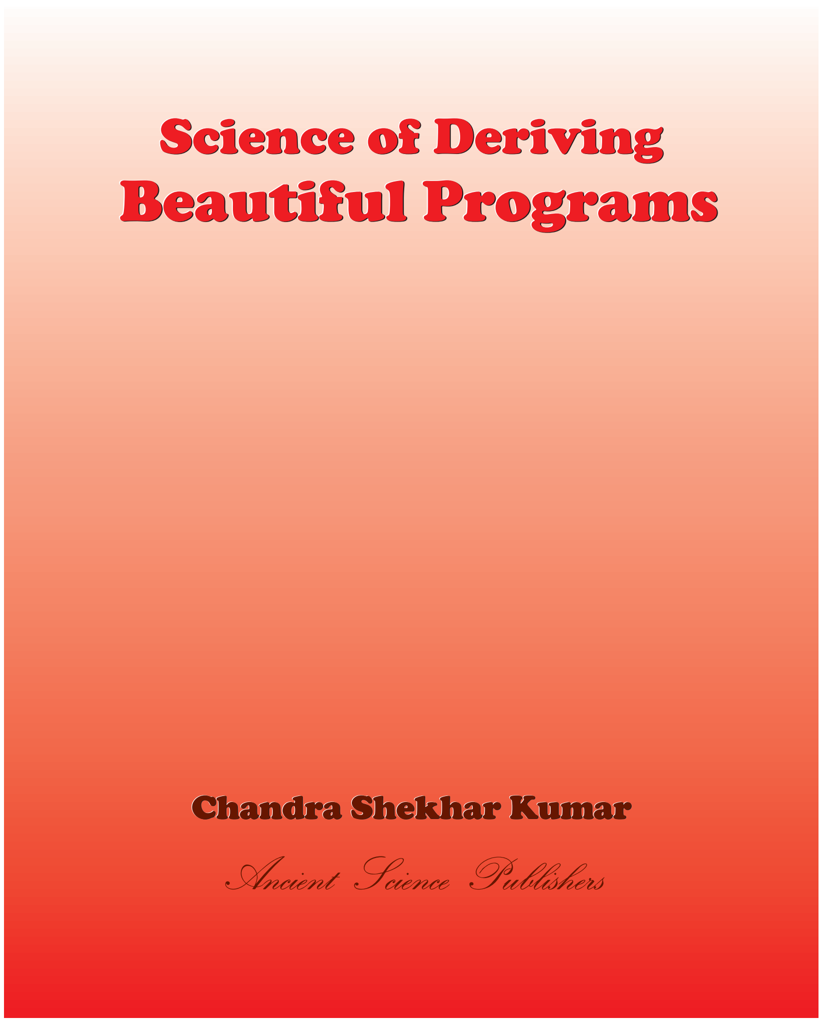
Discipline of Competitive Programming : A Hacker’s Perspective



Monograph

2

Elements of Coding : Science of Deriving Correct Programs



Monograph 3

Elements of Coding Linear Algebra : The Nucleus of Artificial Intelligence

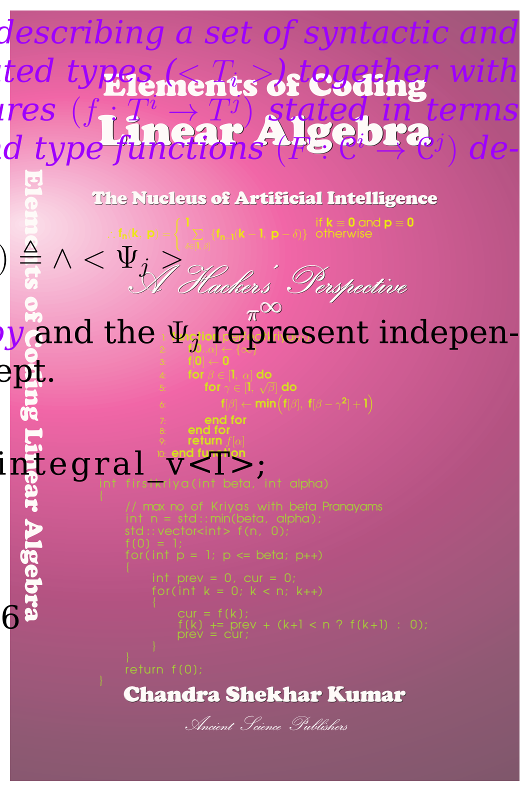
Excerpt from the Chapter Algebraic Concepts

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

$$\therefore \mathcal{C}\,(< T_i >) \triangleq \wedge < \Psi_j >$$

where $\triangleq$ stands for *is defined by* and the Ψ_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. $T \models \mathcal{C}$.

int8_t and uint8_t $\models$ 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* $\mathcal{C}^j$ ($\mathcal{C}^i \looparrowright \mathcal{C}^j$) by addition of more requirements to $\mathcal{C}^j$, i.e. $\mathcal{C}^j$ *weakens* $\mathcal{C}^i$ ($\mathcal{C}^j \looparrowleft \mathcal{C}^i$).

```
template<class T>
    concept signed_integral = integral<T> && is_signed_v<T>;
```

signed_integral $\looparrowright$ integral

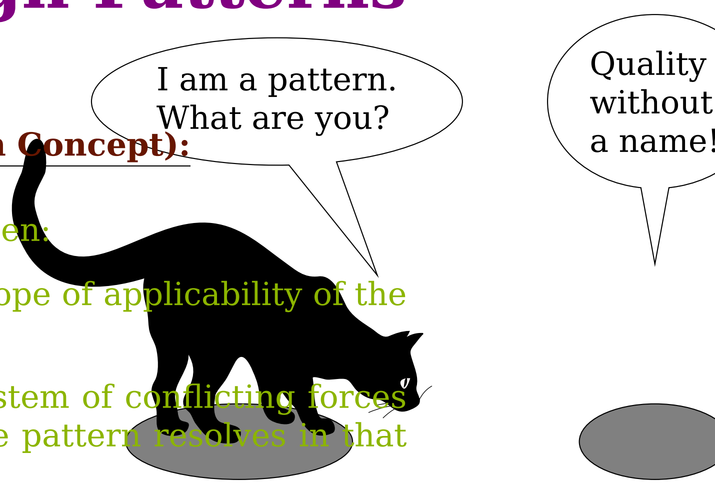
int8_t $\models$ signed_integral

```
template<class T>
concept unsigned_integral = integral<T> && !signed_integral<T>;
```

unsigned_integral $\looparrowright$ integral

uint8_t $\models$ unsigned_integral

Elements of Software Design Patterns

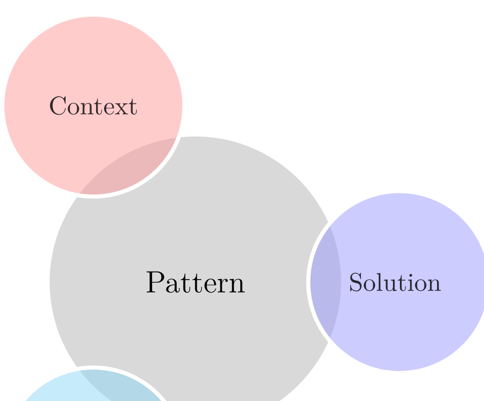


Excerpt from the Chapter (Pattern Concept):

Definition. A pattern is a rule triad expressing a relation between:

1. a certain *context* defining the scope of applicability of the given pattern,
2. a *problem* detailing a certain system of conflicting forces which occurs repeatedly that the pattern resolves in that context and
3. a *solution* in a form of a certain software configuration which can be used repeatedly and uniquely to resolve the given system of forces themselves, wherever the context makes it relevant.

The output of this rule triad is a pattern too.



It leads (but not limited) to the following key observations about pattern:

- → It is both a thing and a process.
- → It is both a description of a thing which is alive and a description of the process which will generate that thing.
- → It is both a thing which happens in the world and the rule which tells us how to create that thing.
- → It can exist at all scales and resolve almost any kind of conflicting forces.
- → Identification of what-why-when-where marks its inner structure explicit and sharable.
- → It starts with defining features worth abstracting.
- → Then it defines the problem, i.e. the field of forces which it brings into balance.
- → It is a sketch rather than a blue-print.
- → It can complement and compound another pattern(s).

- It is generative and self-sustaining.
- It is a micro-architecture.
- It promotes design-reuse.
- The exact range of contexts is defined where the stated problem occurs and where this particular solution to the problem is appropriate.
- Each pattern describes a problem which occurs over and over again in our system and then describes the core of the solution to that problem in such a way that we can use this solution a million times over, without ever doing it the same way twice.

Beyond its elements, each system is defined by a certain patterns of relationships among the elements, and these relationships are integral part of the elements to such an extent that the elements themselves are patterns of relationships. And finally, the so called elements get dissolved, leaving patterns of relationships behind, which is the actual thing that actually repeats itself and gives structure to the system.

Each one of these patterns $\mathcal{P}_i$ is a morphological law onto itself, which establishes a set of relationships in the system in a given context of type $\mathcal{C}$, i.e.

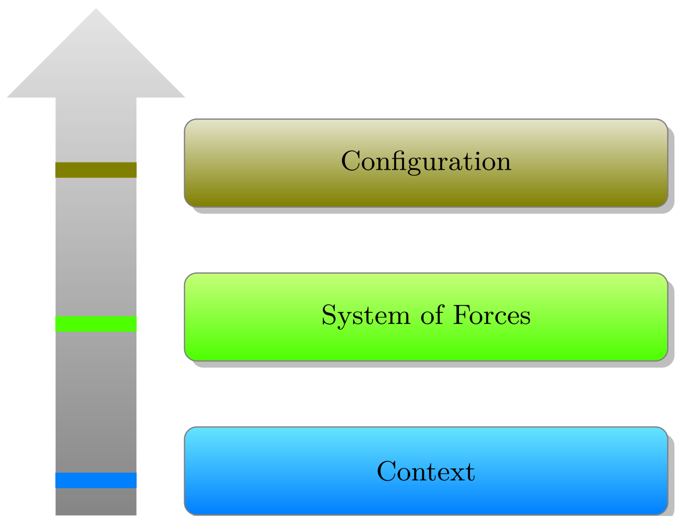
$$\mathcal{P}_i \triangleq \mathcal{C} \rightarrow \mathcal{R}\,(\ldots, \mathcal{P}_{i-1}, \mathcal{P}_{i+1}, \ldots)$$

where $\triangleq$ stands for *is defined by*. The parts (i.e, rest of the patterns except $\mathcal{P}_i$) $\ldots, \mathcal{P}_{i-1}, \mathcal{P}_{i+1}, \ldots$ are related by the relationship $\mathcal{R}$ within a context of type $\mathcal{C}$.

Note that, each law or pattern is itself a pattern of relationships among the remaining laws (i.e. except itself), which are themselves just patterns of relationships again.

Therefore, a pattern is defined by formulating it in the form of a rule triad as depicted before, which establishes a relationship between a context, a system of (often conflicting) forces which arise in that context and configuration which allows these forces to resolve themselves in that context.

Hence, generic form of each pattern is:



Discovery of (the invariant features) pattern(s) always start with observation or purely abstract argument. This process is not sequential from the problem to the solution or vice versa. Rather it is a multidimensional global process to help identify a solid and reliable invariant which relates context, problem, solution in an unchanging way.

The statement of the problem and the forces helps to solidify the pattern which is responsible for making the system of forces come to an equilibrium. Thought it is still tentative, but clear enough to be shared.

There are two components in a pattern definition, which are empirical in nature, i.e. can be tested as true/false:

1. The problem is real, i.e. it is expressible as conflicting real forces within the stated context(s).

2. The configuration solves the problem, i.e. it deals with all the forces in the stated context(s).

Quality without a name is the living essence of a pattern.

Excerpt from the Chapter (Pattern Form):

Each (living) pattern has the same form for the sake of convenience and clarity. It has *nine* parts in the following sequence :

1. A *picture* is drawn to illustrate an archetypal example of the pattern.

2. An *introductory paragraph* to set the context for the pattern.

3. The symbol ॐ marks the beginning of the problem the pattern addresses later.

4. A **headline** set in bold-typeface to provide the essence of the problem.

5. The *body* of the problem describing (but not limited to) the
   - → empirical background of the pattern,
   - → empirical evidence for its validity which sets the motivational tone too,
   - → variations, i.e. the range of different ways of manifesting it in a software.

6. The **solution** set in bold-typeface, encoded in an instructional form, stating the exact steps to build the pattern. It illustrates the field of relationships needed to solve the stated problem in the stated context.

Do what you can to establish coherence in your software. I am smart because I do nothing!

7. A *diagram* that shows the solution as a labeled picture indicating its main components.

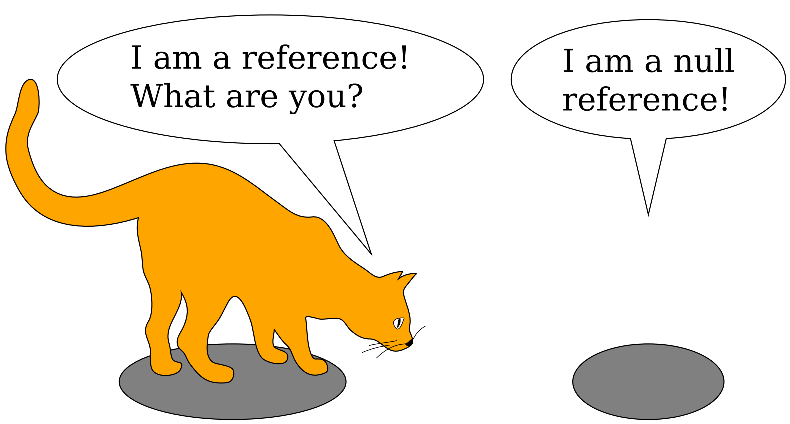
8. The symbol ॐ marking the end of the main body of the pattern.

9. A *paragraph*, which ties the pattern to all those smaller patterns in the pattern language, which are needed to complete this pattern, to embellish it, to fill it out.

This form serves the following two essential purposes :

1. to present each pattern connected to other patterns to help grasp the collection of all these patterns as a whole, as a pattern language, within which an infinite variety of combinations can be created.
2. to present the problem and solution of each pattern in such a way that it sets the exact tone of self-judgment and modifications without losing the central essence.

Excerpt from the Chapter (Null Object):



··· consider now the character of settlements within the object references : what balance of real objects and null references is in keeping with the transparency ?

ॐ

> **Optionally null object references, where the result of a null check is to do nothing, will not come to balance until both the presence of a null reference and the absence of an object be treated in a consistent and transparent manner to establish an independent and coherent sphere of object references.**

Out of a list of objects, some may not exist. Hence no service is expected in such cases which can be an acceptable behavior too. Acceptable inaction is represented at times with repetitive explicit checking for the optional null. Repetition and optional doesn't go together. Absence of objects can be abstracted out to presence of objects doing nothing, i.e. conformance to the interface with no implied functionality. No-op is the correct operation. We need a way to represent the object with appropriate behavior that will allow us to treat all object references in a consistent and uniform way, devoid of special case consideration.

Typical scenarios under consideration are

1. Some object instances are not required to do anything because they correspond to null references.

2. These instances should be treated in the same manner as real instances to avoid explicit constraints.

3. There is a need to reuse the do nothing behavior to enforce consistent and repetitive usage.

Null Object patterns addresses all of these under a single umbrella, typically by encapsulating the do nothingness.

Monograph 5

Elements of Coding AI

Monograph 6

Elements of Coding DL (Deep Learning)

Monograph 7

Elements of Coding ML : Internals of Machine Learning Library MLPack

Monograph

8

Conceptual BitCoin : Blockchain Coding

Monograph 9

Conceptual Data Science Interviews

Monograph

10

Conceptual Dependency Injection : Unwiring Simplified in C++

Monograph

11

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

Monograph 16

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

Monograph

20

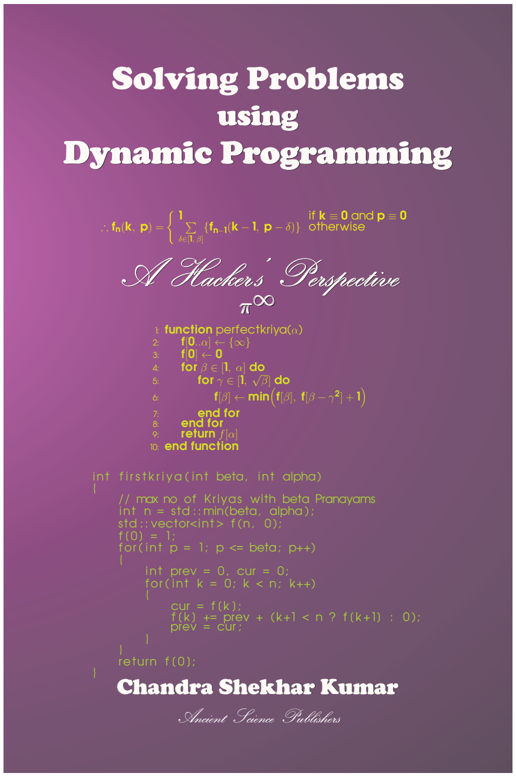
Elements of C++20

Monograph

21

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 π^{∞}.



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 internals of underlying concepts and beyond to computer scientists alike.

Ancient Science Publishers — *Chandra Shekhar Kumar*
July, 2020. — 256 pages — 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 αx : $0 < \alpha < 1$ 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.* ♢

§§ 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

$$y = x - (l - x)^2$$

$$\therefore \frac{dy}{dx} = 1 + 2(l - x) = 0,$$

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

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 $0 \le x_1 \le l_1$. 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 :

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

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 $x, x_1, x_2, \ldots$ is a *policy*.
The policy maximizing $y(x, x_1, x_2, \ldots)$ 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.

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

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

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

This leads to a recurrence relation :

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

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. The 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

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

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

```
function maxseq(s[0..n − 1])
    currentsum ← 0
    maxsum ← 0
    for x ∈ s[0..n − 1] do
        currentsum ← max(currentsum + x, 0)
        maxsum ← max(maxsum, currentsum)
    end for
    return maxsum
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.

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

s_0	s_1	$\cdots$	s_i	$\cdots$	s_{n-1}
$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
s_n	s_{n+1}	$\cdots$	s_{n+i}	$\cdots$	s_{2n-1}

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

$$\boxed{s_0} \boxed{\cdots} \underbrace{\boxed{s_i} \boxed{s_{i+1}} \boxed{\cdots} \boxed{s_{j-1}} \boxed{s_j}}_{\text{max subsequence}} \boxed{\cdots} \boxed{s_{n-1}}$$

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.

$$\underbrace{\boxed{s_0} \boxed{\cdots} \boxed{s_i}}_{\text{max subseq part 2}} \overbrace{\boxed{s_{i+1}} \boxed{\cdots} \boxed{s_{j-1}}}^{\text{min subsequence}} \underbrace{\boxed{s_j} \boxed{\cdots} \boxed{s_{n-1}}}_{\text{max subseq part 1}}$$

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.

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

$$\begin{aligned} \therefore \text{Maximum sum subsequence } &= \text{ Total sum of the sequence} \\ &- \text{ Minimum sum subsequence} \end{aligned}$$

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

```
function maxcircularseq(s[0..n − 1])
    currentmax ← 0
    maxsum ← −∞
    currentmin ← 0
    minsum ← ∞
    totalsum ← 0

    for x ∈ s[0..n − 1] do
        currentmax ← max(currentmax + x, x)
        maxsum ← max(maxsum, currentmax)

        currentmin ← min(currentmin + x, x)
        minsum ← min(minsum, currentmin)

        totalsum ← totalsum + x
    end for

    if totalsum == minsum then            ▷ All elements are -ve
        return maxsum                     ▷ Value of the least -ve element
    else
        return max(maxsum, totalsum − minsum)
    end if
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_su
}
```

■

Brief Table of Contents

List of Algorithms/Programs

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 Zerones 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. β-Dimensional Kriya
63. β-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

Monograph 25

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

Monograph 29

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

Monograph

30

Cracking Programming Interviews : 500 Questions with Solutions

Monograph

31

Top 20 Coding Interview Problems Asked in Google with Solutions

Monograph 32

Top 10 Coding Interview Problems Asked in Google with Solutions

Part II

Physics

PHY

Part III

Mathematics

MATH