ॐ

प्राचीन विज्ञान

ॐ ॐ

मण्डल

Ancient Science Publishers

Decode The Myth

अन्तर्चक्षुः

भगवान् महर्षि हिरण्यगर्भ

Beacons of Light

Edsger Wybe Dijkstra

Richard Phillips Feynman

Leonhard Euler

Artifacts

Monographs

Monographs

Part I

Computer Science

Monograph 1

Discipline of Competitive Programming : A Hacker's Perspective

$x^2 \triangleq$

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 $(f : T^i \rightarrow T^j)$ stated in terms of the properties, attributes and type functions $(F : \mathcal{C}^i \rightarrow \mathcal{C}^j)$ defined on the types.*

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

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 $\mathfrak{C}$, then T ***models*** $\mathfrak{C}$, i.e. $T \vDash \mathfrak{C}$.

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

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

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

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

$$\text{signed\_integral} \looparrowright \text{integral}$$
$$\text{int8\_t} \vDash \text{signed\_integral}$$

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

$$\text{unsigned\_integral} \looparrowright \text{integral}$$
$$\text{uint8\_t} \vDash \text{unsigned\_integral}$$

Monograph 4

Elements of Software Design Patterns

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

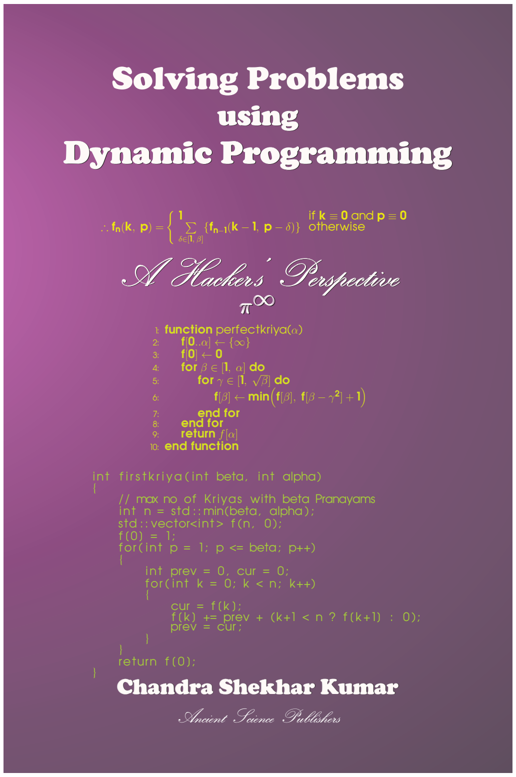
Elements of C++20

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 $\dfrac{d^2y}{dx^2} < 0$.

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 $0 \leq x_1 \leq l_1$. 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 :

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

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($s[0..n-1]$)
  $currentsum \leftarrow 0$
  $maxsum \leftarrow 0$
  for $x \in s[0..n-1]$ **do**
    $currentsum \leftarrow$ **max**$(currentsum + x, 0)$
    $maxsum \leftarrow$ **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.

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

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.

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

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

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 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

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