Conceptual Dynamic Programming

Optimal Coding Simplified

By

Chandra Shekhar Kumar
Integrated M. Sc. in Physics, IIT Kanpur, India
Co-Founder, Ancient Science Publishers
Founder, Ancient Kriya Yoga Mission

Ancient Science Publishers

For comments, suggestions and or feedback, send mail to :
ancientsciencepublishers@gmail.com

Preface

Programming is fun though largely intuitive and less predictable ! Conceptual Programming is more fun and predictable, hence more reliable !!

Includes 100+ solved problems !!!

This book provides conceptual approach to understanding, applying and extending the computational aspect of dynamic programming. This leads to simplification of optimal coding paradigm and enforces correctness as a part of program design with great emphasis on foundational science of dynamic Programming. Familiarity with elementary calculus and probability is assumed. Though aimed primarily at programmers, it imparts the knowledge of deep internals of underlying mathematical concepts to teachers alike.

Care has been taken, as in the forthcoming ones, to present the solutions with multi-concepts and beyond in a simple natural manner, in order to meet the difficulties which are most likely to arise, and to render the work intelligible and instructive.

This work contains several variations of problems, solutions, methods, approaches to enrich, strengthen and enliven the inherent multi-concepts.

Ancient Science Publishers Chandra Shekhar Kumar
February, 2019.

List of Chapters

Chapter 1

Genesis

1.1 Optimal Loot Partition

1.1.1 Deterministic

§ Problem 1.1.1. 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

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

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

$$\therefore x = l + \frac{1}{2}.$$

$$\frac{d^2y}{dx^2} = -2 < 0.$$

$$\therefore y_{max} = l + \frac{1}{2} - \frac{1}{4} = l + \frac{1}{4}.$$

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 :

$$\therefore y(x, x_1, x_2, \ldots, x_{n-1}) = u(x) + v(l - x) + \sum_{i=1}^{n-1} [u(x_i) + v(l_i - x_i)] \,. \tag{1.1}$$

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.

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

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

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

This leads to a recurrence relation :

$$\therefore y_n(l) = \operatorname*{Max}_{x \in [0,l]} [u(x) + v(l - x) + y_{n-1} (\alpha x + \beta(l - x))] \,. \tag{1.2}$$

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. (1.2) is the required optimal strategy. ■

1.1.2 Stochastic

§ Problem 1.1.2. 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 $0 \le x \le l$. From x : they get a return of $u_1(x)$ with a probability p_1 such that they are left with a lesser amount $\alpha_1 x : 0 < \alpha_1 < 1$ and a return of $u_2(x)$ with a probability $p_2 = 1 - p_1$ such that they are left with a lesser amount $\alpha_2 x : 0 < \alpha_2 < 1$. Similarly from $l - x$: a return of $v_1(l - x)$ with probability p_1 such that they are left with a lesser amount $\beta_1(l - x) : 0 < \beta_1 < 1$ and a return of $v_2(l - x)$ with probability $p_2 = 1 - p_1$ such that they are left with a lesser amount $\beta_2(l - x) : 0 < \beta_2 < 1$. So the total amount left after the first step of division is $\alpha_1 x + \beta_1(l - x)$ with probability p_1 and $\alpha_2 x + \beta_2(l - x)$ with probability p_2 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 $y_n(l)$ denote the expected total return of an n-stage process, obtained using an optimal policy, starting with an initial amount l.

Then, the equations are obtained as before :

$$y_1(l) = \operatorname*{Max}_{x \in [0,l]} [p_1 [u_1(x) + v_1(l - x)] + p_2 [u_2(x) + v_2(l - x)]] \,,$$

$$\begin{aligned} y_n(l) = \operatorname*{Max}_{x \in [0,l]} & [p_1 [u_1(x) + v_1(l - x) + y_{n-1} (\alpha_1 x + \beta_1(l - x))] \\ & + p_2 [u_2(x) + v_2(l - x) + y_{n-1} (\alpha_2 x + \beta_2(l - x))]] \,. \end{aligned}$$

As can be noted, using the expected value as the measure of the value of a policy, the stochastic process is structurally reduced to the deterministic counterpart. ■

§ Problem 1.1.3. With the looted amount $l(>0)$ at his disposal, the leader of a gang of robbers decides to buy a sophisticated weapon, which is not readily available. The probability of buying it is $p(x)$ at the expense of amount x, where $x \in [0, l]$. If he is not able to buy the weapon at his first attempt, he continues with the remaining amount $l - x$. How should he proceed in order to maximize his over-all chance of success ? ♢

§§ Solution. Let $y_n(l)$ be the maximum over-all probability of success, given : the initial amount l and n trials.

After utilizing the amount x on the first try, the probability of buying is $p(x)$. So the probability of not buying will be $1 - p(x)$. Then the leader uses an optimal policy starting with the remaining amount $l - x$. Hence, the required optimal procedure is

$$\therefore y_n(l) = \underset{x\in[0,l]}{\text{Max}} \left\{p(x) + [1 - p(x)]\, y_{n-1}(l-x)\right\}.$$

Hence it is relatively easy to formulate the problem if the probability of failure is considered than the probability of success. ■

1.2 Exam Prep

§ Problem 1.2.1. School board decides to declare the final exam's result in such a way that a student, named Ram, is either pass or fail. Initially the probability of his failure is given as p. Proper study will reduce this probability to αp, where $\alpha \in [0, 1]$. Mock test will help him know exactly whether he is fail or pass. Ram wants to pass the exam in a minimum time. What is the optimal procedure he should follow ? ♢

§§ Solution. Let $f(p)$ be the expected minimum time to pass the exam given the initial probability of failing the same is p.

If Ram appears in the mock test, it is known that he is fail with initial probability p and he continues with that knowledge. If he is pass then the process is over.

So if he appears for the mock test as a first step, the expected minimum time is given by

$$1 + pf(1).$$

Otherwise if he follows the proper study plan, then the expected minimum time is given by

$$1 + f(\alpha p).$$

Combining these two results, the required optimal procedure is

$$\begin{aligned} f(p) &= \underset{p\in[0,1]}{\text{Min}} \left[1 + pf(1),\ 1 + f(\alpha p)\right], \\ f(0) &= 0. \end{aligned}$$ ■

§ Problem 1.2.2. Ram (a student) plans to prepare for the final exams using any of two exam-guides where he is allowed to refer one of these two guides at a given stage. There is a probability p_1 of scoring one mark, a probability p_2 of scoring two marks and a probability p_3 of finishing the study plan with the first guide. The second guide has a similar set of probabilities p_1', p_2' and p_3'. Chalk out the optimal study plan to help him maximize the probability of scoring at least n marks. ♢

§§ Solution. Let $f(n)$ be the probability of scoring at least n marks following an optimal study plan.

If Ram scores k marks on the first step, then he continues so as to maximize the probability of scoring at least $n - k$ marks on the following steps. Notice with p_3 or p_3', there is no gain because the process is terminated. This leads to the following optimal study plan :

$$f(n) = \underset{n\geq 2}{\text{Max}} \left[p_1 f(n-1) + p_2 f(n-2),\ p_1' f(n-1) + p_2' f(n-2)\right].$$

This holds even for $n = 0, 1$ with the convention that $f(-k) = 1$ for $k \geq 0$. ■

§ Problem 1.2.3. Ram purchased two sample question papers to help him practice for the final exam. The first paper has m questions while the second has n questions. There is only one solution bank available with him. The probability of solving α percent of the questions of the first paper with this solution bank is p_1, the bank still being useful. There is a probability $(1 - p_1)$ that the bank doesn't help in solving any question and will be of no further use. Similarly, the second question paper has the probabilities p_2 and $(1 - p_2)$ associated with it with solving β percent of the questions. How does Ram proceed in order to maximize the total number of questions solved before the solution bank is rendered useless. ♢

§§ **Solution**. Let $f(x,y)$ be the expected number of questions solved using an optimal sequence of choices, when the first paper L has the number of questions x and the second paper B has the number of questions y. $x,\ y \geq 0$.

Let $\alpha_1 = \dfrac{\alpha}{100}$ and $\beta_1 = \dfrac{\beta}{100}$.

If Ram chooses the first paper L (say), then :

$$f_L(x,y) = p_1 \left\{\alpha_1 x + f\left[(1-\alpha_1)\,x,\ y\right]\right\}$$

If Ram chooses the second paper M (say), then :

$$f_M(x,y) = p_2 \left\{\beta_1 y + f\left[x, (1-\beta_1)\,y\right]\right\}$$

Hence, the optimal procedure is :

$$f(x,y) = \operatorname*{Max}_{x,y\geq 0} \left[f_L(x,y),\ f_M(x,y)\right].$$

■

§ **Problem 1.2.4.** In the § Problem 1.2.3, it is desired to maximize the expected number of the total solved questions, N. Devise the optimal procedure for maximizing the expected value of $\delta(N)$, where δ is a given function. ◊

§§ **Solution**. Let z be the number of questions already solved.

Let $f(x,y,z)$ be the expected $\delta(N)$, using an optimal policy.

Let $\alpha_1 = \dfrac{\alpha}{100}$ and $\beta_1 = \dfrac{\beta}{100}$.

If Ram chooses the first paper L (say), then :

$$f_L(x,y,z) = p_1 f\left[(1-\alpha_1)\,x,\ y,\ z+\alpha_1 x\right] + (1-p_1)\,\delta(z).$$

If Ram chooses the second paper M (say), then :

$$f_M(x,y,z) = p_2 f\left[x, (1-\beta_1)\,y,\ z+\beta_1 y\right] + (1-p_2)\,\delta(z).$$

Hence, the optimal procedure is :

$$f(x,y) = \operatorname*{Max}_{x,y\geq 0} \left[f_L(x,y,z),\ f_M(x,y,z)\right],$$
$$f(0,0,z) = \delta(z).$$

■

§ **Problem 1.2.5.** Reconsider the § Problem 1.2.3 in the case in which Ram knows only the expected number of questions in each paper and the expected number of questions solved each time, without being able to observe the results of individual operations. ◊

§§ **Solution**. Let $f(x,y)$ be the expected number of questions solved using an optimal sequence of choices when the first paper L has expected number of questions x and the second paper B has expected number of questions y.

Let $\alpha_1 = \dfrac{\alpha}{100}$ and $\beta_1 = \dfrac{\beta}{100}$.

If Ram chooses the first paper L (say), then :

$$f_L(x,y) = p_1 \left\{\alpha_1 x + f\left[(1-\alpha_1)\,x,\ y\right]\right\}$$

If Ram chooses the second paper M (say), then :

$$f_M(x,y) = p_2 \left\{\beta_1 y + f\left[x, (1-\beta_1)\,y\right]\right\}$$

Hence, the optimal procedure is :

$$f(x,y) = \operatorname*{Max}_{n\geq 2} \left[f_L(x,y),\ f_M(x,y)\right].$$

Note that this solution is same as in § Problem 1.2.3 though the problems are quite different in structure.

■

1.3 Optimal Coin Tossing

§ **Problem 1.3.1.** Two brothers, Ram and Shyam, Ram possessing an amount x and Shyam possessing an amount y, play a modified coin-tossing game described by the matrix :

$$M = \begin{vmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{vmatrix}.$$

Assuming that each player is motivated by a desire to ruin the other, how does each play ? ◊

§§ **Solution**. Let l be the total amount of money, which remains constant in the game. Hence it is sufficient to specify the amount of money x held by Ram.

Let $f(x)$ be the probability that Shyam is ruined before Ram when Ram has x and Shyam has $l-x$ and when both use the optimal strategies.

Let Ram's strategy be $p = (p_1,\ p_2)$, where p_1 and p_2 represent the respective frequencies with which the first and second rows of M are played.

Let Shyam's strategy be $q = (q_1,\ q_2)$, where q_1 and q_2 represent the respective frequencies with which Shyam chooses the first and second columns of M.

$$\begin{aligned}\therefore f(x) &= p_1 q_1 f\left(x + m_{11}\right) + p_1 q_2 f\left(x + m_{12}\right)\\ &\quad + p_2 q_1 f\left(x + m_{21}\right) + p_2 q_2 f\left(x + m_{22}\right)\\ &= g\left[p, q, f(x)\right] \text{ (say), } \text{ where } x \in (0, l).\end{aligned}$$

If both play optimally, then

$$\begin{aligned}\therefore f(x) &= \underset{p}{\text{Max}}\ \underset{q}{\text{Min}}\ g\left[p, q, f(x)\right]\\ &= \underset{q}{\text{Min}}\ \underset{p}{\text{Max}}\ g\left[p, q, f(x)\right],\ x \in (0, l)\\ f(0) &= 0,\ x \leq 0,\\ f(x) &= 1,\ x \geq l.\end{aligned}$$

Hence $f(x)$ is the value of the game with the payoff matrix as

$$\begin{vmatrix} f\left(x + m_{11}\right) & f\left(x + m_{12}\right) \\ f\left(x + m_{21}\right) & f\left(x + m_{22}\right) \end{vmatrix}.$$

■

1.4 Proving Optimality Principle

As noted earlier, the principle of optimality helps in transforming a single N-dimensional problem at hand to a sequence of N one-dimensional problems in a specified order (i.e. a time-like concept is introduced here : rendering this approach as a dynamic one). The functional equation has recurrent yet independent structure. This in turn establishes the transformation from optimal to functional and vice versa.

Typical optimization problem at hand is finding the maximum of a function ϕ of n variables $x_i : i \in [1, n]$:

$$\phi\left(x_1,\ x_2,\ \ldots,\ x_n\right) = \sum_{i=1}^{n} \delta_i\left(x_i\right) \tag{1.3}$$

taken over the region of values determined by the relations:

$$\sum_{i=1}^{n} x_i = c \tag{1.4}$$

$$x_i \geq 0 \tag{1.5}$$

where c is a positive constant.

This problem can be modeled as an optimal resource allocation one, where c is a fixed quantity of an economic resource. Each potential usage of the resource is an activity. As a result of using all or part of this resource in any single activity, a certain return is derived. Assuming that the return from any activity is independent of the allocations to the other activities and the total return can be obtained as the sum of the individual returns, divide the resources so as to maximize the total return.

Here x_i represents the quantity of the resource assigned to the i^{th} activity and $\delta_i\left(x_i\right)$ represents the return from the i^{th} activity. The allocations are done one at a time, i.e. first a quantity of resources is assigned to the n^{th} activity, then to the $(n-1)^{th}$ activity and so on. Due to this time-like ordering constraint, this is indeed a dynamic allocation process.

Since the maximum of $\phi\left(x_1,\ x_2,\ \ldots,\ x_n\right)$ over the designated region depends upon c and n, a sequence of functions $f_1(c)$, $f_2(c)$, ..., $f_n(c)$ is defined as :

$$f_n(c) = \underset{\{x_i\}}{\text{Max}}\ \phi\left(x_1,\ x_2,\ \ldots,\ x_n\right),\ \sum_{i=1}^{n} x_i = c,\ x_i \geq 0 \tag{1.6}$$

$$f_n(0) = 0\ (\because \delta_i(0) = 0) \tag{1.7}$$

$$f_1(c) = \delta_1(c),\ c \geq 0. \tag{1.8}$$

The function $f_n(c)$ is then the optimal return from an allocation of the quantity of resources c to n activities.

Since x_n is allocated to the n^{th} activity, where $x_n \in [0,\ c]$, the remaining quantity of resources, $c - x_n$, will be used to obtain a maximum return from the remaining $(n-1)$ activities.

Since the optimal return for $n-1$ activities starting with quantity $c - x_n$ is, by definition, $f_{n-1}(c-x_n)$, hence the initial allocation of x_n to the n^{th} activity results in a total return of $\delta_n(x_n) + f_{n-1}(c-x_n)$ from the n-activity process.

An optimal choice of x_n is obviously one which maximizes this function. Hence this results into the following recurrence :

$$f_n(c) = \underset{x_n \in [0,\ c]}{\text{Max}} \left[\delta_n(x_n) + f_{n-1}(c-x_n)\right],\ n \geq 2,\ c \geq 0 \tag{1.9}$$

$$f_1(c) = \delta_1(c). \tag{1.10}$$

With the known value of $f_1(c)$, it is easy to compute the sequence $\{f_n(c)\}$ inductively because $f_2(c)$ is computed from $f_1(c)$ and so on.

It can be noted that :

$$\underset{\sum_{i=1}^{n} x_i = c,\ x_i \geq 0}{\text{Max}} = \underset{x_n \in [0,\ c]}{\text{Max}} \left[\underset{\sum_{i=1}^{n-1} x_i = c - x_n,\ x_i \geq 0}{\text{Max}}\right] \tag{1.11}$$

Hence Eq. (1.6) translates as follows

$$\begin{aligned} f_n(c) &= \underset{\sum_{i=1}^{n} x_i = c,\ x_i \geq 0}{\text{Max}} \left[\sum_{i=1}^{n} \delta_i(x_i)\right] \\ &= \underset{x_n \in [0,\ c]}{\text{Max}} \left[\underset{\sum_{i=1}^{n-1} x_i = c - x_n,\ x_i \geq 0}{\text{Max}} \left[\sum_{i=1}^{n} \delta_i(x_i)\right]\right] \\ &= \underset{x_n \in [0,\ c]}{\text{Max}} \left[\delta_n(x_n) + \underset{\sum_{i=1}^{n-1} x_i = c - x_n,\ x_i \geq 0}{\text{Max}} \left[\sum_{i=1}^{n-1} \delta_i(x_i)\right]\right] \\ &= \underset{x_n \in [0,\ c]}{\text{Max}} \left[\delta_n(x_n) + f_{n-1}(c-x_n)\right]. \end{aligned} \tag{1.12}$$

Note that the functional equation Eq. (1.9) is derived (earlier using the principle of optimality) again as in Eq. (1.12) using elementary mathematics. This establishes the Principle of Optimality. The proof is in the pudding. The functional equation technique is more like a search procedure which is much better compared to the enumeration of all cases.

It is the Principle of Optimality that furnishes the key. According to this principal, once some initial x_n is chosen, then there is no need to examine all policies involving that particular choice of x_n, but rather only those policies which are optimal for an $n-1$ stage process with resources $c - x_n$. In this magical way, the additive operations are in force than multiplicative ones. For example, the time required for a n^2-stage process is now almost precisely n-times the time required for a n-stage process.

This computing paradigm greatly reduce the time required to solve the original problem. Note that this is possible due to combination of two procedures : the utilization of structural properties of the solution and reduction in dimension.

Additionally, though the maximum return is uniquely determined but there may be many optimal policies which yield this return. Determination of these optimal policies require creativity and insight into the problem space.

Chapter 2

Computation

2.1 Ascension to Heaven

§ Problem 2.1.1. Once upon a time, a certain group of daemons and humans on the earth performed together some tantric rituals in a bid to go to heaven. Pleased with their devotion, Indra, the Lord of Heaven, provided his white flying majestic elephant, Airawat, for that purpose. But certain constraints were imposed throughout the process of ascension. Anyone could cling to the tail of Airawat, while allowing others to cling to him and so on, thus forming a chain. Ordering was not important. Airawat continued flying back and forth from earth to heaven with at least one being.

Initially, the number of daemons and humans on the earth were d_e and h_e respectively and the number of daemons and humans in the heaven were d_h and h_h respectively.

In order to prevent the humans from being devoured by the daemons On the earth, the following constraints were imposed :

(a). $\delta_e(d_e, h_e) \geq 0$ on the earth,
(b). $\delta_h(d_h, h_h) \geq 0$ in the heaven, and
(c). $\delta_a(d, h) \geq 0$ on Airawat who would not allow more than $\gamma \geq 1$ beings for a ride.

Find the maximum number of beings ascended to heaven without any human sacrifice. ◊

§§ Solution. Due to prohibition of human sacrifice by the daemons, the total number of beings remains constant throughout the process of ascension : namely, $d_e + h_e + d_h + h_h$. Hence at any time, the state of system is completely specified by the numbers d_e and h_e.

Let the function $f_n(d_e, h_e)$ represent the maximum number of beings in the heaven at the end of n stages, starting with d_e daemons and h_e humans on the earth and d_h daemons and h_h humans in the heaven.

It is assumed that once everyone has ascended to the heaven then Airawat goes back to Indra. Hence the process is terminated, i.e. there is no need for anyone to descend back to the earth.

During one stage of the process, the following sequence of actions takes place :

1. Airawat ascends to heaven with $\alpha_1 \in [0, d_e]$ daemons and $\beta_1 \in [0, h_e]$ humans, followed by
2. Airawat descends back to the earth with $\alpha_2 \in [0, d_h + \alpha_1]$ daemons and $\beta_2 \in [0, h_h + \beta_1]$ humans.

Since Airawat would not allow more than $\gamma \geq 1$ beings for a ride, hence

$$\alpha_1 + \beta_1 \leq \gamma,\ \alpha_2 + \beta_2 \leq \gamma$$

Additionally, on Airawat :

$$\delta_a(\alpha_1, \beta_1) \geq 0,\ \delta_a(\alpha_2, \beta_2) \geq 0$$

On the earth :

$$\delta_e(d_e - \alpha_1,\ h_e - \beta_1) \geq 0$$
$$\delta_e(d_e - \alpha_1 + \alpha_2,\ h_e - \beta_1 + \beta_2) \geq 0.$$

In the heaven :

$$\delta_h \left(d_h + \alpha_1,\ h_h + \beta_1\right) \geq 0$$
$$\delta_h \left(d_h + \alpha_1 - \alpha_2,\ h_h + \beta_1 - \beta_2\right) \geq 0$$

Using the principle of optimality, the functional equation approach leads to the following recurrence relation :

$$f_n \left(d_e, h_e\right) = \underset{\alpha,\beta}{\text{Max}}\, f_{n-1} \left(d_e - \alpha_1 + \alpha_2,\ h_e - \beta_1 + \beta_2\right),\ n \geq 2$$

For $n = 1$:

$$f_1 \left(d_e, h_e\right) = \underset{\alpha,\beta}{\text{Max}} \left[\left(d_h + \alpha_1\right) + \left(h_h + \beta_1\right)\right].$$

where α_1 and β_1 are subject to the foregoing constraints. ■

§ Problem 2.1.2. In § Problem 2.1.1, find the minimum number of round-trips required by Airawat to bring all the beings to heaven (if feasible). ♢

§§ Solution. Let n be the required minimum number of round-trips.

Once all the beings are brought to heaven then total number of beings in heaven is $[(d_e + d_h) + (h_e + h_h)]$. Hence as soon as f_n attains this value, the corresponding n is the required minimum. ■

§ Problem 2.1.3. Once upon a time, a group of three daemons and three humans on the earth performed together some tantric rituals in a bid to go to heaven. Pleased with their devotion, Indra, the Lord of Heaven, provided his white flying majestic elephant, Airawat, for that purpose. Airawat would not allow more than two beings for a ride. Anyone could cling to the tail of Airawat, while allowing others to cling to him and so on, thus forming a chain. Ordering was not important. Airawat continued flying back and forth from earth to heaven with at least one being. As soon as the number of daemons was higher than that of humans, even momentarily, the daemons would devour the humans, whether on the earth or with Airawat or in the heaven. Determine an optimal strategy of ascension of everyone to the heaven without any human sacrifice. ♢

§§ Solution. A generic algorithmic approach to the stated problem, using functional equation technique resulting from the principle of optimality, is already chalked out in the solutions of § Problem 2.1.1 and § Problem 2.1.2.

For the n-stage process, choice of the possible initial states is dictated by the specified constraints. Others lead to the human sacrifice which in undesired for the process.

Let (d, h) represent the following state :

- there are d daemons and h humans on the earth, hence
- there are $3 - d$ daemons and $3 - h$ humans in the heaven.

Here $d \in [0, 3]$ and $h \in [0, 3]$. Since both d and h can take any of the four values : (0, 1, 2, 3), hence the total number of initial states is 16. To avoid human sacrifice, the number of daemons, either on the earth or in the heaven, should not be greater than that of the humans except when there in no human.

$d_e(= d)$	$h_e(= h)$	$d_h(= 3 - d)$	$h_h(= 3 - h)$	human sacrifice
0	0	3	3	N
0	1	3	2	$Y(d_h > h_h)$
0	2	3	1	$Y(d_h > h_h)$
0	3	3	0	$N(d_h > h_h = 0)$
1	0	2	3	N
1	1	2	2	N
1	2	2	1	$Y(d_h > h_h)$
1	3	2	0	$N(d_h > h_h = 0)$
2	0	1	3	N
2	1	1	2	$Y(d_e > h_e)$
2	2	1	1	N
2	3	1	0	$N(d_h > h_h = 0)$
3	0	0	3	$N(d_e > h_e = 0)$
3	1	0	2	$Y(d_e > h_e)$
3	2	0	1	$Y(d_e > h_e)$
3	3	0	0	N

The state $(0, 0)$ implies that everyone is the heaven in which case nothing needs to be done.

The state $(0, 3)$ implies that since all the three daemons are already in heaven, hence they are higher in number leading to human sacrifice in the heaven in the next step.

Hence only the following initial states are feasible (no human sacrifice):

(1, 0), (1, 1), (1, 3), (2, 0), (2, 2), (2, 3), (3, 0) and (3, 3).

With these initial states, computation using the algorithmic solution of § Problem 2.1.1 can be done as follows.

$$\because f_1(d_e, h_e) = \underset{\alpha,\beta}{\text{Max}}\left[(d_h + \alpha_1) + (h_h + \beta_1)\right]$$

$$\therefore f_1(1,0) = 6,\ f_1(1,1) = 6,\ f_1(1,3) = 2,\ f_1(2,0) = 6,$$
$$f_1(2,2) = 3,\ f_1(2,3) = 2,\ f_1(3,0) = 4, f_1(3,3) = 1.$$

For example : to compute $f_1(1, 3)$: the only feasible moves, satisfying the constraints, are : Airawat flies with 2 humans to the heaven and flies back to earth with 1 daemon and 1 human. $\therefore \alpha_1 = 0,\ \beta_1 = 2,\ \alpha_2 = 1,\ \beta_2 = 1$.

Similarly with $f_1(2, 3)$: Airawat flies with 2 daemons to the heaven and flies back to earth with 1 daemon.

Note that the six-valued functions repeat themselves, i.e, if $f_k(i, j) = 6$ then $f_{k+l} = 6$ for $l = 1, 2, \ldots$.

Hence computations using the recurrence relation is done for all non-six values :

$$\therefore f_2(1,3) = 3,\ f_2(2,2) = 4,\ f_2(2,3) = 2,\ f_2(3,0) = 6, f_2(3,3) = 1.$$

For example :

$$\begin{aligned} f_2(1,3) &= f_1(1 - \alpha_1 + \alpha_2, 3 - \beta_1 + \beta_2) \\ &= f_1(1 - 0 + 1, 3 - 2 + 1) = f_1(2,2) = 3. \end{aligned}$$

Similarly

$$\therefore f_3(1,3) = 4,\ f_3(2,2) = 6,\ f_3(2,3) = 3,\ f_3(3,3) = 2.$$
$$\therefore f_4(1,3) = 6,\ f_4(2,3) = 4,\ f_4(3,3) = 3.$$
$$\therefore f_5(2,3) = 6,\ f_5(3,3) = 4.$$
$$\therefore f_6(3,3) = 6.$$

Note that, at sixth stage, the process is over, i.e., everyone is in the heaven. Therefore, the required number of crossings is 6.

Recording the maximizing decision at each stage will determine the optimal policy. ■

2.2 Fibonacci Line Search

A real-valued continuous function $f(x)$ is called a convex function over $x \in [a, b]$, if its value at the mid-point of every interval in $[a, b]$ never exceeds the arithmetic mean of its values at the ends of the interval. For example : x^2 and e^x.

Therefore, for any interval $[x_1, x_2] \in [a, b]$, the following inequality holds :

$$f\left(\frac{x_1 + x_2}{2}\right) \leq \frac{f(x_1) + f(x_2)}{2}.$$

Note that, the graph of a convex function lies below the line segment between any two points $[x_1, x_2] \in [a, b]$.

A real-valued continuous function $f(x)$ is called a concave function over $x \in [a, b]$, if its value at the mid-point of every interval in $[a, b]$ always exceeds the arithmetic mean of its values at the ends of the interval.

$$\therefore f\left(\frac{x_1 + x_2}{2}\right) \geq \frac{f(x_1) + f(x_2)}{2}.$$

Note that, the graph of a concave function lies above the line segment between any two points $[x_1, x_2] \in [a, b]$.

$$\therefore \text{concave function} = -\text{convex function}.$$

A real-valued continuous function $f(x)$ is called a unimodal function over $x \in [a, b]$, if $\exists \alpha \in [a, b]$, such that the following holds :

1. $x \leq \alpha$: $f(x)$ is monotonically increasing, and
2. $x > \alpha$: $f(x)$ is monotonically decreasing.

or, the following holds :

1. $x < \alpha$: $f(x)$ is monotonically increasing, and
2. $x \geq \alpha$: $f(x)$ is monotonically decreasing.

Note that, concave functions are unimodal functions too. Finding the maximum of a concave function $f(x)$ is same as finding the minimum of the convex function $-f(x)$.

§ Problem 2.2.1. Fibonacci line search is an optimal search algorithm for determining the location of the point α of a unimodal function as defined earlier.

Let $f(x)$ be a unimodal function over $x \in [0, C_n]$. Assuming that the number C_n possesses a property that the point, at which $f(x)$ is maximum, can be located within a unit-length sub-interval by calculating at most n values of the function $f(x)$ and making comparisons.

If $F_n = \text{Max}\, C_n$, then prove that F_n is the n^{th} Fibonacci number, i.e.:

$$F_0 = 1 = F_1$$
$$F_n = F_{n-1} + F_{n-2},\ n \geq 2. \qquad \Diamond$$

§§ Solution. If $n = 0$, then the domain of $f(x)$ is $[0, C_0]$. Since no value of $f(x)$ is given and the sub-interval is of unit length,

$$\therefore F_0 = \text{Max}\, C_0 = 1.$$

For $n = 1$, the domain of $f(x)$ is $[0, C_1]$. Since only one value of $f(x)$ is given, this is not sufficient to locate the maximum value,

$$\therefore F_1 = \text{Max}\, C_1 = 1.$$

For $n \geq 2$: for $(x_1, x_2) \in (0, C_n)$, the values of $f(x_1)$ and $f(x_2)$ are computed.

If $f(x_1) > f(x_2)$, then $f_{max} \in (0, x_2)$.

If $f(x_2) > f(x_1)$, then $f_{max} \in (x_1, C_n)$.

If $f(x_1) = f(x_2)$, then $f_{max} \in (x_1, x_2)$. Still either of the previous two intervals is chosen for the sake of simplicity.

Thus, at each stage after the first computation, the process yields a sub-interval of $[0, C_n]$ and the value of $f(x)$ at an interior point within that sub-interval.

Note that, values at the ends of an interval is of no use for the intended purpose.

For $n = 2$: $C_n = C_2 = 2 - \delta,\ x_1 = 1 - \delta,\ x_2 = 1$, for an infinitesimal $\delta > 0$. $\therefore \text{Max}\, C_2 \geq 2$. But as per the foregoing analysis : $\text{Max}\, C_2 < 2 + \gamma$ for any $\gamma > 0$.

$$\therefore F_2 = \text{Max}\, C_2 = 2 = F_1 + F_0.$$

For $n > 2$: inductive approach will be used.

Assume that $F_k = F_{k-1} + F_{k-2}$ for $k \in [2, n-1]$.

To prove : $F_n = F_{n-1} + F_{n-2}$.

Assume $[x_1, x_2] \in [0, C_n]$.

If $f(x_1) > f(x_2)$, then $f_{max} \in (0, x_2)$.

Note that for $k = n - 1$, i.e. when $n - 1$ calculations are allowed, since x_1 is already chosen as the first choice, there are only $n - 2$ more choices are allowed. $\therefore x_2 < F_{n-1}$.

Additionally, since $f_{max} \in (0, x_1)$ with only $n - 2$ choices left, $\therefore x_1 < F_{n-2}$.

Similarly, if $f(x_2) > f(x_1)$: $C_n - x_1 < F_{n-1}$.

$$\therefore C_n < x_1 + F_{n-1} < F_{n-1} + F_{n-2}$$
$$\therefore F_n = \text{Max}\, C_n \leq F_{n-1} + F_{n-2}. \tag{2.1}$$

Note that, the choice of C_n, x_1 and x_2 can be made arbitrarily close to their respective upper bounds, namely : $F_{n-1} + F_{n-2}$, F_{n-2} and F_{n-1} as follows :

$$C_n = \left(1 - \frac{\delta}{2}\right)(F_{n-1} + F_{n-2})$$
$$x_1 = \left(1 - \frac{\delta}{2}\right)F_{n-2}$$
$$x_2 = \left(1 - \frac{\delta}{2}\right)F_{n-1}.$$

Since δ can be made arbitrarily small,

$$\therefore F_n \geq F_{n-1} + F_{n-2}. \tag{2.2}$$

From Eq. (2.1) and Eq. (2.2), it follows :

$$F_n = F_{n-1} + F_{n-2}.$$

Note that, after comparing $f(x_1)$ and $f(x_2)$, length of the interval left is $C_{n-1} = \left(1 - \frac{\delta}{2}\right) F_{n-1}$. Additional there is a value calculated at an optimal first position for the smaller interval.

Similarly, $C_k = \left(1 - \frac{\delta}{2}\right) F_k$ for $k \in [2, n]$.

$\therefore C_2 = \left(1 - \frac{\delta}{2}\right) F_2 = 2 - \delta$, so that the final interval is of unit length. ∎

§ Problem 2.2.2. Let $f(x)$ be a unimodal function defined only for discrete values of x, say, a set of C_n points. Assume that the integer C_n possesses a property that the maximum of $f(x)$ can always be identified in n computations and subsequent comparisons.

If $D_n = \text{Max}\, C_n$, then prove that

$$D_n = -1 + F_{n+1},\ n \geq 1.$$

where F_n is the n^{th} Fibonacci number, i.e.:

$$\begin{aligned} F_0 &= 1 = F_1 \\ F_n &= F_{n-1} + F_{n-2},\ n \geq 2. \end{aligned}$$

◊

§§ Solution. For the sake of simplicity, let the discrete points be in ascending order in units of one, i.e. $x \in [1, 2, 3, \ldots, C_n]$.

It is easy to observe that $D_1 = 1 = -1 + F_2,\ D_2 = 2 = -1 + F_3,\ D_3 = 4 = -1 + F_5$.

As earlier, the proof by induction will be adopted for $n > 3$.

Assume that $D_k = -1 + F_{k-1}$ for $k \in [4, n-1]$.

To prove that $D_n = -1 + F_{n+1}$.

Assume $[x_1, x_2] \in [4, C_n]$.

In the light of similar logic as in § Problem 2.2.1, it can be deduced that

$$\begin{aligned} x_1 &\leq D_{n-2} + 1 \\ C_n - x_1 &\leq D_{n-1} \\ \therefore C_n &\leq x_1 + D_{n-1} \\ &\leq D_{n-2} + 1 + D_{n-1} \\ &= (-1 + F_{n-1}) + 1 + (-1 + F_n) \\ &= -1 + F_{n+1}. \end{aligned}$$

∎

2.3 Coin Change

§ Problem 2.3.1. Given a list of denominations for $k > 0$ coins, $c_i \ : i \in [0..k-1]$ and an unlimited supply of any denomination, determine the minimum number of coins needed to make change for a given amount $s \geq 0$. ◊

§§ Solution. Let $f_n(s)$ be the minimum number of coins needed to make change for a given amount s, obtained using an optimal policy and n steps.

Let c_i be the denomination of the first or last coin, i.e. c_i is used at the 1^{st} or n^{th} step respectively. Then we can use an optimal policy starting with the remaining amount $s - c_i \geq 0$.

Hence the required optimal procedure is

$$\begin{aligned} \therefore f_n(s > 0) &= \underset{\substack{i \in [0,\ k-1] \\ s - c_i \geq 0}}{\text{Min}} [f_{n-1}(s - c_i) + 1] \\ f_n(0) &= 0 \\ f_n(s < 0) &= 0 \end{aligned}$$

i.e. if c_i is the first (or last) coin in the optimal solution to making change for amount s, then one c_i coin plus $f_{n-1}(s - c_i)$ coins to make change for the remaining amount $s - c_i$ is the optimal procedure to make change for the total amount s.

Intuitive Proof : Let us prove that the optimal solution $f_n(s)$ for the amount s contains within it an optimal solution $f_{n-1}(s - c_i)$ for the amount $s - c_i$. Let us assume that the optimal solution $f_n(s)$ uses m coins and it is known that this optimal solution uses a coin c_i. Then there are $m - 1$ coins in the solution $f_{n-1}(s - c_i)$ used within the optimal solution $f_n(s)$. If $f_{n-1}(s - c_i)$ used fewer than $m - 1$ coins, then this solution can be used to produce a solution $f_n(s)$ that uses less than k coins, which contradicts the optimality of our solution. Hence the solution $f_{n-1}(s - c_i)$ is also an optimal one.

Algorithm 1 Minimum Coin Change : Iterative (Bottom-up) Approach

```
 1: function min-coin-change($c[0..k-1]$, $s$)
 2:     $f[0] \leftarrow 0$                ▷ 0 coins needed to make change for the amount 0
 3:     for $i \leftarrow 1$, $s$ do
 4:         $f[i] \leftarrow \infty$
 5:         for $j \leftarrow 0$, $k-1$ do
 6:             if $c[j] \leq i$ then
 7:                 $f[i] \leftarrow \text{Min}\,(f[i - c[j]]) + 1$
 8:             end if
 9:         end for
10:     end for
11:     return $f[s]$
12: end function
```

This is also known as **forward dynamic programming**.

We need to try k denominations of coins per state in the amount s, hence the time complexity is $\mathcal{O}(ks)$. We are using a storage of size $s+1$ to store (and possibly reuse)[*] the optimally computed states, therefore the space complexity is $\mathcal{O}(s)$.

```cpp
int min_coin_change(std::vector<int> & coins, int amount)
{
    std::vector<int> f(amount + 1, std::numeric_limits<int>::max()/2);

    f[0] = 0;

    for(int i = 1; i <= amount; ++i)
    {
        for(int c : coins)
        {
            if(c <= i)
            {
                f[i] = std::min(f[i], f[i-c] + 1);
            }
        }
    }
    return f[amount] > amount ? -1 : f[amount];
}
```

Algorithm 2 Minimum Coin Change : Recursive (Top-down) Approach

```
 1: $f[0..s+1] \leftarrow 0$
 2: function min-coin-change($c[0..k-1]$, $s$)
 3:     if $f[s] \neq 0$ then
 4:         return $f[s]$
 5:     end if
 6:     $min \leftarrow \infty$
 7:     for $i \leftarrow 0$, $k-1$ do
 8:         $res \leftarrow$ min-coin-change($c[0..k-1]$, $s-i$)
 9:         if $res \geq 0$ and $res < min$ then
10:             $min \leftarrow 1 + res$
11:         end if
12:     end for
13:     $f[s] \leftarrow min$
14:     return $f[s]$
15: end function
```

This is also known as **backward dynamic programming**.

[*]Memoization

```
int min_coin_change(std::vector<int> & coins, int amount, std::vector<int> & f)
{
    if(amount < 0) return -1;

    if(amount == 0) return 0;

    if(f[amount] != 0) return f[amount];

    int min = std::numeric_limits<int>::max()/2;

    int res = 0;

    for(int c : coins)
    {
        res = min_coin_change(coins, amount - c, f);

        if(res >= 0 and res < min)
        {
            min = 1 + res;
        }
    }

    f[amount] = (min == std::numeric_limits<int>::max()/2) ? -1 : min;

    return f[amount];
}


int min_coin_change(std::vector<int> & coins, int amount)
{
    std::vector<int> f(amount + 1, 0);

    return min_coin_change(coins, amount, f);
}
```

■

§ Problem 2.3.2. In § Problem 2.3.1, identify the optimal set of coins for making the change for the amount s. ◊

§§ Solution. In § Problem 2.3.1, we need to construct the optimal solution from the computed information.

Algorithm 3 Minimum Coin Change : Optimal set of coins

```
1: function min-coin-change($c[0..k-1]$, $s$)
2:     $f[0] \leftarrow 0$
3:     for $i \leftarrow 1, s$ do
4:         $f[i] \leftarrow \infty$
5:         for $coin \in c[0..k-1]$ do
6:             if $coin \leq i$ then
7:                 if $f[i-coin] + 1 < f[i]$ then
8:                     $f[i] \leftarrow f[i-coin] + 1$
9:                     coinset[i] $\leftarrow coin$
10:                end if
11:            end if
12:        end for
13:    end for          ▷ coinset[s] is the first coin in the optimal solution for amount $s$

14:    $changes \leftarrow \emptyset$

15:    if $f[s] > s$ then
16:        return $changes$                                          ▷ No solution
17:    end if

18:    $j \leftarrow s$
19:    while $j > 0$ do
20:        $changes.add(coinset[j])$
21:        $j \leftarrow j - coinset[j]$
22:    end while
23:    return $changes$
24: end function
```

```cpp
std::vector<int> min_coin_change(std::vector<int> & coins, int amount)
{
    std::vector<int> f(amount + 1, std::numeric_limits<int>::max()/2);
    std::vector<int> coinset(amount + 1, 0);
    std::vector<int> changes;

    f[0] = 0;

    for(int i = 1; i <= amount; ++i)
    {
        for(int c : coins)
        {
            if(c <= i)
            {
                if(f[i-c] + 1 < f[i])
                {
                    f[i] = f[i-c] + 1;
                    coinset[i] = c;
                }
            }
        }
    }

    if(f[amount] > amount) return changes;

    int j = amount;

    while(j > 0)
    {
        changes.push_back(coinset[j]);
        j -= coinset[j];
    }

    return changes;
}
```

There is an additional time complexity of $\mathcal{O}(s)$ due to while loop and an additional space complexity of $\mathcal{O}(s)$ for the additional storage.

Hence total time complexity is $\mathcal{O}(ks)$ and space complexity is $\mathcal{O}(s)$.

coins	**amount**	**changes**
25, 10, 5	30	25, 5
2,3,5,6,7,8	10	2,8
1,2,5	11	1,5,5
1,2,3	4	1,4
2,5,3,6	10	5,5
9,6,5,1	11	6,5
3	5	
1	3	1,1,1

■

§ Problem 2.3.3. In § Problem 2.3.1, determine total number of ways to make change for the amount s. ◊

Algorithm 4 Coin Change : No of Ways

§§ Solution.

```
1: function ways-coin-change($c[0..k-1]$, $s$)
2:     $f[0] \leftarrow 1$                ▷ 1 way (empty set) to make change for the amount 0
3:     for $coin \in c[0..k-1]$ do
4:         for $i \leftarrow coin, s$ do
5:             $f[i] \leftarrow f[i] + f[i - coin]$
6:         end for
7:     end for
8:     return $f[s]$
9: end function
```

Time complexity is $\mathcal{O}(ks)$ and space complexity is $\mathcal{O}(s)$.

```cpp
int ways_coin_change(std::vector<int> & coins, int amount)
{
    std::vector<int> f(amount + 1, 0);

    f[0] = 1;

    for(int coin : coins)
    {
        for(int i = coin; i <= amount; ++i)
        {
            f[i] += f[i-coin];
        }
    }
    return f[amount];
}
```

There are 4 number of ways to make change for amount = 5 with the coins = {1, 2, 5}

[5, [2,2,1], [2,1,1,1], [1,1,1,1,1]].

■

2.4 Constrained Subsequence

2.4.1 Maximum Sum

§ Problem 2.4.1. 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

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

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.

Algorithm 5 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:         $currentsum \leftarrow \textbf{max}(currentsum + x, 0)$
6:         $maxsum \leftarrow \textbf{max}(maxsum, currentsum)$
7:     end for
8:     return $maxsum$
9: end function
```

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

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

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

■

§ Problem 2.4.2. In § Problem 2.4.1, identify the start and end indices of the contiguous subsequence having the largest possible sum.. ◊

§§ Solution. We need to construct the optimal solution from the computed information, i.e. identify the indices i and j such that $\underset{i \le j}{\mathrm{Max}}(s_i + .. + s_j)$.

Algorithm 6 Maximum sum contiguous subsequence : compute indices

```
 1: function maxseq($s[0..n-1]$)
 2:     $currentsum \leftarrow 0$
 3:     $maxsum \leftarrow 0$

 4:     $currentsumstart \leftarrow 0$
 5:     $maxsumstart \leftarrow 0$
 6:     $maxsumend \leftarrow 0$

 7:     for $i \in [0, n)$ do
 8:         $currentsum \leftarrow currentsum + s[i]$
 9:         if $currentsum < 0$ then
10:             $currentsum \leftarrow 0$
11:             $currentsumstart \leftarrow i + 1$
12:         else if $currentsum > maxsum$ then
13:             $maxsum \leftarrow currentsum$
14:             $maxsumstart \leftarrow currentsumstart$
15:             $maxsumend \leftarrow i$
16:         end if
17:     end for
18:     return $maxsumstart, maxsumend$
19: end function
```

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

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

    int current_sum_start = 0;
    int max_sum_start = 0;
    int max_sum_end = 0;

    int n = s.size();

    for(int i = 0; i < n; i++)
    {
        current_sum = current_sum + s[i];

        if(current_sum < 0)
        {
            current_sum = 0;
            current_sum_start = i + 1;
        }
        else
        if(current_sum > max_sum)
        {
            max_sum = current_sum;
            max_sum_start = current_sum_start;
            max_sum_end = i;
        }
    }

    if(max_sum != 0) return {max_sum_start, max_sum_end};
    else return {-1, -1};
}
```

Sequence	<Start Index, End Index>	Max Subsequence	Max Sum
34, -50, 42, 14, -5, 86	<2, 5>	42, 14, -5, 86	137
-5, -1, -8, -9	<-1, -1>		0
-2, 1, -3, 4, -1, 2, 1, -5, 4	<3, 6>	4, -1, 2, 1	6
4, -1, 2, 1	<0, 3>	4, -1, 2, 1	6
4	<0, 0>	4	4

■

§ Problem 2.4.3. Given a sequence of $n \in (-\infty,\ \infty)$ integers, determine a non-contiguous subsequence having the largest possible sum. ◊

§§ Solution. Let $f_n(i)$ be the maximum sum of a non-contiguous subsequence ending at index i, obtained using an optimal policy of a n-stage process.

Let s_i be the value of the element at index i. Since no two elements are adjacent, there are 2 options:

1. s_i is included : $\therefore f_n^{inclusive}(i) = f_n(i-2) + s_i$
2. s_i is excluded : $\therefore f_n^{exclusive}(i) = f_n(i-1)$

Hence the required optimal procedure is

$$f_n(i) = \text{Max}\{f_n(i-2) + s_i,\ f_n(i-1)\}$$
$$f_n(0) = s_0$$
$$f_n(1) = \text{Max}(s_0,\ s_1)$$

Algorithm 7 Maximum sum non-contiguous subsequence : compute sum

```
1: function maxncseq($s[0..n-1]$)
2:     $f[0..n-1] \leftarrow \{0\}$
3:     $f[0] \leftarrow s[0]$
4:     $f[1] \leftarrow$ max$(s[0],\ s[1])$

5:     for $i \in [2, n)$ do
6:         $f[i] \leftarrow$ max$(f[i-2] + s[i],\ f[i-1])$
7:     end for

8:     return $f[n-1]$
9: end function
```

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

```
int maxncseq(std::vector<int> & s)
{
    if(s.empty()) return 0;

    int n = s.size();

    std::vector<int> f(n, 0);

    f[0] = s[0];
    f[1] = std::max(s[1], s[0]);

    for(int i = 2; i < n; ++i)
    {
        f[i] = std::max(f[i-2] + s[i], f[i-1]);
    }
    return f[n-1];
}
```

Algorithm 8 Maximum sum non-contiguous subsequence : compute sum : space optimized

```
1: function maxncseq($s[0..n-1]$)
2:     $f2 \leftarrow 0$                          ▷ sum till $i-2$
3:     $f1 \leftarrow 0$                          ▷ sum till $i-1$
4:     $f \leftarrow 0$                           ▷ sum till $i$

5:     for $e \in s[0..n-1]$ do
6:         $f \leftarrow$ max$(f2 + e,\ f1)$
7:         $f2 \leftarrow f1$
8:         $f1 \leftarrow f$
9:     end for

10:    return $f1$
11: end function
```

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

```cpp
int maxncseq(std::vector<int> & s)
{
    if(s.empty()) return 0;

    int f2 = 0, f1 = 0;

    for(int e : s)
    {
        int f = std::max(f2 + e, f1);
        f2 = f1;
        f1 = f;
    }
    return f1;
}
```

Or,

```cpp
int maxncseq(std::vector<int> & s)
{
    if(s.empty()) return 0;

    int lastsum = 0, prev_maxsum = 0, maxsum = 0;

    for(int e : s)
    {
        prev_maxsum = maxsum;
        maxsum = std::max(lastsum + e, maxsum);
        lastsum = prev_maxsum;
    }
    return maxsum;
}
```

Sequence	Non Contiguous Subsequence	Max Sum
1,2,5,2	1,5	6
1,7,8,4,2	1,8,2	11

■

§ Problem 2.4.4. Given a sequence s of $n \in (-\infty,\ \infty)$ integers, find a contiguous subsequence of s having the smallest possible sum. ◊

§§ Solution. The solution is similar to § Problem 2.4.1. Let $f_n(i)$ be the minimum sum of a contiguous subsequence ending at index i, obtained using an optimal policy of a n-stage process.

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

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

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

1. leverage the previous accumulated minimum sum if $f_{n-1}(i-1) + s_i < s_i$, or
2. start afresh with a new range with s_i.

Also note that:

- If all the elements are positive, then the required sum is value of the least +ve element.
- If all the elements are negative, then the entire sequence is the required subsequence, i.e. the required sum is the sum of all the elements of the sequence.

Algorithm 9 Minimum sum contiguous subsequence

```
1: function minseq($s[0..n-1]$)
2:     $currentmin \leftarrow \infty$
3:     $minsum \leftarrow 0$
4:     for $x \in s[0..n-1]$ do
5:         $currentmin \leftarrow$ min$(currentmin + x, x)$
6:         $minsum \leftarrow$ min$(minsum, currentmin)$
7:     end for
8:     return $minsum$
9: end function
```

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

```cpp
int minseq(std::vector<int> & s)
{
    int current_min = 0;
    int min_sum = std::numeric_limits<int>::max();

    for(int x : s)
    {
        current_min = std::min(current_min + x, x);
        min_sum = std::min(min_sum, current_min);
    }
    return min_sum;
}
```

After multiplication with a negative element (say -1), maximum becomes minimum and vice versa:

Algorithm 10 Min sum contiguous subsequence : Find max of -ve

```
1: function minseq($s[0..n-1]$)
2:     $currentmax \leftarrow -\infty$
3:     $maxsum \leftarrow 0$
4:     for $x \in s[0..n-1]$ do
5:         $currentmax \leftarrow$ max$(currentmax + (-x),\ -x)$
6:         $maxsum \leftarrow$ max$(maxsum, currentmax)$
7:     end for
8:     return $-maxsum$
9: end function
```

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

```cpp
int minseq(std::vector<int> & s)
{
    int current_max = 0;
    int max_sum = std::numeric_limits<int>::min();

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

■

§ Problem 2.4.5. In § Problem 2.4.4, identify the start and end indices of the contiguous subsequence having the smallest possible sum.. ◊

§§ Solution. We need to construct the optimal solution from the computed information, i.e. identify the indices i and j such that $\underset{i \le j}{\mathrm{Min}}(s_i + .. + s_j)$.

Algorithm 11 Minimum sum contiguous subsequence : compute indices

```
 1: function minseq($s[0..n-1]$)
 2:     $currentmin \leftarrow 0$
 3:     $minsum \leftarrow \infty$

 4:     $curminstart \leftarrow 0$
 5:     $minstart \leftarrow 0$
 6:     $minend \leftarrow 0$

 7:     for $i \in [0, n)$ do
 8:         $currentmin \leftarrow currentmin + s[i]$
 9:         if $currentmin > s[i]$ then
10:             $currentmin \leftarrow s[i]$
11:             $curminstart \leftarrow i$
12:         end if
13:         if $minsum > currentmin$ then
14:             $minsum \leftarrow currentmin$
15:             $minstart \leftarrow curminstart$
16:             $minend \leftarrow i$
17:         end if
18:     end for
19:     return $minstart$, $minend$
20: end function
```

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

```cpp
std::pair<int, int> minseq(std::vector<int> & s)
{
    int current_min = 0;
    int min_sum = std::numeric_limits<int>::max();

    int curmin_start = 0, min_start = 0, min_end = 0;

    int n = s.size();

    for(int i = 0; i < n; i++)
    {
        current_min = current_min + s[i];

        if(current_min > s[i])
        {
            current_min = s[i];
            curmin_start = i;
        }

        if(min_sum > current_min)
        {
            min_sum = current_min;
            min_start = curmin_start;
            min_end = i;
        }
    }
    return {min_start, min_end};
}
```

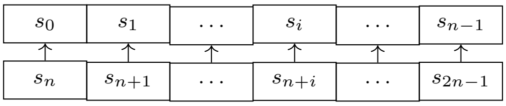
Sequence	<Start Index, End Index>	Min Subsequence	Min Sum
34, -50, 42, 14, -5, 86	<1, 4>	-50,42,-43,-5	-56
-5, -1, -8, -9	<0, 3>	-5,-1,-8,-9	-23
-2, 1, -3, 4, -1, 2, 1, -5, 4	<7, 7>	-5	-5
4, -3, 2, -5	<1, 3>	-3,2,-5	-6
4,1,5,2,3	<1,1>	1	1

■

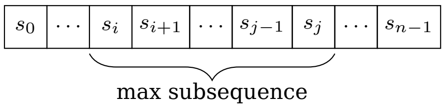
§ Problem 2.4.6. Given a circular sequence s of $n \in (-\infty, \infty)$ integers, find the maximum possible sum of a non-empty contiguous subsequence of s. ◊

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

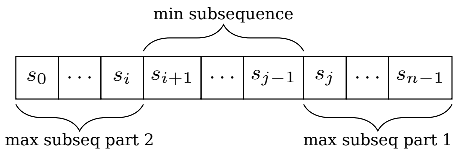
$$\because i \equiv (i+n) \textbf{ mod } n \quad \forall i \in [0,n)$$
$$\therefore s_i \equiv s_{(i+n) \textbf{ mod } n} \quad \forall i \in [0,n).$$



For a maximum contiguous subsequence $[s_i \cdots s_j]$, the solution of § Problem 2.4.1 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

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

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

Algorithm 12 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 max($maxsum,\ totalsum - minsum$)
18:    end if
19: end function
```

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

```cpp
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);
        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 - min_sum)
        ;
}
```

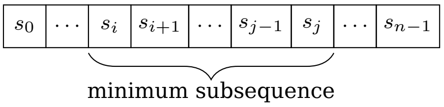
Circular Sequence	Max Sum Subsequence	Max Sum
1,-2,3,-2	3	3
5,3,-5	5,5	10
3,-1,2,-1	3,-1,2 and 2, -1, 3	4
3,-2,2,-3	3 and 3,-2,2	3
-2,-3,-1	-1	-1
8,-1,3,4	3,4,8	15
5,-3,5,5,-3	5,-3,5,5 and 5,5,-3,5	12

■

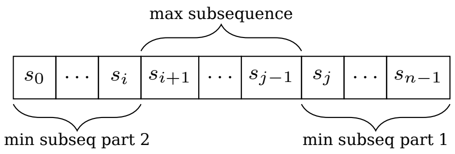
§ Problem 2.4.7. Given a circular sequence s of $n \in (-\infty, \infty)$ integers, find the minimum possible sum of a non-empty contiguous subsequence of s. $\lozenge$

§§ Solution. This is similar to § Problem 2.4.6.

For a minimum contiguous subsequence $[s_i \cdots s_j]$, the solution of § Problem 2.4.1 can be used.



For a minimum 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 maximum contiguous subsequence.



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

$$\sum_{k=0}^{k=n-1} s_k - \sum_{k=i+1}^{k=j-1} s_k$$

This is minimum when $\sum\limits_{k=i+1}^{k=j-1} s_k$ is maximum.

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

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

Algorithm 13 Minimum sum circular subsequence

```
function mincircularseq($s[0..n-1]$)
    $currentmax \leftarrow 0$
    $maxsum \leftarrow -\infty$
    $currentmin \leftarrow 0$
    $minsum \leftarrow \infty$
    $totalsum \leftarrow 0$

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

        $currentmin \leftarrow$ min$(currentmin + x, x)$
        $minsum \leftarrow$ min$(minsum, currentmin)$

        $totalsum \leftarrow totalsum + x$
    end for

    if $totalsum == maxsum$ then                    ▷ All elements are +ve
        return $minsum$                     ▷ Value of the least +ve element
    else
        return min$(minsum,\ totalsum - maxsum)$
    end if
end function
```

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

```
int minsum_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);
        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 == max_sum,
    // i.e. total_sum - max_sum becomes 0 => empty subsequence
    // but min_sum still holds the value of the least +ve element,
    // hence return this singleton than an empty one
    return total_sum == max_sum ? min_sum : std::min(min_sum, total_sum - max_sum)
        ;
}
```

Circular Sequence	Min Sum Subsequence	Min Sum
-1,2,-3,2	-3	-3
-5,3,-5	-5,-5	-10
-3,1,-2,1	-3,1,-2 and -2, 1, -3	-4
-3,2,-2,3	-3 and -3,2,-2	-3
2,3,1	1	1
-8,1,-3,-4	-3,-4,-8	-15
-5,3,-5,-5,3	-5,3,-5,-5 and -5,-5,3,-5	-12

■

§ Problem 2.4.8. Given a circular sequence s of $n \in (-\infty, \infty)$ integers, find the minimum possible sum of a non-empty non-contiguous subsequence of s. ◊

§§ Solution. This is similar to § Problem 2.4.3 with the additional constraint that the elements s_0 and s_{n-1} are adjacent, hence both elements together cannot be part of the solution.

Let $f_n(s, i, j)$ be the maximum sum of the sequence s between indices i and j, obtained using an optimal sequence of choices of a n-stage process.

$$\therefore f_n(s, 0, n-1) = \text{Max}\{f_n(s, 0, n-2),\ f_n(s, 1, n-1)\}$$

```
int maxncseq(std::vector<int> & s, int l, int r)
{
    int n = r-l+1;

    std::vector<int> f(n, 0);

    f[0] = s[l];
    f[1] = std::max(s[l], s[l+1]);

    for(int i = 2; i < n; ++i)
    {
        f[i] = std::max(f[i-2] + s[l+i], f[i-1]);
    }
    return f[n-1];
}

int maxncseq_circular(std::vector<int> & s)
{
    if(s.empty()) return 0;

    int n = s.size();

    return std::max(maxncseq(s,0,n-2), maxncseq(s,1,n-1));
}
```

Circular Sequence	Max Sum Subsequence	Max Sum
4,7,4	7	7
4,7,9,1	4,9	13

■

2.4.2 Maximum Product

§ Problem 2.4.9. Given a sequence s of $n \in (-\infty, \infty)$ integers, find a contiguous subsequence which has the largest possible product. ◊

§§ Solution. Note that the product of a running minimum with a negative element is also a running maximum so far and the product of a running maximum with a negative element is also a running minimum so far.

Let $f_n(i)$ be the maximum product, ending at index i, obtained using an optimal sequence of choices of a n-stage process and s_i be the i^{th} element.

$$f_n^{max}(i) = \underset{i\in[0,n)}{\text{Max}}\Big(s_i,\ f_{n-1}^{max}(i-1)\cdot s_i,\ f_{n-1}^{min}(i-1)\cdot s_i\Big)$$
$$f_n^{min}(i) = \underset{i\in[0,n)}{\text{Min}}\Big(s_i,\ f_{n-1}^{max}(i-1)\cdot s_i,\ f_{n-1}^{min}(i-1)\cdot s_i\Big)$$
$$\therefore f_n(i) = \underset{i\in[0,n)}{\text{Max}}\Big[f_n^{max}(i),\ f_n^{min}(i)\Big]$$

Algorithm 14 Maximum product contiguous subsequence : compute product

```
1: function maxprodseq($s[0..n-1]$)
2:     $maxprod \leftarrow 1$
3:     $minprod \leftarrow 1$
4:     $result \leftarrow 1$

5:     for $x \in s[0..n-1]$ do
6:         $prevmaxprod \leftarrow maxprod$
7:         $prevminprod \leftarrow minprod$

8:         $maxprod \leftarrow \mathbf{max}(x,\ prevmaxprod * x,\ prevminprod * x)$
9:         $minprod \leftarrow \mathbf{min}(x,\ prevmaxprod * x,\ prevminprod * x)$
10:        $result \leftarrow \mathbf{max}(maxprod,\ minprod)$
11:    end for

12:    return $result$
13: end function
```

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

```
int maxprodseq(std::vector<int> & s)
{
    if(s.empty()) return 0;

    int maxprod = 1, minprod = 1, result = 1;

    for(int x : s)
    {
        int prev_maxprod = maxprod, prev_minprod = minprod;

        maxprod = std::max(std::max(x, prev_maxprod * x), prev_minprod * x);
        minprod = std::min(std::min(x, prev_maxprod * x), prev_minprod * x);
        result = std::max(maxprod, minprod);
    }
    return result;
}
```

Algorithm 15 Maximum product contiguous subsequence : compute product : modified

```
function maxprodseq($s[0..n-1]$)
    $maxprod \leftarrow 1$
    $minprod \leftarrow 1$
    $result \leftarrow 1$

    for $x \in s[0..n-1]$ do
        if $x < 0$ then        ▷ Multiply with -ve : max becomes min and min becomes max
            swap($maxprod$, $minprod$)
        end if

        $maxprod \leftarrow$ max($x$, $maxprod * x$)
        $minprod \leftarrow$ min($x$, $minprod * x$)
        $result \leftarrow$ max($maxprod$, $minprod$)
    end for

    return $result$
end function
```

```cpp
int maxprodseq(std::vector<int> & s)
{
    if(s.empty()) return 0;

    int maxprod = 1, minprod = 1, result = 1;

    for(int x : s)
    {
        if(x < 0) std::swap(maxprod, minprod);

        maxprod = std::max(x, maxprod * x);
        minprod = std::min(x, minprod * x);
        result = std::max(maxprod, minprod);
    }
    return result;
}
```

Sequence	**Max Product Subsequence**	**Max Product**
2, 4, 5	2, 4, 5	40
-2, -4, -5	-4, -5	20
2, -4, -5	2, -4, -5	40
-1, 0, -3	0	0
0, -4, 0, -2	0	0
-1, 2, -3	-1, 2, -3	6
-3, -5, 0, -6, -5	-6, -5	30
-8	-8	-8
-2, 5, 2, -3	-2, 5, 2, -3	60
-6, -3, 3, -40	-3, 3, -40	360

■

2.5 Stock Trading

§ Problem 2.5.1. There is a sequence p of prices of a given stock over n consecutive days. Determine the maximum profit with the constraint of at most one transaction. ◊

§§ Solution. Let $f_n(i)$ be the maximum profit for selling the stock on day i, using an optimal policy of a n-stage process.

Let p_i be the price of the stock on day i. In order to maximize the profit, one has to buy at the minimum possible price and sell at the maximum possible price. Of course, a stock has to be bought before it can be sold.

Hence the required optimal procedure is

$$\therefore f_n(i>1) = \text{Max}\Big[f_{n-1}(i-1),\ p_i - \underset{j<i}{\text{Min}}\, p_j\Big]$$
$$f_n(i \le 1) = 0$$

Algorithm 16 Stock Trading : Maximum Profit : One Transaction

```
1: function max-profit($p[0..n-1]$)
2:     $maxprofit \leftarrow 0$
3:     $buyprice \leftarrow \infty$

4:     for $price \in p[0..n-1]$ do
5:         $buyprice \leftarrow$ min($buyprice,\ price$)
6:         $maxprofit \leftarrow$ max($maxprofit,\ price - buyprice$)
7:     end for

8:     return $maxprofit$
9: end function
```

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

```cpp
int max_profit(std::vector<int> & prices)
{
    // selling price >= buying price to make a profit
    int maxprofit = 0, buyprice = std::numeric_limits<int>::max();

    for(int price : prices)
    {
        buyprice = std::min(buyprice, price);
        // profit => sellingprice - buyprice
        maxprofit = std::max(maxprofit, price - buyprice);
    }
    return maxprofit;
}
```

Assuming that the stock is bought on day k and sold on day i, the profit is

$$p_i - p_k = (p_i - p_{i-1}) + (p_{i-1} - p_{i-2}) + \cdots + (p_{k+1} - p_k)$$
$$\therefore \underset{i>k}{\text{Max}}(p_i - p_k) = \text{Max} \sum_{j=i}^{j=k+1} (p_j - p_{j-1}))$$
$$= \text{Max} \sum_{j=k+1}^{j=i} (p_j - p_{j-1}))$$

So, maximizing the profit is equivalent to the maximum sum contiguous subsequence § Problem 2.4.1.

Hence the required optimal procedure is

$$f_n(i) = \text{Max}[f_{n-1}(i-1) + (p_i - p_{i-1})]$$

Algorithm 17 Maximize Profit : Maximum sum contiguous subsequence

```
1: function maxprofit($p[0..n-1]$)
2:     $currentprofit \leftarrow 0$
3:     $maxprofit \leftarrow 0$
4:     for $i \in [1, n)$ do
5:         $currentprofit \leftarrow \mathbf{max}\{currentprofit + (p_i - p_{i-1}),\ 0\}$
6:         $maxprofit \leftarrow \mathbf{max}(maxprofit,\ currentprofit)$
7:     end for
8:     return $maxprofit$
9: end function
```

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

```cpp
int max_profit(std::vector<int> & prices)
{
    int currentprofit = 0, maxprofit = 0;

    int n = prices.size();

    for(int i = 1; i < n; i++)
    {
        currentprofit = std::max(currentprofit + prices[i] - prices[i-1], 0);
        maxprofit = std::max(maxprofit, currentprofit);
    }
    return maxprofit;
}
```

In order to identify the buying day and the selling day corresponding to the maximum profit, we need to construct the optimal solution from the computed information:

Algorithm 18 Maximize Profit : Buy and Sell Days

```
1: function max-profit($p[0..n-1]$)
2:     $currentprofit \leftarrow 0$
3:     $maxprofit \leftarrow 0$

4:     $curbuyday \leftarrow 0$
5:     $buyday \leftarrow 0$
6:     $sellday \leftarrow 0$

7:     for $i \in [0, n)$ do
8:         $currentprofit \leftarrow currentprofit + p[i] - p[i-1]$        ▷ buy at $p[i-1]$, sell at $p[i]$
9:         if $currentprofit < 0$ then
10:            $currentprofit \leftarrow 0$
11:            $curbuyday \leftarrow i$        ▷ Move to the next buy day $i$
12:        else if $currentprofit > maxprofit$ then
13:            $maxprofit \leftarrow currentprofit$
14:            $buyday \leftarrow curbuyday$
15:            $sellday \leftarrow i$
16:        end if
17:    end for

18:    return $buyday,\ sellday$
19: end function
```

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

```cpp
std::pair<int, int> max_profit(std::vector<int> & prices)
{
    int currentprofit = 0, maxprofit = 0;
    int curbuyday = 0;
    int buyday = 0, sellday = 0;

    int n = prices.size();

    for(int i = 1; i < n; i++)
    {
        currentprofit = currentprofit + prices[i] - prices[i-1];

        if(currentprofit < 0)
        {
            currentprofit = 0;
            curbuyday = i;
        }
        else
        if(currentprofit > maxprofit)
        {
            maxprofit = currentprofit;
            buyday = curbuyday;
            sellday = i;
        }
    }
    if(maxprofit != 0) return {buyday, sellday};
    else return {-1, -1};
}
```

Prices	<Buy Day Index, Sell Day Index>	<Buy Price, Sell Price>	Max Profit
9,1,7,3,7,5	<1, 2>	<1, 7>	6
9,6,5,3,1	<-1, -1>		
1,3,7,9	<0, 3>	<1, 9>	8

Or

```
// returns <buy price, sell price>
std::pair<int, int> max_profit(std::vector<int> & prices)
{
    int maxprofit = 0, buyprice = std::numeric_limits<int>::max();

    for(int price : prices)
    {
        buyprice = std::min(buyprice, price);
        maxprofit = std::max(maxprofit, price - buyprice);
    }
    if(maxprofit != 0) return {buyprice, buyprice + maxprofit};
    else return {-1, -1};
}
```

■

§ Problem 2.5.2. Determine the maximum profit if at most 2 transactions are allowed in § Problem 2.5.1. ◊

§§ Solution. The logic for the first buy and sell remains the same. For the second transaction, the profit of the first transaction has to be integrated with the second buy to propagate it to the total profit with the second sell.

Algorithm 19 Stock Trading : Maximum Profit : Two Transactions

```
1: function max-profit($p[0..n-1]$)
2:     $firstbuyprice \leftarrow \infty$
3:     $firstmaxprofit \leftarrow 0$
4:     $secondbuyprice \leftarrow \infty$
5:     $finalmaxprofit \leftarrow 0$

6:     for $price \in p[0..n-1]$ do
7:         $firstbuyprice \leftarrow$ min$(firstbuyprice,\ price)$
8:         $firstmaxprofit \leftarrow$ max$(firstmaxprofit,\ price - firstbuyprice)$
9:         $secondbuyprice \leftarrow$ min$(secondbuyprice,\ price - firstmaxprofit)$
10:        $finalmaxprofit \leftarrow$ max$(finalmaxprofit,\ price - secondbuyprice)$
11:    end for

12:    return $finalmaxprofit$
13: end function
```

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

```cpp
int max_profit(std::vector<int> & prices)
{
    int firstbuyprice = std::numeric_limits<int>::max();
    int firstmaxprofit = 0, finalmaxprofit = 0;
    int secondbuyprice = std::numeric_limits<int>::max();

    for(int price : prices)
    {
        firstbuyprice = std::min(firstbuyprice, price);
        firstmaxprofit = std::max(firstmaxprofit, price - firstbuyprice);

        secondbuyprice = std::min(secondbuyprice, price - firstmaxprofit);
        finalmaxprofit = std::max(finalmaxprofit, price - secondbuyprice);
    }
    return finalmaxprofit;
}
```

Simplified presentation leads to a case of at most $m < prices.size()$ transactions :

```cpp
int max_profit(std::vector<int> & prices, int m = 2)
{
    std::vector<int> buyprice(m+1, std::numeric_limits<int>::max());
    std::vector<int> maxprofit(m+1, 0);

    for(int price : prices)
    {
        for(int i = 1; i <= m; i++) // m is number of transactions
        {
            buyprice[i] = std::min(buyprice[i], price - maxprofit[i-1]);
            maxprofit[i] = std::max(maxprofit[i], price - buyprice[i]);
        }
    }
    return maxprofit[m];
}
```

Algorithm 20 Stock Trading : Maximum Profit : $m(< n)$ Transactions

```
 1: function max-profit($p[0..n-1]$, $m$)
 2:     $buyprice[0..m] \leftarrow \infty$
 3:     $maxprofit[0..m] \leftarrow 0$

 4:     for $price \in p[0..n-1]$ do
 5:         for $i \in [1, m]$ do
 6:             $buyprice[i] \leftarrow \mathbf{min}(buyprice[i],\ price - maxprofit[i-1])$
 7:             $maxprofit[i] \leftarrow \mathbf{max}(maxprofit[i],\ price - buyprice[i])$
 8:         end for
 9:     end for

10:     return $maxprofit[m]$
11: end function
```

Time complexity is $\mathcal{O}(mn)$. Space complexity is $\mathcal{O}(m)$.

If $m > n$ then it can be simplified further (same is the case with unlimited transactions):

Algorithm 21 Stock Trading : Maximum Profit : $m(> n)$ or Unlimited Transactions

```
1: function max-profit($p[0..n-1]$)
2:     $buyprice \leftarrow \infty$
3:     $maxprofit \leftarrow 0$

4:     for $price \in p[0..n-1]$ do
5:         $buyprice \leftarrow \mathbf{min}(buyprice,\ price - maxprofit)$
6:         $maxprofit \leftarrow \mathbf{max}(maxprofit,\ price - buyprice)$
7:     end for

8:     return $maxprofit$
9: end function
```

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

```cpp
int max_profit(std::vector<int> & prices)
{
    int maxprofit = 0, buyprice = std::numeric_limits<int>::max();

    for(int price : prices)
    {
        buyprice = std::min(buyprice, price - maxprofit);
        maxprofit = std::max(maxprofit, price - buyprice);
    }
    return maxprofit;
}
```

Or,

Algorithm 22 Stock Trading : Maximum Profit : $m(> n)$ or Unlimited Transactions : Alternative

```
1: function max-profit($p[0..n-1]$)
2:     $maxprofit \leftarrow 0$

3:     for $i \in [1, p.size())$ do
4:         $maxprofit \leftarrow maxprofit + \mathbf{max}(p[i] - p[i-1],\ 0)$
5:     end for

6:     return $maxprofit$
7: end function
```

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

```
int max_profit(std::vector<int> & prices)
{
    int maxprofit = 0;
    int n = prices.size();

    for(int i = 1; i < n; i++)
    {
        maxprofit += std::max(prices[i] - prices[i-1], 0);
    }
    return maxprofit;
}
```

Prices	No of Transactions	<Buy Price, Sell Price>	Max Profit
4,4,6,0,0,3,1,9	2	<0, 3> <1, 9>	11
9,6,5,3,1	2		0
1,3,7,9	2	<1, 9>	8
4,2,9,8,0,7	2	<2, 9> <0, 7>	14
4,2,9,8,0,7,6,9	3	<2, 9> <0, 7> <6, 9>	17

Alternatively :

```
int max_profit(std::vector<int> & prices, int m = 2)
{
    std::vector<int> curprofit(m+1, 0);
    std::vector<int> maxprofit(m+1, 0);

    int n = prices.size();

    for(int i = 0; i < n-1; i++)
    {
        int dailygain = prices[i+1] - prices[i];

        for(int j = m; j >= 1; j--)
        {
            curprofit[j] = std::max(curprofit[j] + dailygain, maxprofit[j-1] + std
                ::max(dailygain, 0));
            maxprofit[j] = std::max(maxprofit[j], curprofit[j]);
        }
    }

    return maxprofit[m];
}
```

Similarly in case of unlimited transactions with a fee per transaction:

```
int max_profit(std::vector<int> & prices, int fee)
{
    int maxprofit = 0, buyprice = std::numeric_limits<int>::max();

    for(int price : prices)
    {
        buyprice = std::min(buyprice, price - maxprofit + fee);
        maxprofit = std::max(maxprofit, price - buyprice);
    }
    return maxprofit;
}
```

Prices	Fee	<Buy Price, Sell Price>	Max Profit
1,3,2,7,4,8	2	<1, 7> <4, 8>	6 [(7-1) - 2 + (8-4) - 2]

With a constraint of no buy next day of a sell, unlimited transactions:

```
int max_profit(std::vector<int> & prices)
{
    int prev_maxprofit = 0, prev_buyprice = std::numeric_limits<int>::max();
    int maxprofit = 0, buyprice = std::numeric_limits<int>::max();

    for(int price : prices)
    {
        prev_buyprice = buyprice;
        buyprice = std::min(prev_buyprice, price - prev_maxprofit);

        prev_maxprofit = maxprofit;
        maxprofit = std::max(prev_maxprofit, price - prev_buyprice);
    }
    return maxprofit;
}
```

Prices	<Buy Price, Sell Price>	Max Profit
1,3,5,1,9	<1, 3> <1, 9>	10

■

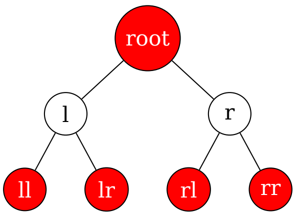
2.6 Binary Tree Mall Loot

§ Problem 2.6.1. In the Binary Tree mall with root as the only entry, all the shops are located in a binary form with a burglar alarm which comes into action in case of loot from any two directly-linked shops. Determine the maximum amount of loot, possible without raising the alarm. ◊

§§ Solution. Let $f_n(root)$ be the maximum amount of loot with entry at the root, using an optimal policy and n steps.

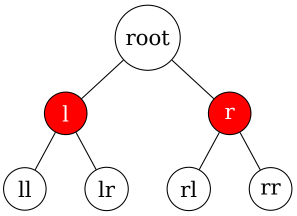
There are two choices :

1. root is looted : then it is not possible to loot its left and right shops because these two are directly linked, but next level shops can be looted : left->left, left->right, right->left and right->right.



$$\therefore f_n^{loot}(root) = amount_{root} + f_n(ll) + f_n(lr) + f_n(rl) + f_n(rr)$$

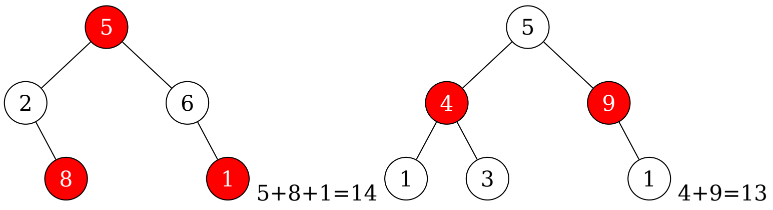
2. root is not looted : its left and right shops can be looted.



$$\therefore f_n^{no\ loot}(root) = f_n(l) + f_n(r)$$

$$\therefore f_n(root) = \text{Max}\Big\{f_n^{loot}(root),\ f_n^{no\ loot}(root)\Big\}$$

```
struct Shop
{
    int amount;
    Shop * left;
    Shop * right;

    Shop(int amt) : amount(amt), left(nullptr), right(nullptr) {}
};

std::unordered_map<Shop*, int> cache;

int loot(Shop * shop)
{
    if(not shop) return 0;

    if(cache.find(shop) != cache.end()) return cache[shop];

    int l_plus_r = loot(shop->left) + loot(shop->right);

    int ll_plus_lr = shop->left == nullptr ? 0 : loot(shop->left->left) + loot(
        shop->left->right);

    int rl_plus_rr = shop->right == nullptr ? 0 : loot(shop->right->left) + loot(
        shop->right->right);

    cache[shop] = std::max(shop->amount + ll_plus_lr + rl_plus_rr, l_plus_r);

    return cache[shop];
}
```



■

2.7 Binary Search Tree Generation

§ Problem 2.7.1. Determine the total number of binary search trees possible with $n \geq 1$ integers as keys. ◊

§§ Solution. Let $f(i)$ be the total number of binary search trees possible with root holding an integer $i \in [1, n]$ as its key, following an optimal sequence of choices.

Hence total number of unique binary search trees is

$$C_n = \sum_{i=1}^{i=n} f(i) \tag{2.3}$$

When root is i :

1. its left subtree can hold the integers from 1 to $i-1$, therefore number of left BSTs is C_{i-1}.
2. its right subtree is possible using the integers $i+1$ to n, hence number of right BSTs is C_{n-i}.

and cartesian product of these two yields $f(i)$:

$$\therefore f(i) = C_{i-1} \times C_{n-i} \tag{2.4}$$

Combining Eq. (2.3) and Eq. (2.4):

$$C_n = \sum_{i=1}^{i=n} C_{i-1} \times C_{n-i}[†]$$

$$C_0 = 1 \qquad \text{(Counting the empty BST as 1)}$$
$$C_1 = 1 \qquad \text{(Only one BST with only a root)}$$

[†]Also known as Catalan numbers.

Algorithm 23 Count Unique BSTs

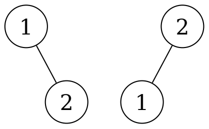
```
1: function countbst($n$)
2:     $C[0..n] \leftarrow \{0\}$
3:     $C[0] \leftarrow 1$
4:     $C[1] \leftarrow 1$

5:     for $i \in [2, n]$ do
6:         for $j \in [1, i]$ do
7:             $C[i] \leftarrow C[i] + C[j-1] \cdot C[i-j]$
8:         end for
9:     end for

10:    return $C[n]$
11: end function
```

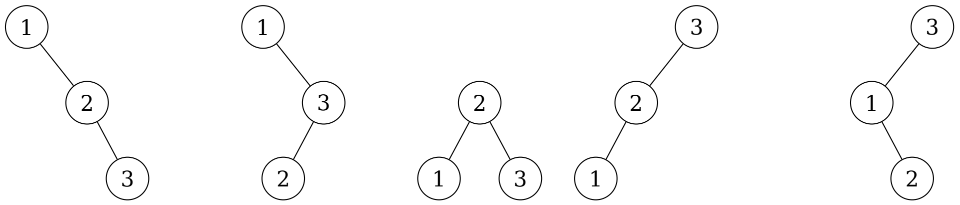
Time complexity is $\mathcal{O}(n^2)$. Space complexity is $\mathcal{O}(n)$.

```
int count_bst(int n)
{
    std::vector<int> c(n+1, 0);
    c[0] = c[1] = 1;

    for(int i = 2; i <= n; i++)
    {
        for(int j = 1; j <= i; j++)
        {
            c[i] += c[j-1] * c[i-j];
        }
    }
    return c[n];
}
```

C[2]	2
C[3]	5
C[4]	14
C[5]	42
C[6]	132

C[2]





C[3]

For generating the unique BSTs, reconstruction of the optimal solution leads to

Algorithm 24 Generate Unique BSTs

```
1: function genbst($n$)
2:     $C[0..n] \leftarrow$ {list<Node>()}
3:     $C[0].add(null)$

4:     for $i \in [1, n]$ do
5:         for $j \in [1, i]$ do
6:             for $l \in C[j-1]$ do
7:                 for $r \in C[i-j]$ do
8:                     Node tn ← new Node(j)
9:                     tn.left ← $l$
10:                    tn.right ← copyadjust(r, j)          ▷ right subtree is at offset j
11:                    $C[i].add(tn)$
12:                end for
13:            end for
14:        end for
15:    end for

16:    return $C[n]$
17: end function

18: function copyadjust(root, offset)                      ▷ Time Complexity : $\mathcal{O}(n)$
19:    Node tn ← new Node(root.key + offset)
20:    tn.left ← copyadjust(root.left, offset)
21:    tn.right ← copyadjust(root.right, offset)

22:    return $tn$
23: end function
```

Time complexity is $\mathcal{O}(n^5)$. Space complexity is $\mathcal{O}(n^2)$.

```cpp
struct tnode
{
    int key;
    tnode * left;
    tnode * right;

    tnode(int k) : key(k), left(nullptr), right(nullptr) {}
};

tnode * copyadjust(tnode * node, int offset)
{
    if(node == nullptr) return nullptr;

    tnode * tn = new tnode(node->key + offset);
    tn->left = copyadjust(node->left, offset);
    tn->right = copyadjust(node->right, offset);
    return tn;
}

std::vector<tnode*> gen_bst(int n)
{
    std::vector<std::vector<tnode*>> c(n+1);

    c[0].push_back(nullptr);

    for(int i = 1; i <= n; i++)
    {
        for(int j = 1; j <= i; j++)
        {
            for(auto l : c[j-1]) // left subtrees
            {
                for(auto r : c[i-j]) // right subtrees
                {
                    tnode * tn = new tnode(j);
                    tn->left = l; // reuse the left subtree
                    // root of the right subtree is at an offset j
                    tn->right = copyadjust(r, j);
                    c[i].push_back(tn);
                }
            }
        }
    }
    return c[n];
}
```

■

2.8 Quantify Yogic Effect

§ Problem 2.8.1. Ancient Kriya Yoga Mission, a monastry of realized sages, devised a divine yogic system : Drink Air Therapy, represented as a full binary tree, a path to self-cure and immortality.. Leaf nodes represent the techniques associated with the system. Mastery of a given technique leads to a certain gain in longevity, measured in years. Cumulative gain of a given internal node is measured by the product of the maximum gains associated with the leafs in its left and right subtrees respectively. Given a gain-list representing the years in the leaves in an inorder traversal and considering all the possible binary trees, determine the minimum possible sum of all the non-leaf nodes (i.e. minimum aggregate of the cumulative gains) to help quantify the yogic effect of the system. ♢

§§ Solution. Let $f_n(l,r)$ be the minimum aggregate of the internal nodes for a given gain-list $g[l,r]$, following an optimal sequence of choices of a n-stage process.

$$\therefore f_n(l,r) = \underset{k\in[l,r)}{\text{Min}} [f_{n-1}(l,k) + f_{n-1}(k+1,r) + \text{Max}\, g[l,k] \cdot \text{Max}\, g[k+1,r]]$$

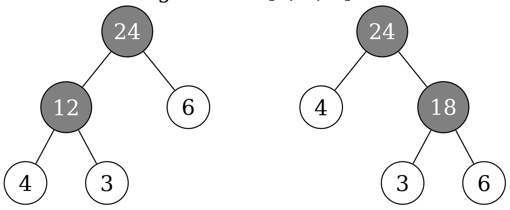
Algorithm 25 Quantify Yogic Effect : Drink Air Therapy

```
 1: function drinkairtherapy(a[0..n-1])
 2:     for $i \in [0,n)$ do
 3:         $g[i][i] \leftarrow a[i]$
 4:         $f[i][i] \leftarrow 0$
 5:     end for
 6:     for $l \in [0,n)$ do
 7:         for $i \in [0,n-l]$ do
 8:             $j \leftarrow i+l$
 9:             for $k \in [i,j)$ do
10:                 $g[i][j] \leftarrow \mathbf{max}(g[i][k],\ g[k+1][j]$
11:                 $f[i][j] \leftarrow \mathbf{min}(f[i][j],\ f[i][k] + f[k+1][j] + g[i][k] \cdot g[k+1][j])$
12:             end for
13:         end for
14:     end for
15:     return $f[0][n-1]$
16: end function
```

Time complexity is $\mathcal{O}(n^3)$. Space complexity is $\mathcal{O}(n^2)$.

```
int drinkairtherapy(std::vector<int> & v)
{
    int n = v.size(); // number of leaves

    // g[l][r] : maximum years in the leaf-nodes between [l, r]
    std::vector<std::vector<int>> g(n, std::vector<int>(n, 0));

    // f[l][r] : minimum sum of years in internal nodes between [l, r]
    std::vector<std::vector<int>> f(n, std::vector<int>(n, std::numeric_limits<int
        >::max()));

    for(int i = 0; i < n; i++)
    {
        g[i][i] = v[i];
        f[i][i] = 0;
    }

    for(int l = 0; l < n; l++)
    {
        for(int i = 0; i < n - l; i++)
        {
            int j = i + l;

            for(int k = i; k < j; k++)
            {
                // max years in leaf node
                g[i][j] = std::max(g[i][k], g[k+1][j]);

                int left = f[i][k];
                int right = f[k+1][j];

                f[i][j] = std::min(f[i][j], left + right + g[i][k] * g[k+1][j]);
            }
        }
    }
    return f[0][n-1];
}
```

gain-list : [4, 3, 6]



Minimum sum of internal nodes is 36 (12+24 < 24+18).

■

§ Problem 2.8.2. Khechari Kriya is a mysterious and divine yogic system to attain immortality. There is an ordered sequence of n sub-kriyas in the system, each bearing a specific number of years. After practicing a given sub-kriya, longevity of the practitioner is increased by the product of the years associated with that sub-kriya and its left and right adjacent sub-kriyas. Moreover that sub-kriya after practice is marked out of the sequence because it is not available for practice any further, making its left and right sub-kriyas as adjacent to each other. Determine the maximum possible number of years gained after practicing all sub-kriyas assuming that a year is associated in the absence of adjacent sub-kriya(s). ♢

§§ Solution. Let $f_n(l, r)$ be the maximum possible number of years gained for a given sequence of sub-kriyas $g[l, r]$, following an optimal sequence of choices of a n-stage process.

Let the sub-kriya i be the last one available for practice. There is no adjacent sub-kriyas per se because all except the i^{th} one were already put to practice. For simplicity, we can add one sub-kriya at the very start (i.e. l-1) and another one at very end (i.e. r+1), associating each with 1 year respectively. Hence the years gained, after practicing the last sub-kriya i, is

$$g[l-1] \cdot g[i] \cdot g[r+1] = 1 \cdot g[i] \cdot 1 = g[i]$$

Note that the sentinel entries : $g[l-1]$ and $g[r+1]$: holding 1 year each respectively, doesn't depend further on any sub-kriya as well as doesn't affect the final outcome, thus making the start of the computation easier and so on.

Hence, the optimal procedure to maximized longevity is given by :

$$f_n(l, r) = \underset{i \in [l,r]}{\text{Max}} [f_{n-1}(l, i-1) + f_{n-1}(i+1, r) + g[l-1] \cdot g[i] \cdot g[r+1]]$$

Algorithm 26 Quantify Yogic Effect : Khechari Kriya

```
 1: function khechari(g[0..n-1])
 2:     $g[0..n+1] \leftarrow 1, g[0..n-1], 1$                                  ▷ Sentinels
 3:     $f[0..n+1][0..n+1] \leftarrow \{0\}$

 4:     for $l \in [1, n]$ do
 5:         for $i \in [1, n-l+1]$ do
 6:             $j \leftarrow i + l - 1$
 7:             for $k \in [i, j]$ do
 8:                 $f[i][j] \leftarrow \mathbf{max}(f[i][j],\ f[i][k-1] + f[k+1][j] + g[i-1] \cdot g[k] \cdot g[j+1])$
 9:             end for
10:         end for
11:     end for

12:     return $f[1][n]$
13: end function
```

Time complexity is $\mathcal{O}(n^3)$. Space complexity is $\mathcal{O}(n^2)$.

```cpp
int khechari(std::vector<int> & g)
{
    int n = g.size(); // total number of sub-kriyas in khechari kriya

    // Add sentinels worth 1 year each
    g.insert(g.begin(), 1);
    g.push_back(1);

    std::vector<std::vector<int>> f(n+2, std::vector<int>(n+2, 0));

    for(int l = 1; l <= n; l++)
    {
        for(int i = 1; i <= n-l+1; i++)
        {
            int j = i+l-1;

            for(int k = i; k <= j; k++)
            {
                f[i][j] = std::max(f[i][j], f[i][k-1] + f[k+1][j] + g[i-1] * g[k]
                    * g[j+1]);
            }
        }
    }
    return f[1][n];
}
```

khechari list : [4, 1, 5, 6]

modified list	4, 1, 5, 9	4, 5, 9	4, 9	9	[]
years	$4 \cdot 1 \cdot 5$	$4 \cdot 5 \cdot 9$	$1 \cdot 4 \cdot 9$	$1 \cdot 9 \cdot 1$	$= 245$

f-matrix

$$\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 4 & 40 & 236 & \mathbf{245} & 0 \\
0 & 0 & 20 & 200 & 236 & 0 \\
0 & 0 & 0 & 45 & 54 & 0 \\
0 & 0 & 0 & 0 & 45 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

khechari list : [1, 2, 3, 4, 5]

modified list	1, 2, 3, 4, 5	1, 2, 3, 5	1, 2, 5	1, 5	5	[]
years	$3 \cdot 4 \cdot 5$	$2 \cdot 3 \cdot 5$	$1 \cdot 2 \cdot 5$	$1 \cdot 1 \cdot 5$	$1 \cdot 5 \cdot 1$	$= 110$

f-matrix

0	0	0	0	0	0	0
0	2	9	36	105	**110**	0
0	0	6	32	100	105	0
0	0	0	24	90	100	0
0	0	0	0	60	75	0
0	0	0	0	0	20	0
0	0	0	0	0	0	0

■

§ Problem 2.8.3. Mool Kriya is a fundamental yet subtle ancient yogic process to attain divinity by unblocking the breath channels. Each sub-kriya is identified by a unique integral id. Given an ordered sequence of sub-kriyas, potentially repetitive, the number of breath channels unblocked after practicing a contiguous sequence of $\alpha \geq 0$ identical sub-kriyas is α^2. After practice, the particular instance(s) of sub-kriya is(are) removed from the ordered sequence. Determine the maximum possible number of breath channels unblocked by practicing wisely. ♢

§§ Solution. Let $f(l, r, \alpha)$ be the maximum number of breath channels unblocked by practicing the sub-kriya-list $[l, r]$ such that there is a contiguous sequence of $\alpha \in [0,\ l]$ identical sub-kriyas, to the adjacent left of the sub-kriya l, with ids being the same as that of the sub-kriya l.

There are two choices with the sub-kriya l :

1. practicing it now leads to the maximal possible number of breath channels as
$$(\alpha+1)^2 + f(l+1,\ r,\ 0)$$
2. practicing it later with a identical sub-kriya $k \in (l, r]$ leads to
$$f(l+1,\ k-1,\ 0) + f(k,\ r,\ \alpha+1)$$

$$\therefore f(l,\ r,\ \alpha) = \underset{\alpha\in[0,\ l]}{\text{Max}} \begin{cases} (\alpha+1)^2 + f(l+1,\ r,\ 0) \\ f(l+1,\ k-1,\ 0) + f(k,\ r,\ \alpha+1) \quad k \in (l, r] \text{ and sub-kriya } l == \text{sub-kriya } k \end{cases}$$

Sentinels :

$$f(l,\ l-1,\ \alpha) = 0 \quad \text{(no sub-kriya : no unlocking)}$$
$$f(l,\ l,\ \alpha) = (\alpha+1)^2 \quad \text{(one sub-kriya left)}$$

Algorithm 27 Quantify Yogic Effect : Mool Kriya

```
1: function mool($s[0..n-1]$)
2:     $f[0..n-1][0..n-1][0..n-1] \leftarrow \{0\}$

3:     for $l \in [0, n)$ do
4:         for $\alpha \in [0, l]$ do
5:             $f[l][l][\alpha] \leftarrow (\alpha+1)^2$
6:         end for
7:     end for

8:     for $i \in [1,\ n)$ do
9:         for $r \in [i,\ n)$ do
10:            $l \leftarrow r - i$
11:            for $\alpha \in [0,\ l]$ do
12:                $maxbreaths \leftarrow (\alpha+1)^2 + f[l+1][r][0]$
13:                for $k \in [l+1,\ r]$ do
14:                    if $s[k] == s[l]$ then
15:                        $maxbreaths \leftarrow \mathbf{max}(maxbreaths, f[l+1][k-1][0] + f[k][r][\alpha+1])$
16:                    end if
17:                end for

18:                $f[l][r][\alpha] \leftarrow maxbreaths$
19:            end for
20:        end for
21:    end for

22:    return $f[0][n-1][0]$
23: end function
```

Time complexity is $\mathcal{O}(n^4)$. Space complexity is $\mathcal{O}(n^3)$.

```cpp
int mool(std::vector<int> & s)
{
    int n = s.size(); // total number of sub-kriyas in Mool Kriya

    int f[n][n][n] = {0};

    for(int l = 0; l < n; l++)
    {
        for(int alpha = 0; alpha <= l; alpha++)
        {
            f[l][l][alpha] = (alpha + 1) * (alpha + 1);
        }
    }

    for(int i = 1; i < n; i++)
    {
        for(int r = i; r < n; r++)
        {
            int l = r - i;

            for(int alpha = 0; alpha <= l; alpha++)
            {
                int maxbreaths = (alpha + 1) * (alpha + 1) + f[l + 1][r][0];

                for(int k = l + 1; k <= r; k++)
                {
                    if(s[k] == s[l])
                    {
                        maxbreaths = std::max(maxbreaths, f[l+1][k-1][0] + f[k][r
                            ][alpha+1]);
                    }
                }
```

```

                    f[l][r][alpha] = maxbreaths;
                }
            }
        }

        return (n == 0 ? 0 : f[0][n-1][0]);
}
```

Mool Kriya Sequence : $\{1,5,4,4,4,4,5,6,5,3,2,2,2,3,2,1\}$

Modified Sequence	**Unlocked Breath Count**
1, 5, 5, 6, 5, 3, 2, 2, 2, 3, 2, 1	4^2
1, 5, 5, 5, 3, 2, 2, 2, 3, 2, 1	1^2
1, 3, 2, 2, 2, 3, 2, 1	3^2
1, 3, 2, 2, 2, 2, 1	1^2
1, 3, 1	4^2
1, 1	1^2
[]	2^2
	$\sum$ Unlocked Breath Count = **48**

■

§ Problem 2.8.4. Tandav Kriya is a path to attain the state of Lord Shiva. The practitioner passes through intermediate states of attaining to respective gods during the course of Sadhana. Given a contiguous sequence of years of completion of the participating sub-kriyas respectively, attainment of a given state β is possible by practicing all sub-kriyas in a specific way : only a contiguous subsequence of β sub-kriyas needs to be practiced together at a time and so on. After practice, β sub-kriyas are replaced by a sub-kriya of equivalent number of years and the Sadhak continues to practice the transformed sequence subsequently. Determine the minimum possible number of years to attain a given godliness. ♢

§§ Solution. Let $f_n(l,\ r)$ be the minimum possible number of years to attain the state of god β after practicing a contiguous sequence of years of completion of the participating sub-kriyas $y[l,\ r]$, following an optimal policy with n-steps.

Note that after the practice, β is reduced to 1, i.e. $\beta - 1$ is vanished corresponding to $r - l$, i.e. length of the subsequence is now $(r-l)\ mod\ (\beta-1)+1$, which is less than β and no more vanishing is possible further. Hence the reduction of the length of $[l,\ r]$ to 1 is possible when $(r-l)\ mod\ (\beta-1) == 0$. Note that number of reductions possible is $p = \dfrac{r-l}{\beta-1}$.

$$\therefore f_n(l,\ r) = \underset{\substack{m\in[l,\ r) \\ m=l+p(\beta-1) \\ p\in\left[0,\ \frac{r-l}{\beta-1}\right]}}{\text{Min}} \left\{ f_{n-1}(l,\ m) + f_{n-1}(m+1,\ r) + \sum_{\substack{i\in[l,\ r] \\ (r-l)\ mod\ (\beta-1)==0}} y_i \right\}$$

$$f_n(l,\ l) = 0$$

Algorithm 28 Quantify Yogic Effect : Tandav Kriya

```
1: function tandav($y[0..n-1],\ \beta$)
2:     if $(n-1)$ mod $(\beta-1) \neq 0$ then                    ▷ Infeasible Solution
3:         abort
4:     end if
5:     $prefixsum[0..n] \leftarrow \{0\}$          ▷ $sum[i,\ j] \equiv prefixsum[j+1] - prefix[i]$
6:     for $i \in [0,\ n)$ do
7:         $prefixsum[i+1] \leftarrow prefixsum[i] + y[i]$
8:     end for
9:     $f[0..n-1][0..n-1] \leftarrow \{\infty\}$
10:    for $l \in [0,\ n)$ do
11:        $f[l][l] \leftarrow 0$                                   ▷ Singletons
12:    end for

13:    for $i \in [2,\ n]$ do
14:        for $l \in [0,\ n-i]$ do
15:            $r \leftarrow l + i - 1$
16:            for $m \leftarrow l,\ m < r;\ m \leftarrow m + \beta - 1$ do
17:                $f[l][r] \leftarrow$ min$(f[l][r],\ f[l][m] + f[m+1][r])$
18:            end for

19:            if $(r-l)$ mod $(\beta-1) \equiv 0$ then
20:                $f[l][r] \leftarrow f[l][r] + prefixsum[r+1] - prefixsum[l]$
21:            end if
22:        end for
23:    end for

24:    return $f[0][n-1]$
25: end function
```

Time complexity is $\mathcal{O}\left(\frac{n^3}{\beta}\right)$. Space complexity is $\mathcal{O}(n^2)$.

```cpp
int tandav(std::vector<int> & y, int beta)
{
    int n = y.size(); // total number of sub-kriyas

    // no feasible solution
    if((n-1) % (beta - 1) != 0) return -1;

    // handy to compute the sum of y[i, j] as prefixsum[j+1] - prefix[i]
    std::vector<int> prefixsum(n+1, 0);

    for(int i = 0; i < n; i++)
    {
        prefixsum[i+1] = prefixsum[i] + y[i];
    }

    // min years to transform y[l,r] to a length of (r-l)%(beta - 1) + 1
    std::vector<std::vector<int>> f(n, std::vector<int>(n, std::numeric_limits<int
        >::max()));

    for(int l = 0; l < n; l++)
    {
        f[l][l] = 0;
    }

    for(int i = 2; i <= n; i++)
    {
        for(int l = 0; l <= n-i; l++)
        {
            int r = l + i -1;

            for(int m = l; m < r; m += beta-1)
            {
                f[l][r] = std::min(f[l][r], f[l][m] + f[m+1][r]);
            }

            if((r-l) % (beta-1) == 0)
            {
                f[l][r] += prefixsum[r+1] - prefixsum[l];
            }
        }
    }

    return f[0][n-1];
}
```

Tandav Kriya Sequence : [6, 2, 8, 3]

$\beta = 2$

modified sequence	8, 8, 3	8, 11	19	[]
years	6 + 2	8 + 3	8 + 11	

$\sum$ years $= 38$

Tandav Kriya Sequence : [6, 2, 8, 3, 7]

$\beta = 3$

modified sequence	6, 13, 7	26	[]
years	2 + 8 + 3	6 + 13 + 7	

$\sum$ years $= 39$

■

§ Problem 2.8.5. Guru selects a kriya out of n ordered ones in the range $[1,\ n]$, suitable for her disciple where there is n pranayams associated with the n^{th} kriya and so on. The disciple needs to guess the selected one. Guru hints whether the guessed kriya is higher or lower. Each wrong guess costs the associated pranayams. Guru is happy if guessed right and grants a boon. Determine the pranayams needed to guarantee the boon. ◊

§§ Solution. Let $f_n(l,\ r)$ be the minimum pranayams to guarantee the boon for kriyas in the range $[l,\ r]$, following an optimal sequence of n-steps.

If the Sadhak selects the kriya m as her guess, then m pranayams are needed and now the next guess is either from $[l,\ m-1]$ or $[m+1,\ r]$ based on the hint from the Guru. She needs to account for the worst case to guarantee the boon. [‡].

Note that :

For $[1,\ 2]$: guessing 1 leads to minimum pranayams even if it is wrong because it is still smaller than guessing 2. So minimum pranayams = 1.

For $[1,\ 2,\ 3]$: guessing 2 first helps determine the correct kriya based on hint from Guru. So minimum pranayams = 2.

For $[1,\ 2,\ 3,\ 4]$: optimal strategy is to use $m \in [1,\ 4]$ to iterate over the entire sequence, then divide the left and right sequences based on m, selecting the higher pranayams plus m :

1. $m = 1$: left sequence is empty $[]$: 0 pranayam, right sequence is $[2,\ 3,\ 4]$: 3 is selected as before : total pranayams = $1 + 3 = 4$.
2. $m = 2$: left sequence is $[1]$: 0 pranayam, right sequence is $[3,\ 4]$: 3 pranayams : total pranayams = $2 + 3 = 5$.
3. $m = 3$: left sequence is $[1,\ 2]$: 1 pranayam, right sequence is $[4]$: 0 pranayam : total pranayams = $3 + 1 = 4$.
4. $m = 4$: left sequence is $[1,\ 2,\ 3]$: 2 pranayam, right sequence is empty $[]$: 0 pranayam : total pranayams = $4 + 2 = 6$.

[‡] Minimax Algorithm

Hence the minimum number of pranayams needed in the worst case to guarantee the boon being granted = 4.

Therefore the optimal procedure is

$$f_n(l,\ r) = \underset{m\in[l,\ r]}{\text{Min}}[m + \text{Max}\{f_{n-1}(l,\ m-1),\ f_{n-1}(m+1,\ r)\}]$$
$$f_n(l,\ l) = 0 \quad \text{(the only kriya must be correct)}$$

Algorithm 29 Quantify Yogic Effect : Minimax Kriya Selection

```
1: function minmaxkriya($n$)
2:     $f[0..n][0..n] \leftarrow \{0\}$

3:     for $i \in [1,\ n]$ do
4:         for $l \in [0,\ n-i]$ do
5:             $r \leftarrow l + i$
6:             $f[l][r] \leftarrow \infty$
7:             for $m \in [l,\ r)$ do
8:                 $f[l][r] \leftarrow$ min$[f[l][r],\ m +$ max$(f[l][m-1],\ f[m+1][r])]$
9:             end for
10:        end for
11:    end for

12:    return $f[0][n]$
13: end function
```

Time complexity is $\mathcal{O}(n^3)$. Space complexity is $\mathcal{O}(n^2)$.

```
int minmaxkriya(int n)
{
    std::vector<std::vector<int>> f(n+1, std::vector<int>(n+1, 0));

    for(int i = 1; i <= n; i++)
    {
        for(int l = 0; l <= n - i; l++)
        {
            int r = l + i ;

            f[l][r] = std::numeric_limits<int>::max();

            for(int m = l; m < r; m++)
            {
                f[l][r] = std::min(f[l][r], m + std::max(f[l][m-1], f[m+1][r]));
            }
        }
    }

    return f[0][n];
}
```

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Pranamayams	0	1	2	4	6	8	10	12	14	16	18	21	24	27	30

To re-iterate :

$$f_n(l,\ r) = \underset{m\in[l,\ r]}{\text{Min}}[m + \text{Max}\{f_{n-1}(l,\ m-1),\ f_{n-1}(m+1,\ r)\}]$$

Note that $f_n(l,\ r)$ is a monotonically increasing function in terms of the length of the interval $[l,\ r]$:

$$\therefore f_n(l_1,\ r) \leq f_n(l_2,\ r) \quad \forall\, l_1 \leq l_2, \text{ and}$$
$$f_n(l,\ r_1) \leq f_n(l,\ r_2) \quad \forall\, r_1 \leq r_2$$

With increasing m : length of the interval $[l,\ m-1]$ increases whereas length of the interval $[m+1,\ r]$ decreases.

In other words, $f_{n-1}(l,\ m-1)$ in a monotonically increasing function in m and $f_{n-1}(m+1,\ r)$ is a monotonically decreasing function in m.

Suppose $\exists\, m_\beta$:

$$f_{n-1}(l,\ m_\beta - 1) = f_{n-1}(m_\beta + 1,\ r)$$

$$\therefore\ \forall\, m < m_\beta,\ f_{n-1}(l,\ m-1) < f_{n-1}(l,\ m_\beta - 1) = f_{n-1}(m_\beta + 1,\ r) < f_{n-1}(m+1,\ r)$$

In other words :

$$m_\beta = \text{Max}\{m : f_{n-1}(l,\ m-1) \le f_{n-1}(m+1,\ r)\},\ \text{or}$$
$$m_\beta = \text{Min}\{m : f_{n-1}(l,\ m-1) > f_{n-1}(m+1,\ r)\}.$$

$$\therefore \text{Max}\{f_{n-1}(l,\ m-1),\ f_{n-1}(m+1,\ r)\} = \begin{cases} f_{n-1}(m+1,\ r) & \text{if } m \in [l,\ m_\beta] \\ f_{n-1}(l,\ m-1) & \text{if } m \in (m_\beta,\ r] \end{cases}$$

$$\therefore f_n(l,\ r) = \underset{m\in[l,\ r]}{\text{Min}}[f_L(l,\ r),\ f_R(l,\ r)]$$

where

$$f_L(l,\ r) = \underset{m\in[l,\ m_\beta]}{\text{Min}}\{m + f_{n-1}(m+1,\ r)\},\ \text{and}$$
$$f_R(l,\ r) = \underset{m\in(m_\beta,\ r]}{\text{Min}}\{m + f_{n-1}(l,\ m-1)\}$$
$$= 1 + m_\beta + f_{n-1}(l,\ m_\beta)\ \because \text{minimum value of } m = m_\beta + 1$$

Algorithm 30 Quantify Yogic Effect : Minimax Kriya Selection : Optimized Computation

```
 1: function minmaxkriya($n$)
 2:     $f[0..n][0..n] \leftarrow \{0\}$

 3:     for $r \in [2,\ n]$ do
 4:         $m \leftarrow r - 1$
 5:         $flmlist$ : deque<pair<fl, m> >                        ▷ $m \in [l,\ m_\beta)$

 6:         for $l \in [r-1,\ 0)$ do
 7:             while $f[l][m-1 > f[m+1][r]$ do                    ▷ Find $m_\beta$
 8:                 if flmlist is not empty and $flmlist.front().second == m$ then
 9:                     $flmlist.pop\_front()$;
10:                 end if
11:                 $m \leftarrow m - 1$
12:             end while

13:             $fl \leftarrow l + f[l+1][r]$

14:             while flmlist is not empty and $fl < flmlist.back().first$ do
15:                 $flmlist.pop\_back()$
16:             end while

17:             $flmlist.push\_back(fl, l)$

18:             $f[l][r] \leftarrow$ min$[flmlist.front().first,\ 1 + m + f[l][m]]$
19:         end for
20:     end for

21:     return $f[1][n]$
22: end function
```

Time complexity is $\mathcal{O}(n^2)$. Finding m_β is $\mathcal{O}(1)$ for a fixed $[l,\ r]$. Computation of f_L is also $\mathcal{O}(1)$ due to sliding window minimum logic for a fixed $[l,\ r]$.

Space complexity is $\mathcal{O}(n^2)$.

```
int minmaxkriya(int n)
{
    std::vector<std::vector<int>> f(n + 1, std::vector<int>(n + 1, 0));

    for(int r = 2; r <= n; r++)
    {
        int m = r - 1;

        // stores <f_L, m>, where l <= m < m_beta
        std::deque<std::pair<int, int>> flmlist;

        // find f_L(l, r) : sliding window minimum
        for(int l = r - 1; l > 0; l--)
        {
            // find m_beta
            while(f[l][m-1] > f[m+1][r])
            {
                if(not flmlist.empty() and flmlist.front().second == m)
                {
                    flmlist.pop_front();
                }
                --m;
            }

            int fl = l + f[l + 1][r];

            while(not flmlist.empty() and fl < flmlist.back().first)
            {
                flmlist.pop_back();
            }

            flmlist.emplace_back(fl, l);

            f[l][r] = std::min(flmlist.front().first, 1 + m + f[l][m]);
        }
    }

    return f[1][n];
}
```

■

§ Problem 2.8.6. Trikaldarshi is a yogic state achieved by practicing the n kriyas as the vertices $v_i : i \in [1, n]$ of a convex polygon in a triangulated way where practicing the kriyas of a triangle with vertices $< v_i, v_j, v_k >$ requires $\phi(v_i, v_j, v_k)$ years and v_i represents the number of years required for the corresponding kriya. Determine the minimum possible years required to be a Trikaldarshi. $\Diamond$

§§ Solution. Note that a triangulation of a convex polygon requires drawing all possible non-intersecting (except at a vertex) diagonals between non-adjacent vertices, thus forming $(n-2)$ triangles for a n-sided convex polygon. It is required to find the minimum possible sum of respective ϕ years of its component triangles, i.e. we need to find an optimal triangulation.

Let $f_m(i,\ j)$ be the minimum possible years to triangulate a polygon $v_i \dots v_j$ in an optimal way with m steps.

If $j < i + 2$, then there are less than 3 points, hence no triangulation is possible. Otherwise, we enumerate all $v_k : k \in (i,\ j)$ to form a triangle with vertices $< v_i, v_j, v_k >$ which in turn results into left and right polygons to triangulate and so on.

Hence, the optimal procedure is

$$f_m(i,\ j) = \begin{cases} 0 & \text{If } j < i + 2 \\ \underset{i<k<j}{\text{Min}} \{f_{m-1}(i,\ k) + f_{m-1}(k,\ j) + \phi(v_i, v_j, v_k)\} & \text{otherwise} \end{cases}$$

Algorithm 31 Quantify Yogic Effect : Trikaldarshi

```
1: function trikal($v[0..n-1]$)
2:     $f[0..n-1][0..n-1] \leftarrow \{0\}$

3:     for $d \in [2,\ n)$ do
4:         for $i \in [0,\ n-d)$ do
5:             $j \leftarrow i + d$
6:             $f[i][j] \leftarrow \infty$
7:             for $k \in [i+1,\ j)$ do
8:                 $f[i][j] \leftarrow$ min$[f[i][j],\ f[i][k] + f[k][j] + \phi(v_i, v_j, v_k)]$
9:             end for
10:        end for
11:    end for

12:    return $f[0][n-1]$
13: end function
```

Time complexity is $\mathcal{O}(n^3)$. Space complexity is $\mathcal{O}(n^2)$.

Assuming $\phi(v_i, v_j, v_k) = v[i] * v[j] * v[k]$:

```cpp
int trikal(std::vector<int> & v)
{
    int n = v.size(); // total number of kriya-vertices

    std::vector<std::vector<int>> f(n, std::vector<int>(n, 0));

    for(int d = 2; d < n; d++)
    {
        for(int i = 0; i < n - d; i++)
        {
            int j = i + d;

            f[i][j] = std::numeric_limits<int>::max();

            for(int k = i + 1; k < j; k++)
            {
                f[i][j] = std::min(f[i][j], f[i][k] + f[k][j] + v[i] * v[j] * v[k
                    ]);
            }
        }
    }

    return f[0][n-1];
}
```

Sequence of Kriya Years	**Triangulation**	**Min Years**
1, 2, 3	1*2*3	6
1, 2, 3, 4	min(1*2*3 + 1*4*3, 1*2*4 + 2*3*4)	18
1, 2, 3, 4, 5	optimal : 1*2*3 + 1*5*4 + 1*3*4	38

For determining the structure of the optimal solution, i.e. a list of the kriya-triangles participating in triangulation of the polygon can be printed after reconstruction from the optimal solution :

Algorithm 32 Quantify Yogic Effect : Trikaldarshi : Print Kriya-Triangles

```
1: function printtrikal($f$<pair<mincost, mink> >$[0..n-1][0..n-1],\ v[0..n-1],\ i,\ j$)
2:     if $j \geq i+2$ then
3:         print $v[i],\ v[f[i][j].mink],\ v[j]$

4:         $printtrikal(f, v, i, f[i][j].mink)$
5:         $printtrikal(f, v, f[i][j].mink, j)$
6:     end if
7: end function

8: function trikal($v[0..n-1]$)
9:     $f$<pair<mincost, mink> >$[0..n-1][0..n-1] \leftarrow \{0\}$    ▷ 2D matrix of pair<mincost , mink>

10:    for $d \in [2,\ n)$ do
11:        for $i \in [0,\ n-d)$ do
12:            $j \leftarrow i+d$
13:            $f[i][j].mincost \leftarrow \infty$

14:            for $k \in [i+1,\ j)$ do
15:                $localmin \leftarrow$ min$[f[i][j].mincost,\ f[i][k]+f[k][j]+\phi(v_i, v_j, v_k)]$

16:                if $localmin < f[i][j].mincost$ then
17:                    $f[i][j].mincost \leftarrow localmin$
18:                    $f[i][j].mink = k$
19:                end if
20:            end for
21:        end for
22:    end for

23:    $printtrikal(f, v, 0, n-1)$

24:    return $f[0][n-1]$
25: end function
```

Time complexity of $printtrikal$ is $\mathcal{O}(n)$ because total number of kriya-triangles is $n-2$. Hence total time complexity is $\mathcal{O}(n^3)$. Space complexity is $\mathcal{O}(n^2)$.

```cpp
void print_trikal(std::vector<std::vector<std::pair<int, int>>> & f, std::vector<
    int> & v, int i, int j)
{
    if(j >= i+2)
    {
        std::cout << "<" << v[i] << "," <<  v[f[i][j].second] << "," << v[j] << ">
            ";
        print_trikal(f, v, i, f[i][j].second);
        print_trikal(f, v, f[i][j].second, j);
    }
}

int trikal(std::vector<int> & v)
{
    int n = v.size(); // total number of kriya-vertices

    // 2-D matrix of pair<mincost, mink>
    std::vector<std::vector<std::pair<int, int>>> f(n, std::vector<std::pair<int,
        int>>(n));

    for(int d = 2; d < n; d++)
    {
        for(int i = 0; i < n - d; i++)
        {
            int j = i + d;

            f[i][j].first = std::numeric_limits<int>::max();

            for(int k = i + 1; k < j; k++)
            {
                int local_min = f[i][k].first + f[k][j].first + v[i] * v[j] * v[k
                    ];

                if(local_min < f[i][j].first)
                {
                    f[i][j].first = local_min;
                    f[i][j].second = k; // min k
                }
            }
        }
    }
    print_trikal(f, v, 0, n-1);

    return f[0][n-1].first;
}
```

Kriya-Polygon	Kriya-Triangles in Optimal Triangulation	Min Years
1, 2, 3	<1,2,3>	6
1, 2, 3, 4	<1,3,4> <1,2,3>	18
1, 2, 3, 4, 5	<1,4,5> <1,3,4> <1,2,3>	38
6, 2, 9, 8, 4, 12, 3, 5	<6,2,5> <2,3,5> <2,4,3> <2,8,4> <2,9,8> <4,12,3>	466

■

2.9 Stairway to Heaven

§ Problem 2.9.1. Following the Kriya initiation of his disciple Ram to a specific monastic order, Guru shares the details of a stairway-like mystical yogic path of n-steps to reach the Heavenly abode. Ram is allowed to take at most $\delta \le n$ steps at a given time. Determine the total number of distinct ways to reach Heaven. ◊

§§ Solution. Let $f_m(k)$ be the number of distinct ways to reach the step-k. Note that Ram can reach the step-k in one of the δ ways : a single step from the $(n-1)^{th}$ step or a step of 2 from the $(n-2)^{th}$ step, $\cdots$, or a step of δ from the $(n-\delta)^{th}$ step.

$$\therefore f_m(k) = \begin{cases} \sum\limits_{i\in[1,\ \delta]} f_{m-1}(k-i) & \text{if } k > 1 \\ 1 & \text{if } k \le 1 \end{cases}$$

Algorithm 33 Staircase to Heaven : Count Distinct Ways

```
1: function heaven($n,\ \delta$)
2:     if $\delta > n$ then
3:         return 0
4:     end if
5:     $f[0..n] \leftarrow \{0\}$
6:     $f[0] \leftarrow 1$
7:     $f[1] \leftarrow 1$

8:     for $i \in [2,\ n]$ do
9:         $s \leftarrow 0$
10:        for $j \in [1,\ \delta],\ j \le i$ do
11:            $s \leftarrow s + f[i-j]$
12:        end for
13:        $f[i] \leftarrow s$
```

```
14:     end for

15:     return $f[n]$
16: end function
```

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

```cpp
int heaven(int n, int delta)
{
    if(delta > n) return 0;

    std::vector<int> f(n + 1, 0);

    f[0] = f[1] = 1;

    for(int i = 2; i <= n; i++)
    {
        int s = 0;

        for(int j = 1; j <= delta & j <= i; j++)
        {
            s += f[i-j];
        }

        f[i] = s;
    }

    return f[n];
}
```

n	δ	**ways**	**no of ways**
2	1	[1,1]	1
2	2	[1,1] [2]	2
3	2	[1,1,1] [1,2] [2,1]	3
3	3	[1,1,1] [1,2] [2,1] [3]	4
4	2	[1,1,1,1] [1,2,1] [2,1,1] [1,1,2] [2,2]	5

Suppose that Ram is allowed to take steps only from a given sequence, say $s_i : i \in [a,\ b]$, at a given time, then

$$\therefore f_m(k) = \begin{cases} \sum\limits_{i\in[a,\ b]} f_{m-1}(k - s_i) & \text{if } k > 1 \\ 1 & \text{if } k \leq 1 \end{cases}$$

Algorithm 34 Staircase to Heaven : Count Distinct Ways with step-list

```
1: function heaven($n,\ s[0..m-1]$)
2:     $f[0..n] \leftarrow \{0\}$
3:     $f[0] \leftarrow 1$
4:     $f[1] \leftarrow 1$

5:     for $i \in [2,\ n]$ do
6:         $sum \leftarrow 0$
7:         for $e \in s[0..m-1]$ do
8:             if $e \leq i$ then
9:                 $sum \leftarrow sum + f[i-e]$
10:            end if
11:        end for
12:        $f[i] \leftarrow sum$
13:    end for

14:    return $f[n]$
15: end function
```

Time complexity is $\mathcal{O}(nm)$, m is the number of steps in the step-list. Space complexity is $\mathcal{O}(n)$.

```
int heaven(int n, std::vector<int> & s)
{
    std::vector<int> f(n + 1, 0);

    f[0] = f[1] = 1;

    int ls = s.size();

    for(int i = 2; i <= n; i++)
    {
        int sum = 0;

        for(auto e : s)
        {
            if(e <= i)
            {
                sum += f[i-e];
            }
        }

        f[i] = sum;
    }

    return f[n];
}
```

n	**step-list**	**ways**	**no of ways**
2	1, 2	[1,1] [2]	2
4	2, 4	[2,2] [4]	2
6	2, 4, 6	[2,2,2] [2,4] [4,2] [6]	4

■