Dynamic Programming Collection

Radim Urban | www.radimurban.com | 2022

This is a summary of some of the DP exercises from the first semester Computer Science course *Algorithms & Data Structures*.
I think it might be helpful as soon as you understand the basic principles of the problem solving approach. This document does not teach you dynamic programming from scratch, it rather assumes you know what you're dealing with. After going through this summary and understanding the problems in it, many of the next problem you will be dealing with next will start to look and be very similar. And that is the main goal of this summary. To have a lot of solved exercises to learn and to gain experience from.
The problems in this summary are implemented in Java, which is the language used in the course for the *Code Expert* exercises.

DP Problems in this summary

1. Longest Positive Subsequence
2. Energy for a Trip
3. Maximizing an Expression
4. Shortest Common Supersequence
5. DNA Sequence Alignment
6. Square
7. Longest Alternating Subarray
8. Palindromic Distance
9. Shuffle
10. Longest Palindromic Subsequence
11. Art Gallery
12. Partition of an Array
13. Grid Traversal
14. Levenshtein (Edit) Distance
15. Is A and B particular sum of array elements?

Longest Positive Subsequence

Task

The task is to program an algorithm that for an array, computes the length of its longest positive subarray, where a subarray is called positive if all its elements are positive.

For example, the longest posistive subarray of the array $A = \{1, -1, 3, 2, 5, -4, 5\}$ is $\{3, 2, 5\}$, and its length is 3.

Solution

The approach here is to keep track of the *longest positive subsequence* at each point of the array. Formally that is for an array element a_i the number max_i is a length of the longest positive subsequence of the array $A[1..i]$. Than we define the recursion as follows:

$$counter[i] = \begin{cases} 1 & \text{if } i = 0 \text{ and } A[i] > 0 \\ 0 & \text{if } A[i] \leq 0 \\ counter[i-1] + 1 & \text{otherwise} \end{cases}$$

and the max like this:

$$max[i] = max \begin{cases} counter[i] \\ max[i-1] \end{cases} \quad \text{for } i > 0$$

We additionally define the variable max to keep track of the biggest $counter$ so far. At the end we return the max. The table would look like this:

A	1	-1	3	2	5	-4	5
counter	1	0	1	2	3	0	1
max	1	1	1	2	3	3	**3**

Table of the entries.

The running time is $\mathcal{O}(n)$.

```
public static int longest_positive_subarray(int[] array){
        int counter = 0;
        int max = 0;
        for (int i=0; i<array.length; i++){
                if (array[i] > 0){
                        counter++;
                } else {
                        max = Math.max(counter, max);
                        counter = 0;
                }
        }
        max = Math.max(counter, max);
        return max;
}
```

Energy for a Trip

Task

Your task is to program an algorithm that computes the minimum energy to complete a trip. More formally, you need to compute the minimum energy for moving from position 0 to position n . There are walls in some positions, and you can using the following three ways to move.

- **Walk**: Move 1 position using 1 unit of energy but the ending position should NOT be a wall. For example, if you are in Position 5 but
  Position 6 is a wall, then you are not allowed to use Walk.
- **Climb**: Move 1 position using 2 units of energy.
- **Jump**: Move 6 position using 8 units of energy but the ending position should NOT be a wall.

For example, if you are in Position 5, Positions 6-10 are walls, but Position 11 is NOT a wall, then you can use *Jump*.
The input is a Boolean array A: for $1 \leq i \leq n$, if $A[i] = 1$, Position i is a wall, and if $A[i] = 0$, Position is Not a Wall.

Solution

At the very beginning the cost is clearly 0 as we do not need to move. This is our base case. After that we move based on if there is wall or not. If there is wall at position i, we have to climb the wall which costs 2. If position i is not a wall, based on whether we already are at position $i \geq 5$ we take the cheaper out of these two options: We either move back by 6 positions and jump forward again (with the cost of 8 and only applicable in case $i \geq 5$) or simply move forward with cost of 1.
Formally, we can say:

$$dp(i) = \begin{cases} 0 & \text{if } i = 0, \\ dp(i-1) + 2 & \text{road[i]} = \text{true}, \\ min \begin{cases} dp(i-1) + 1 \\ dp(i-6) + 8 \end{cases} & \text{road[i]} = \text{false and } i > 5, \\ dp(i-1) + 1 & \text{otherwise.} \end{cases}$$

The DP table could then look like this:

i	0	1	2	3	4	5	6	7	8	9	10
road	–	true	false	false	true	true	false	true	true	false	true
dp	0	2	3	4	6	8	8	10	12	12	**14**

Table of DP entries.

The running time is $\mathcal{O}(n)$.

```
private static int energy_for_trip(boolean [] road, int n) {
        int [] dp =new int[n+1];
        dp[0] = 0;
        int cost = 0;
        for (int i=1; i<=n; i++){
                if (road[i]){
                        dp[i] = dp[i-1]+2;
                } else {
                        if (i>5){
                                dp[i] = Math.min(dp[i-6]+8, dp[i-1]+1);
                        } else {
                                dp[i] = dp[i-1]+1;
                        }
                }
        }
        return dp[n];
}
```

Maximizing an Expression

Task

The task is to program an algorithm that computes the maximum value
of a given expression by inserting parentheses. More formally, for an arithmetic expression containing n integers and $n-1$ operators each which is either $+$ or $\cdot$, the
goal is to maximize the value of the expression by inserting parentheses.

For instance, if the expression is $7+4+(-3)\cdot 6+(-5)$, the maximum value is attained by
$(((7+4)+(-3))\cdot 6)+(-5)=43$.

Your solution has to be in $\mathcal{O}(n^3)$ time, where n is the number of integers in the expression.

Hint: You may require two tables, one for the maximum values and one for the minimum values. In particular, completing either table requires the other table.
Attention: You need to be very careful about how the signs affect the recursion.

For example, if you multiply numbers from intervals $[a,b]$ and $[c,d]$, then the maximum possible value of the product is $b\cdot d$ if all values a,b,c,d are positive. But the maximum is $a\cdot c$ if all values are negative, and it might also be $b\cdot c$ or $a\cdot d$ in other cases.

Solution

We use two DP tables to construct the final result, one keeping track of the smallest possible values and one for the biggest possible values. We deal with every single possible value coming out for patricular sign at every point. Then the smallest and biggest possible outcomes are saved into the respective DP table.

For example $7+4+(-3)\cdot 6+(-5)$, the two corresponding DP tables would look like this:

MinTB	**0**	**1**	**2**	**3**	**4**
0	7	11	8	-7	-12
1	Integer.MAX_VALUE	4	1	-14	-19
2	Integer.MAX_VALUE	Integer.MAX_VALUE	-3	-18	-23
3	Integer.MAX_VALUE	Integer.MAX_VALUE	Integer.MAX_VALUE	6	1
4	Integer.MAX_VALUE	Integer.MAX_VALUE	Integer.MAX_VALUE	Integer.MAX_VALUE	-5

The DP table of minimal possible outcomes

MaxTB	**0**	**1**	**2**	**3**	**4**
0	7	11	8	48	**43**
1	Integer.MIN_VALUE	4	1	6	1
2	Integer.MIN_VALUE	Integer.MIN_VALUE	-3	-18	-3
3	Integer.MIN_VALUE	Integer.MIN_VALUE	Integer.MIN_VALUE	6	1
4	Integer.MIN_VALUE	Integer.MIN_VALUE	Integer.MIN_VALUE	Integer.MIN_VALUE	-5

The DP table of maximal possible outcomes

Note also that we only need to fill the upper triangular matrix to get to the result. The $\mathcal{O}(n^3)$ implementation:

```java
// This is official solution (not mine)
private static int maximizing_an_expression(int n, int[] Operand, char[] Operator) {
        // "Operand" stores n Operands
        // "Operator" stores n-1 Operators
        int[][] MinTB=new int[n][n]; // Store the MAX values
        int[][] MaxTB=new int[n][n]; // Store the MIN values
        int i, j, k, w, l, t1, t2, t3, t4, min, max; // Temparory values
        // Initialization
        for(i=0;i<n;i++){
                for(j=0;j<n;j++){
                        MaxTB[i][j]=Integer.MIN_VALUE;
                        MinTB[i][j]=Integer.MAX_VALUE;
                }
        }
        for(i=0;i<n;i++){
                MaxTB[i][i]=Operand[i];
                MinTB[i][i]=Operand[i];
        }
        // Complete DP-Table
        for(w=1;w<=n-1;w++){
                for(i=0;i<n-w;i++){
                        j=i+w;
                        for(k=i;k<j;k++){
                                if(Operator[k]=='+'){
                                        // For the addition operation, there is no sign issue
                                        min=MinTB[i][k]+MinTB[k+1][j];
                                        max=MaxTB[i][k]+MaxTB[k+1][j];
                                } else{
                                        // Need to deal with four possible values
                                        t1=MinTB[i][k]*MinTB[k+1][j];
                                        t2=MinTB[i][k]*MaxTB[k+1][j];
                                        t3=MaxTB[i][k]*MinTB[k+1][j];
                                        t4=MaxTB[i][k]*MaxTB[k+1][j];
                                        // Select the Min value
                                        min=min(t1,t2,t3,t4);
                                        // Select the Max value
                                        max=max(t1,t2,t3,t4);
                                }
                                MinTB[i][j]=Math.min(min,minTB[i][j]);
                                MaxTB[i][j]=Math.max(max,maxTB[i][j]);
                        }
                }
        }
        return MaxTB[0][n-1];
}
public int max(int a, int b, int c, int d){
        return Math.max(a,Math.max(b,Math.max(c,d));
}
public int min(int a, int b, int c, int d){
        return Math.min(a,Math.min(b,Math.min(c,d));
}
```

Shortest Common Supersequence

Program an algorithm that computes the length of the shortest common supersequence between two given seqeuences. More formally, for an n-element sequence A and an m-element sequence B, the *shortest common supersequence* between A and B is the **shortest** sequence C such that both A and B are a subsequence of C.

```
public static int shortest_common_supersequence(int n, int m, String A, String B) {
        int[][] SCS = new int[n+1][m+1]; // DP-Table
        for (int i = 0; i<=n; i++){
                for (int j = 0; j<=m; j++){
                        if (j==0){
                                SCS[i][j] = 0;
                        } else if (i==0){
                                SCS[i][j] = 0;
                        } else {
                                if (A.charAt(i-1)==B.charAt(j-1)){
                                        SCS[i][j] = 1+SCS[i-1][j-1];
                                } else{
                                        SCS[i][j] = Math.max(SCS[i-1][j-1],
                                                             Math.max(SCS[i-1][j],
                                                                      SCS[i][j-1]));
                                }
                        }
                }
        }
        return ((n+m)-SCS[n][m]);
}
```

DNA Sequence Alignment

A DNA sequence is a string of characters from a four-character alphabet $\{A, T, G, C\}$. For a pair of two strings x and y, an alignment is given by inserting gaps into both x and y at arbitrary places until they have the same length. Each insertion of a gap costs 2. Afterwards, for each position, we have an additional cost of 1 for each position in which the extended strings do not match. Thus the aligning operations and their costs are:

- Inserting a gap costs 2
- Aligning two mismatched characters costs 1
- Aligning two matched characters costs 0

Your task is to compute the minimal cost c of aligning two DNA sequences x and y. Note that length of x does not have be the same as length of y.

Let $dp[i][j]$ denote the cost of aligning the strings $x[0..i]$ and $y[0...j]$. The DP will work as follows:

$$dp[i][j] = \begin{cases} 2 \cdot i & \text{if } j = 0 \\ 2 \cdot j & \text{if } i = 0 \\ min \begin{cases} dp[i-1][j-1] + \begin{cases} 0 & \text{if } x[i] == y[j] \\ 1 & \text{otherwise} \end{cases} \\ dp[i-1][j] + 2 \\ dp[i][j-1] + 2 \end{cases} \end{cases}$$

We then extract the solution from $dp[\text{x.length()}][\text{y.length()}]$.
The DP table for $x =$ "TGACA" and $y =$ "GACCA" yields the result 3 and looks like this:

DP	**-**	**T**	**G**	**A**	**C**	**A**
-	0	2	4	6	8	10
G	2	1	3	5	7	9
A	4	2	2	4	6	8
C	6	4	2	3	5	6
C	8	6	4	2	3	5
A	10	8	6	4	3	**3**

Table of DP entries.

This solution works in $\mathcal{O}(x.\text{length}() \cdot y.\text{length}())$.

```java
public static int DNA_Alignment(String dna1, String dna2) {
        int[][] dt = new int[dna1.length() + 1][dna2.length() + 1];
        for (int i = 0; i < dt.length; i++) {
                dt[i][0] = 2*i;
        }
        for (int i = 0; i < dt[0].length; i++) {
                dt[0][i] = 2*i;
        }
        for (int i = 1; i < dt.length; i++) {
                for (int j = 1; j < dt[0].length; j++) {
                        int same = dna1.charAt(i-1) == dna2.charAt(j-1) ? 0 : 1;
                        dt[i][j] = min(dt[i-1][j-1] +   same,
                                                dt[i][j-1] + 2,
                                                dt[i-1][j] + 2);
                }
        }
        return dt[dna1.length()][dna2.length()];
}
public static int min(int a, int b, int c) {
        return Math.min(a, Math.min(b, c));
}
```

Square (Find biggest square 1 matrix)

You are given a 2-dimensional binary matrix B having M rows and N columns filled with only 0's and 1's. Your task is to find the area of a largest square submatrix that contains only 1's.

```
private static int Square(int M, int N, int[][] B){
        int [][] DP=new int [M][N];
        if (M==0||N==0){
                return 0;
        }
        for (int i=0;i<N;i++){
                DP[0][i] = B[0][i];
        }
        for (int i=0;i<M;i++){
                DP[i][0] = B[i][0];
        }
        for (int i=1; i<M; i++){
                for (int k=1; k<N; k++){
                        if (B[i][k] == 0){
                                DP[i][k] = 0;
                        } else {
                                DP[i][k] = 1+Math.min(
                                        Math.min(DP[i-1][k],
                                                        DP[i-1][k-1]),
                                        DP[i][k-1]);
                        }
                }
        }
        int max=0;
        for (int i=0;i<M;i++){
                for (int k=0;k<N;k++){
                        max=Math.max(max,DP[i][k]);
                }
        }
        return max*max;
}
```

Longest Alternating Subarray

The task is to program an algorithm that, for an array of distinct integers, computes the length of its longest alternating subarray. A subarray is called alternating if its values form a zig-zag pattern. More precisely, for a subarray, write down comparison signs (< and >) between its elements. Then, the subarray is called alternating if the comparison signs form a sequence $(<, >, <, >, \dots)$ or $(>, <, >, <, \dots)$.

To illustrate, consider the subarray $\{1, 5, 3, 4, 2, 6\}$. Then the comparison signs are $(<, >, <, >, <)$, so the subarray is alternating. On the other hand, consider the subarray $\{1, 5, 3, 4, 6, 2\}$. Then the comparison signs are $(<, >, <, <, >)$, so the subarray is not alternating.

As a final example, for the array $\{1, 9, 2, 3, 5, 4, 6, 7, 8\}$, the longest alternating subarray has length 4 and corresponds to either $\{1, 9, 2, 3\}$ or $\{3, 5, 4, 6\}$.

```
public static int longest_alternating_subarray(int n, int[] A) {
        // n: length of A
        // A: an array of distinct integers
        int counter = 1;
        int max_length = 1;
        boolean button = (A[1]<A[2]) ? true : false;
        for (int i=2;i<=n;i++){
                if ((A[i-1]<A[i] && button) || (A[i-1]>A[i] && !button)) {
                        counter++;
                        button = !button;
                        max_length = Math.max(max_length, counter);
                } else {
                        button = (A[i]<A[i-1]) ? true : false;
                        if ((n!=i)&&((!button && A[i+1]<A[i] && A[i]>A[i-1]) ||
                                                (button && A[i+1]>A[i] && A[i]<A[i-1]))) {
                                counter = 2;
                        } else {
                                counter = 1;
                        }
                }
        }
        return max_length;
}
```

Palindromic Distance

Given a sequence A of n characters, your task is to compute the minimum number of operations that is required to turn A into a **palindrome**.
We consider the following operations:

- Change the character at any position
- Remove the character at any position
- Insert a character at any position

Let $dp(i, j)$ denote the minimum number of operations to convert substring $s[i \dots j]$ into palindrome.
More formally, for string s, the palindromic distance is:

$$dp(i,j) = \begin{cases} 0 & \text{if } i \geq j \\ \text{dp(i+1, j-1)} & \text{if } s[i] == s[j] \\ 1 + \min \begin{cases} \text{dp(i+1, j-1)} & (1) \\ \text{dp(i+1, j)} & (2) \\ \text{dp(i, j-1)} & (3) \end{cases} \end{cases}$$

The operations $(1), (2), (3)$ correspond to following operations:

(1) - ***replace*** *characters of $s[i]$ to $s[j]$ or $s[j]$ to $s[i]$*
(2) - ***remove*** *i-th character or **insert** a new character right after j-th position*
(3) - ***remove*** *j-th character or **insert** a new character right before i-th position*

For example, if s is "ETHZETHZ", the answer is 3.

DP	**E**	**T**	**H**	**Z**	**E**	**T**	**H**	**Z**
E	0	1	1	2	1	2	2	**3**
T	0	0	1	1	2	1	2	2
H	0	0	0	1	1	2	1	2
Z	0	0	0	0	1	1	2	1
E	0	0	0	0	0	1	1	1
T	0	0	0	0	0	0	1	1
H	0	0	0	0	0	0	0	1
Z	0	0	0	0	0	0	0	0

DP table entries. The result is at dp[0][n-1].

```
public static int Palindromic_Edit_Distance(char []A, int n) {
        // A: The input sequence
        // n: The length of A;
        int [][]DP_table=new int[n][n];
        for(int i=0;i<n;i++){
                DP_table[i][i]=0;
        }
        for(int i=0;i<n-1;i++){
                if(A[i]==A[i+1]){
                        DP_table[i][i+1]=0;
                } else {
                        DP_table[i][i+1]=1;
                }
        }
        //only fill up for j>i (upper triangular matrix, other values are irrelevant)
        for(int x=2;x<n;x++) {
                for(int i=0;i<n-j;i++) {
                        int min;
                        int j = i+x;
                        if(A[i]==A[j]){
                                DP_table[i][j] = DP_table[i+1][j-1];
                        } else{
                                DP_table[i][j] = 1 + min3( DP_table[i+1][j-1],
                                DP_table[i+1][j],
                                DP_table[i][j-1] );
                        }
                }
        }
        return DP_table[0][n-1];
}
int min3(int a, int b, int c){
        return Math.min(a,Math.min(b,c));
}
```

Shuffle

Given three strings A, B, C with $|A| = n, |B| = m, |C| = n + m$, your task is to decide if C is a **shuffle** of A and B. A **shuffle** of A and B is formed by merging A and B into a new string while maintaining both the internal order of the characters of A and the internal order of the characters of B .

For example, $C =$ **P**INE**AP**P**LE** is a shuffle of $A =$ **PAPLE** and $B = $ INEP.

```
public static boolean Shuffle(int n, int m, char[] A, char[] B, char[] C) {
        //The input arrays A, B and C are indexed from 1 instead of 0 in the code.
        boolean[][] DP=new boolean[n+1][m+1];
        DP[0][0] = true;
        for (int i=1;i<=n;i++){
                DP[i][0] = DP[i-1][0]&&C[i]==A[i];
        }
        for (int i=1;i<=m;i++){
                DP[0][i] = DP[0][i-1]&&C[i]==B[i];
        }
        for (int i=1;i<=n;i++){
                for (int j=1;j<=m;j++){
                        DP[i][j] = ((DP[i-1][j]&&C[i+j]==A[i])||
                                                (DP[i][j-1]&&C[i+j]==B[j]));
                }
        }
        return DP[n][m];
}
```

Longest Palindromic Subsequence

Given a sequence A of n characters, your task is to compute the length of the longest palindromic subsequence of A , i.e., the length of the longest subsequence of A that is a palindrome.

For instance, if A is **"ETHZEBEHU"**, **"HEBEH"** is the longest palindromic subsequence of A , and its length is 5.

DP	E	T	H	Z	E	B	E	H	U
E	**1**	1	1	1	3	3	3	5	**5**
T	0	**1**	1	1	1	1	3	5	5
H	0	0	**1**	1	1	1	3	5	5
Z	0	0	0	**1**	1	1	3	3	3
E	0	0	0	0	**1**	1	3	3	3
B	0	0	0	0	0	**1**	1	1	1
E	0	0	0	0	0	0	**1**	1	1
H	0	0	0	0	0	0	0	**1**	1
U	0	0	0	0	0	0	0	0	**1**

Table showing the entries of the DP table. The result is at dp[0][n-1] and the base cases are in **bold**.

The running time should be $\mathcal{O}(n^2)$.

```
//This is an official solution (not mine)
public static int Palindrome(char[] A, int n) {
        int [][] DP_table=new int[n][n];
        for(int i=0;i<n;i++)
                DP_table[i][i]=1;
                for(int len=1;len<n;len++) {
                        for(int i=0;i<n-len;i++) {
                        int j=i+len;
                        if(A[i]==A[j]) {
                                DP_table[i][j]=2+DP_table[i+1][j-1];
                        } else {
                                if(DP_table[i][j-1]>= DP_table[i+1][j])
                                        DP_table[i][j]=DP_table[i][j-1];
                                else
                                        DP_table[i][j]=DP_table[i+1][j];
                        }
                }
        }
        return DP_table[0][n-1];
}
```

Art Gallery

A gang of thieves is robbing an art gallery. They own a truck with a volume of V which they plan to fill with valuable sculptures. There are n sculptures in the gallery, the n-th of which is guarded by a sophisticated alarm system that requires $t \geq 0$ minutes to disable. The i-th sculpture has a value $p_i > 0$ on the black market and occupies $v_i > 0$ in the truck.

The thieves only have T minutes before the police arrives. The task is to design an algorithm that computes the maximum amount of money they can make form the heist.

The program should run in time $\mathcal{O}(nVT)$ with reasonable hidden constants).

```
static int solve(int n, int V, int T, int[] volume, int[] time, int[] price) {
        int[][][] dp = new int[n+1][V+1][T+1];
        for(int i = 1; i <= n; ++i){
                for(int j = 0; j <= V; ++j){
                        for(int k = 0; k <= T; ++k){
                                if(j >= volume[i] && k >= time[i]){
                                        dp[i][j][k] = Math.max(dp[i-1][j][k],
                                                                                                dp[i-1][j-volum
                                                                                                + price[i]);
                                } else {
                                        dp[i][j][k] = dp[i-1][j][k];
                        }
                        }
                }
        }
        return dp[n][V][T];
}
```

Partition of an Array

You are given an array of n natural numbers $a_1, \ldots, a_n \in \mathbb{N}$ summing to A, which is a multiple of 3.

You want to determine whether it is possible to partition $\{1, \ldots, n\}$ into three disjoint subsets I, J, K such that the corresponding elements of the array yield the same sum, that is $|I| = |J| = |K| = \frac{A}{3}$.

For example, the answer for the input $\{2, 4, 8, 1, 4, 5, 3\}$ is **yes**, because there is the partition $(3, 4), (2, 6), (1, 5, 7)$ - corresponding to the subarrays $\{8, 1\}, \{4, 5\}, \{2, 4, 3\}$, which are all summing to 9. On the other hand, the answer for the input $\{3, 2, 5, 2\}$ is **no**.

Provide a dynamic programming algorithm that determines whether such a partition exists.

Your algorithm should have an $\mathcal{O}(nA^2)$ runtime.

```
//not tested, only demonstrates the approach
public static boolean solution(int[] array, int n){
        int A = 0; //sum of the array elements
        for(int a:array){
                A += a;
        }
        boolean[][] dp = new boolean[n+1][A/3 + 1][A/3 + 1];
        //Base Cases
        dp[0][0][0] = 1;
        for (int j = 0; j<=(A/3); j++){
                for (int k = 0; k<=(A/3); k++){
                        if (!(j==0 && k==0)){
                                dp[0][j][k] = 0
                        }
                }
        }
        //Rest of the table
        for (int i = 1; j<=n; j++){
                for (int j = 0; j<=(A/3); j++){
                        for (int k = 0; k<=(A/3); k++){
                                dp[i][j][k] = dp[i-1][j][k] ||
                                        dp[i-1][j-array[i]][k] ||
                                        dp[i-1][j][k-array[i]];
                        }
                }
        }
        return dp[n][A/3][A/3];
}
```

Grid Traversal

Given is a square grid of size $N \times N$. Rows and columns are indexed from 0 to $N-1$ and each cell at index (row, column) contains a positive value (its cost) as shown in the example below. You are asked to traverse the grid from the top row to the bottom row, one row at a time. In each step, you can move from a cell (i, j) in row i to either of the cells $(i+1, j-1)$, $(i+1, j)$, or $(i+1; j+1)$ in row $i+1$. As start you can pick any cell in the first row. The overall cost of a traversal is the sum of the costs of all visited cells, including start and end.

Devise an algorithm to find a path from top to bottom of the grid that minimizes the overall traversal cost.

The algorithm should work in $\mathcal{O}(N^2)$.

```
public static int solveGrid (int [][] grid) {
        int n = grid.length;
        int dp[][] = new int[n][n];
        for (int i=0;i<n;i++){
                dp[0][i] = grid[0][i];
        }
        for (int i=1;i<n;i++){
                for (int j=0;j<n;j++){
                        if (j!=0 && j!=n-1){
                                dp[i][j] = grid[i][j] + min3(dp[i-1][j], dp[i-1][j-1], dp[i-1][j+1
                        } else if (j==0 && j!=n-1){
                                dp[i][j] = grid[i][j] + Math.min(dp[i-1][j], dp[i-1][j+1]);
                        } else if (j!=0 && j==n-1){
                                dp[i][j] = grid[i][j] + Math.min(dp[i-1][j], dp[i-1][j-1]);
                        } else {
                                dp[i][j] = grid[i][j] + dp[i-1][j];
                        }
                }
        }
        int res = Integer.MAX_VALUE;
        for (int i=0;i<n;i++){
                if (dp[n-1][i] < res){
                        res = dp[n-1][i];
                }
        }
        return res;
}
public static int min3(int a, int b, int c) {
        return Math.min(a, Math.min(b, c));
}
```

Levenshtein (Edit) Distance

Levenshtein distance is a string metric for measuring the difference between two sequences. Informally, the Levenshtein distance between two words is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other.

The expected runtime is $\mathcal{O}(|a| \cdot |b|)$.

More formally, for two strings a and b, the Leveshtein (Edit) Distance is:

$$dp(i,j) = \begin{cases} \text{i} & \text{if } j == 0 \\ \text{j} & \text{if } i == 0 \\ \text{dp(i-1, j-1)} & \text{if } a[i] == a[j] \\ 1+\min \begin{cases} \text{dp(i-1, j)} \\ \text{dp(i, j-1)} \\ \text{dp(i-1, j-1)} \end{cases} & \text{otherwise} \end{cases}$$

Example: For strings $a =$ "OTTAWA" and $b =$ "ORLOVA", the result is 4 and we would get following DP table:

DP	-	O	T	T	A	W	A
-	0	1	2	3	4	5	6
O	1	0	1	2	3	4	5
R	2	1	1	2	3	4	5
L	3	2	2	2	3	4	5
O	4	3	3	3	3	4	4
V	5	4	4	4	4	4	5
A	6	5	5	5	5	5	4

Table with the DP entries.

```
//not tested, just demonstrating approach
public static int levenhstein(String A, String B){
        int a = A.length();
        int b = B.length();
        int[][] dp = new int[a+1][b+1];
        //Base Case
        for (int i=0; i<=a; i++){
                dp[i][0] = i;
        }
        for (int i=0; i<=b; i++){
                dp[0][i] = i;
        }
        for (int i=1; i<=a; i++){
                for (int j=1; j<=b; j++){
                        if (A.charAt(i-1) == B.charAt(j-1)){
                                dp[i][j] = dp[i-1][j-1];
                        } else {
                                dp[i][j] = 1 + min3( dp[i-1][j],
                                dp[i][j-1],
                                dp[i-1][j-1]);
                        }
                }
        }
        return dp[a][b];
}
static int min3(int a, int b, int c){
        return Math.min(a, Math.min(b,c));
}
```

Is A and B particular sum of array elements?

Given is an array X with n elements $(x_1, \ldots, x_n)$. Additionally, two numbers A and B are given.

We want to find out if A is a sum of elements in X and B is a sum of elements squared and in X. Formally, we want to know, if $A = \sum_{i=1}^{n}(x_i)$ and $B = \sum_{i=1}^{n}(x_i^2)$ is true.

Let's define the dp table with dimension $(n+1) \times (A+1) \times (B+1)$ and the recursion as follows:

$$dp[i][j][k] = \begin{cases} \text{true} & \text{if } i = j = k = 0 \\ \text{false} & \text{if } i = 0 \text{ and } i \neq j \text{ and } i \neq k \\ dp[i-1][j][k] \text{ or } dp[i-1][j-x_i][k-x_i^2] & \text{otherwise} \end{cases}$$

We can implement this in $\mathcal{O}(nAB)$. The solution is at $dp[n][A][B]$.

```
public static boolean summation(int[] array, int n, int A, int B) {
        boolean dp[][][] = new boolean[n+1][A+1][B+1];
        dp[0][0][0] = true;
        for (int i = 0; i<=A; i++) {
                for (int j = 0; j<=B; j++) {
                        if (j!=0 || i!=0) {
                                dp[0][i][j] = false;
                        }
                }
        }
        for (int i = 1; i<=n; i++) {
                for (int j = 1; j<=A; j++) {
                        for (int k = 1; k<=B; k++) {
                                if (j>=array[i-1] && k>=(array[i-1]*array[i-1])) {
                                        dp[i][j][k] = dp[i-1][j][k] ||
                                                dp[i-1][j-array[i-1]][k-(array[i-1]*array[i-1])];
                                } else {
                                        dp[i][j][k] = dp[i-1][j][k];
                                }
                        }
                }
        }
        return dp[n][A][B];
}
```