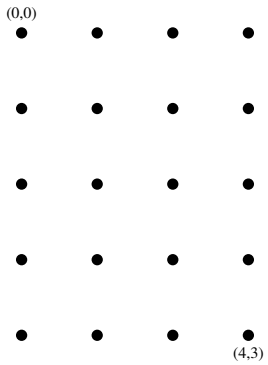
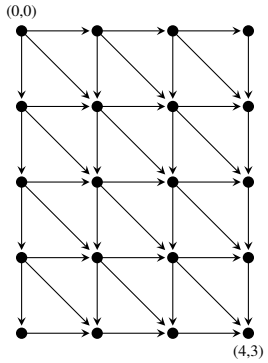
[redacted]

Least Cost Path in a Rectangular Grid with Diagonal Moves

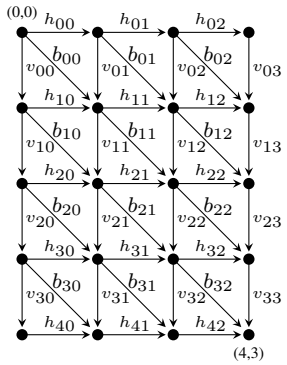
Consider a rectangular grid points/nodes at locations (i, j) in a matrix where i and j are integers with $0 \leq i \leq m$ and $0 \leq j \leq n$. Here, as usual for locations in a matrix, the index i refers to a row and index j refers to a column. For example, we can take $m = 4$ and $n = 3$ and our grid is shown in the following figure.



Assume that for each node we connect it to its neighbor on its right (if there is one) by a horizontal arrow, and to its neighbor below by a vertical arrow (if there is one). In addition, we allow or the possibility of a move along a diagonal, from a node (i, j) to $(i + 1, j + 1)$.

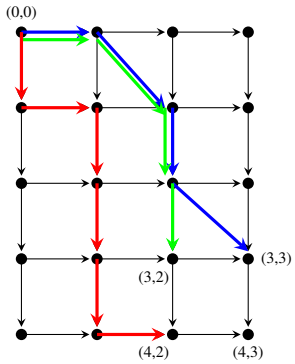


Assume each horizontal arrow from (i, j) to $(i, j + 1)$ is assigned a cost denoted by h_{ij}, each vertical arrow from (i, j) to $(i + 1, j)$ is assigned a cost v_{ij}, and each diagonal arrow from (i, j) to $(i + 1, j + 1)$ is assigned a cost b_{ij} (here "b" stands for "both").



The problem is to find the least cost path from $(0, 0)$ to (m, n) along a sequence of arrows, where the cost of a path is the sum of the costs at all of its individual arrows. This is an example of a problem that can be solved using dynamic programming. The basic idea is to work backwards. In order to get from $(0, 0)$ to (m, n) we first have to first visit either the node $(m - 1, n - 1)$, or $(m, n - 1)$ or $(m - 1, n)$.

For example, suppose we already know the best path from $(0, 0)$ to $(4, 2)$ shown in red, the best path from $(0, 0)$ to $(3, 3)$ shown in blue, and the best path from $(0, 0)$ to $(3, 2)$ as in the following figure.



Then we need only decide between the completions of three possible paths.

The best path from $(0, 0)$ to $(4, 3)$ would involve following the red path and then a final horizontal move, taking the blue path followed by a final vertical, or following the green path then a final diagonal move. Assuming the cost of the red path is $C_{4,2}$ the cost of it plus the final move is

$$C_{4,2} + h_{42}.$$

If the cost of the blue path is $C_{3,3}$ then the cost of that path plus the final move is

$$C_{3,3} + v_{33}.$$

Finally, if the cost of the green path is $C_{3,2}$ then the cost of that path plus the final move is

$$C_{3,2} + b_{32}.$$

Comparing these three costs leads to our decision as to which path to take. If there is a tie, we can any pathe that leads to a minimum cost. We then conclude that

$$C_{4,3} = \min\{C_{4,2} + h_{42}, C_{3,3} + v_{33}, C_{2,3} + b_{32}\}.$$

We refer a node (i', j') as a *predecessor* of a node (i, j) if there is an arrow pointing from (i', j') to (i, j). Observer that a node can have 0, 1, 2 or 3 predecessors.

- $(0, 0)$ has 0 predecessors,
- $(i, 0)$ has 1 predecessor if $i > 0$,
- $(0, j)$ has 1 predecessor if $j > 0$, and
- (i, j) has 3 predecessors if $i > 0$ and $j > 0$.

For any node (i, j) we denote the cost of an optimal path from $(0, 0)$ to (i, j) by $C_{i,j}$

We define a *direction* $d_{i,j}$ associated with any node with at least one predecessor.

(a) If there is a least cost path from $(0, 0)$ to (i, j) whose final step is from $(i - 1, j)$ to (i, j) we define $d_{ij} = V$.

(b) If (a) is not true and there is a least cost path from $(0, 0)$ to (i, j) whose final step is from $(i, j - 1)$ to (i, j) and we define $d_{ij} = H$.

(c) If (a) and (b) are not true, there must be a least cost path from $(0, 0)$ to (i, j) whose final step is from $(i - 1, j - 1)$ to (i, j), and we define $d_{ij} = B$.

Key observation. Once we have determined $d_{i,j}$ for every node (i, j) we can find an optimal path from $(0, 0)$ to (m, n) working backwards from (m, n) to find out least cost path from $(0, 0)$ to (m, n).

- If $d_{m,n} = H$ then by moving along an optimal path to $(m, n - 1)$ followed by a horizontal move, we get an optimal path to (m, n).

- If $d_{m,n} = V$ then by moving along an optimal path to $(m-1, n)$ followed by a vertical move, we get an optimal path to (m, n).
- If $d_{m,n} = B$ then by moving along an optimal path to $(m-1, n-1)$ followed by a diagonal vertical move, we get an optimal path to (m, n).

Whichever node we find it would be best to have visited, we repeat this process to determine the previously visited node, until we get to $(0, 0)$.

Getting started

To get started with the algorithm, note that there is only one path along the right hand border of our grid from $(0, 0)$ to $(m, 0)$ so we can initialize $C_{0,0} = 0$ and $C_{i,0} = \sum_{p=0}^{m-1} v_{p0}$ for $i = 1, \ldots, m$, and similarly there is only one path across the top from $(0, 0)$ to $(0, n)$ we can immediately calculate the values of $C_{0,j} = \sum_{p=0}^{n-1} h_{0p}$ for $j = 1, \ldots, n$.

Now whenever we know the values of $C_{i',j'}$ for every predecessor of (i', j') of (i, j) so we can calculate $C_{i,j}$ and $d_{i,j}$, so we proceed to compute these quantities for the nodes in the following order

$$
\begin{array}{cccccccccc}
 & (1,1) & \to & (1,2) & \to & (1,3) & \cdots & \to & (1,n) & \to \\
\to & (2,1) & \to & (2,2) & \to & (2,3) & \cdots & \to & (2,n) & \to \\
\to & (3,1) & \to & (3,2) & \to & (3,3) & \cdots & \to & (3,n) & \to \\
\to & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \to \\
\to & (m-1,1) & \to & (m-1,2) & \to & (m-1,3) & \cdots & \to & (m-1,n-1) & \to \\
\to & (m,1) & \to & (m,2) & \to & (m,3) & \cdots & \to & (m,n) &
\end{array}
$$

Observe that if we use this ordering of the nodes, whenever we reach node (i, j) we will have already found $C_{i',j'}$ and $d_{i',j'}$ for its two predecessors.

Pseudo-Code

We can now write some pseudo-code for finding a solution to our problem in the general case. The inputs to our problem are:

- m, n
- an $m \times (n+1)$ matrix $V = (v_{ij})_{i=0,\ldots,m-1}^{j=0,\ldots,n}$ with v_{ij} giving the cost of a vertical (down) move from (i, j) to $(i+1, j)$
- an $(m+1) \times n$ matrix $H = (h_{ij})_{i=0,\ldots,m}^{j=0,\ldots,n-1}$ with h_{ij} giving the cost of a horizontal (right) move from (i, j) to $(i, j+1)$.

- an $m \times n$ matrix $B = (b_{ij})_{i=0,\ldots,m-1}^{j=0,\ldots,n-1}$ with h_{ij} giving the cost of a diagonal (right and down) move from (i, j) to $(i, j + 1)$.

We'll use an $(m + 1) \times (n + 1)$ matrix $C = (c_{ij})_{i=0,\ldots,m}^{j=0,\ldots,n}$ to store the *cost* of a shortest path from $(0, 0)$ to (i, j) for $i = 0, \ldots, m$ and $j = 0, \ldots, n$.

We'll use an $(m + 1) \times (n + 1)$ matrix of strings $D = (d_{ij})_{i=0,\ldots,m}^{j=0,\ldots,n}$ to store, at each node (i, j) the value "H" if there is a best path from $(0, 0)$ to (i, j) whose last step is a move from predecessor $(i, j - 1)$ to (i, j) and if not, the value is "V" if there is a best path from $(0, 0)$ to (i, j) whose last step is a move from the predecessor $(i - 1, j)$ to (i, j), and the value "B", if neither of the above hold so there is a least cost path whose last step is a move from $(i - 1, j - 1)$ to (i, j).

Now we proceed as follows to calculate the terms in the matrices C and D.

Initialize $c_{0,0} = 0$
Initialize $c_{i,0} = \sum_{p=0}^{i-1} v_{p0}$ and $d_{i,0}$ ="V" for $i = 1, \ldots, m$.
Initialize $c_{0j} = \sum_{p=0}^{j-1} h_{0p}$ and $d_{0,j}$ ="H' for $j = 1, \ldots, n$.
For $i = 1, \ldots, m$:
  For $j = 1, \ldots, n$:
    If $C_{i,j-1} + h_{i,j-1} \leq \min\{C_{i-1,j} + v_{i-1,j}, C_{i-1,j-1} + b_{i-1,j-1}\}$,
      Take $C_{i,j} = C_{i,j-1} + h_{i,j-1}$ and $d_{i,j}$ ="H"
    Else if $C_{i-1,j} + v_{i-1,j} \leq C_{i-1,j-1} + b_{i-1,j-1}$,

      Take $C_{i,j} = C_{i-1,j} + v_{i-1,j}$ and $d_{i,j}$ ="V"
    Else
      Take $C_{i,j} = C_{i-1,j-1} + b_{i-1,j-1}$ and $d_{i,j}$ ="B"

Observe that once we run this algorithm, for every node (i, j) except for $(0, 0)$ it is the case that

- either $(i - 1, j)$ is a predecessor of (i, j) and $d_{i,j}$ ="V", or
- $(i, j - 1)$ is a predecessor of (i, j) and $d_{i,j}$ ="H", or
- $(i - 1, j - 1)$ is a predecessor of (i, j) and $d_{i,j}$ ="B". or

This gives us the cost $C_{i,j}$ of the optimal path from $(0, 0)$ to (i, j) with the cost of $(0, 0)$ to (m, n) as a special case. Next we carry out the following steps to find the actual optimal path - the list of nodes visited by the optimal path.

Initialize list of OptimalPathNodes list as $[(m, n)]$
While length of Optimal PathNodes list is less than $m + n + 1$

Let (i, j) be the last element in OptimalPathNodes.
If $(i, j-1)$ is a predecessor of (i, j) and $d_{(i,j)}$ ="H"
    Append $(i, j-1)$ to OptimalPathNodes
ElseIf $(i-1, j)$ is a predecessor of (i, j) and $d_{(i,j)}$ ="V"
    Append $(i-1, j)$ to OptimalPathNodes
Else $(i-1, j-1)$ is a predecessor of (i, j) and $d_{(i,j)}$ ="B" so
    Append $(i-1, j-1)$ to OptimalPathNodes
Reverse the OptimalPathNodes list
Return OptimalPathNodes list

Solving by brute-force

For small values of m and n it is possible to enumerate every possible path from $(0, 0)$ to (m, n). A path is a function that assigns an allowed direction to every possible possible position (i, j). We can make a list of all nodes that could possibly be visited before reaching the final node (m, n), together with a list of allowed moves from such a node. For example, if $m = 3$ and $n = 2$ the list looks like this

i	j	directions
0	0	H,V,B
0	1	H,V,B
0	2	V
1	0	H,V,B
1	1	H,V,B
1	2	V
2	0	H,V,B
2	1	H,V,B
2	2	V
3	0	H
3	1	H

Then there is a path for every choice of a direction for every node. In mathematical terms, we have 11 sets of directions, $L_{0,0} = \{H, V, B\}$, $L_{0,1} = \{H, V, B\}$, $L_{0,2} = \{V\}, \ldots$, and the set of paths corresponds to the elements of the *product set*

$$L_{0,0} \times L_{0,1} \times L_{0,2} \times \ldots \times L_{3,1} = \{(x_{0,0}, x_{0,1}, \ldots, x_{3,1}) \ : \ x_{0,0} \in L_{0,0}, x_{0,1} \in L_{0,1}, \ldots, x_{3,1} \in L_{3,1}\}.$$

In Python, we can use the itertools package to create a generator for the product of an arbitrary number of lists. For example, the following code

```
import itertools as it
prod=it.product([1,2,3],[4,5],[6,7])
for p in prod:
print(p)
```

produces the following output

```
(1,4,6)
(1,4,7)
(1,5,6)
(1,5,7)
(2,4,6)
(2,4,7)
(2,5,6)
(2,5,7)
(3,4,6)
(3,4,7)
(3,5,6)
(3,5,7)
```

If the lists to compute the product of are combined into a single list that is only known at run time, we can use the following construct to achieve the same objective.

```
import itertools as it
L=[[1,2,3],[4,5],[6,7]]
prod=it.product(*L)
for p in prod:
print(p)
```

The brute forth approach will only be practical for small values of m and n. For example, if $m = 5$ and $n = 4$ we have $m \times n = 20$ nodes with 3 choices of directions and the remaining nodes only have 1. This leads to $3^{20} = 3,486,784,401$ paths. The following table shows some values of m and n and the corresponding number of paths needed to be checked in the brute force approach to path finding.

m	n	3^{mn}
2	2	81
2	3	729
2	4	6,561
2	5	59,049
3	3	19,683
3	4	531,441
3	5	14,348,907
4	4	43,046,721

Application: String Alignment

We are going to apply what we know about this particular least cost path problem to the problem of aligning strings. This problem is particularly important in bioinformatics when, for example, comparing different strands of DNA.

Given a pair of strings, e.g. `that there is a hot dog` and
`he ate three hot-dogs` might be aligned like this:

```
t___at _there is a hot dog_
_he ate three _____hot-dogs
```

When we align the strings, we try as best we can to make the same characters appear in the same position. We are allowed to insert an underscore character in either string to make a "better" match.

In order to go any further, we need a rigorous way of measuring the quality of a match. Think of scoring the quality of the match by summing up a measure of the quality of each character/character pairing. For example, maybe we get 5 points for a perfect match of characters from both strings, 1 point for a mis-match of two characters, and 3 points when a character is paired with an underscore (_).

In the alignment above we can write down the individual scores
t is paired with _: 3 points
_ is paired with h: 3 points
_ is paired with e: 3 points
_ is paired with space: 3 points
a is paired with a: 5 points
t is paired with t: 5 points
and so on.

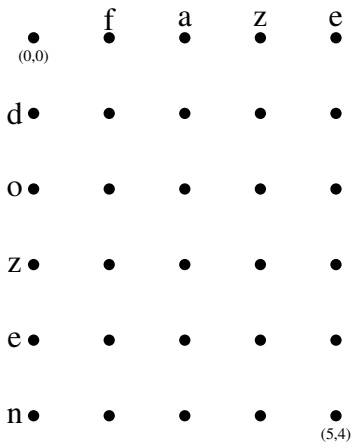
We add up the scores corresponding to the individual pairs to get

```
t___at _there is a hot dog_
_he ate three _____hot-dogs
33335513551155333335551555 1
```
giving a total score of 60.

Generally, to score alignments, assume we score a pairing of characters using a function $s(c, c')$. Here, the underscore character is a special character we use to insert space, and we assume that we use as this special character one that cannot occur in any strings to be aligned. (Otherwise we use a different character.)

We can define cost to be minus the score and try to find an alignment with minimal cost.

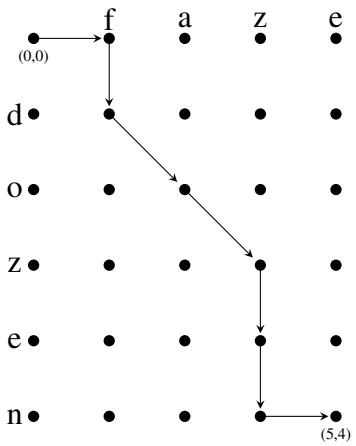
So what's the connection with the least cost path problem? An alignment can be viewed as a path in a rectangular grid. Write down the first string across the top of a grid and the second string vertically. To make things simpler, consider the two strings `faze` and `dozen`.



We start at position $(0, 0)$ with two empty strings σ and σ' and move one step at a time horizontally, vertically, or diagonally until we reach $(5, 4)$. As we move, we modify these two strings as follows:

- If we move horizontally, we add to σ the letter in the column corresponding to the node we end up in, and we add _ to σ'
- If we move vertically, we add _ to σ and add to σ' the letter in the row we end up in.
- If we move diagonally, we add to σ the letter in the column of the node we end up in and we add to σ' the letter in the row of the node we end up in.

To illustrate, suppose are moves are H, V, B, B, V, V, H (here B denotes a diagonal move - both H and V at the same time) as shown here



then the alignment is built up as follows:

Start with empty strings:
σ=
σ'=

Horizontal move leads to:
σ= f
σ'= _

Vertical move leads to:
σ= f_
σ'= _d

Diagonal move leads to:
σ= f_a
σ'= _do

Diagonal move leads to:
σ= f_az
σ'= _doz

Vertical move leads to:
σ= f_az_

σ'= _doze

Vertical move leads to:
σ= f_az__
σ'= _dozen

Horizontal move move leads to:
σ= f_az__e
σ'= _dozen_

Every possibly alignment of the two strings is associated with a path from $(0, 0)$ to $(5, 4)$. For every possible step (horizontal, vertical, both) in a path, we can calculate the score (or equivalently the cost by taking minuse the score) of the alignment by adding up the score $s(c, c')$ associated with each pairing of characters c and c' we add along the way. For every possible pairing of characters we can ever use in a path, we can calculate the score if we have been given a score function.

To find the optimal alignment, we need to find the path that minimize cost (or maximize score = -cost).