Dynamic programming problems seem tricky because it’s difficult to look beyond the naïve solution. The trick to remember is, **Dynamic Programing is a brute-force search**. The only thing is, the search happens multiple times over a small problem space, so you can save the results to get better time complexity. Thus, Dynamic Programming is used for optimization problems. When the problem space is not small and overlapping, you get an NP-hard problem.  
So, your runtime can be O(n^2) or O(n^3), your space can be O(n) or even O(n^2) (though for the simpler problems, O(1) is often possible). Once you accept this, a lot of dynamic programming problems are just find-the-pattern problems, where you find the recursive pattern that has a lot of overlapping problems.  
Another thing to notice in Dynamic programming problems is that, once you have the recursive function fun( ) that gets us the optimal, storing the *paths* upto fun(i) is consumes a lot of space (we’ll need an array or hashtable or linked list for the path). But you often don’t need to do that, because usually we are asked to get the **value** of the optimal, which is much easier than getting the path taken to get that optimal. And it is often also possible to reconstruct the path from the optimal value.

1. **Longest increasing subsequence** problem: in an array, find the length of the longest subsequence such that all elements in the subsequence are in ascending order. The subsequence does not need to be continuous.   
   eg: for a={ 10, 22, 9, 33, 21, 50, 41, 60, 80 }, longest increasing subsequence is {10, 22, 33, 50, 60, 80} of length 6.  
   **Solution:**   
   First, recognize what is the term you must maximize here: the length of the subsequence. Also recognize that in the worst case, the longest subsequence contains all the elements in the array. Naively getting all the possible subsequences of an array and checking them for the longest is an O(2^N) task (as there are at least 2^N subsequences). We can do better. The trick is to generate a recursive rule which allows us to solve this problem, even if it takes exponential time.   
   Notice this: for **any** increasing subsequence, a[i] must be appended to the subsequence if the previous element is a[j], such that j < i and a[j] < a[i].   
   Next, notice that for an element a[i], there are many increasing sequences it might belong to. Eg: a={ 10, 22, 9, 33, 21, 50, 41, 60, 80 }, for i=6, i.e. a[i]=41, 41 belongs to the following increasing sequences {41}, {10, 41}, {22, 41}, {9,41}, {33, 41}, {21,41}, {10, 22, 41}, {10, 33, 41}, {10, 21, 41}, {22, 33, 41}, {9,33,41}, {9,21,41}, {10, 22, 33, 41}. Out of these, the last one is the longest.   
   So, to get the longest increasing subsequence upto 41, *which has 41 as the last element*, we need to check all the subsequences, and pick the longest one. This is the basis of our recursive rule.  
   LIS\_including\_a\_of\_i(a, i) = 1 + best( LIS\_including\_a\_of\_i(a, j) ). We thus recursively find the best, and use that to further the calculation.   
   Now the question comes, how do we select j? The answer is, we don’t. We use brute-force and check all values of j, following the rules that j < i and a[j] < a[i]. In the base case, i.e. with one element in the array, we return 1, as the longest increasing subsequence has to at least be of length 1.   
   Concretely, here is the recursive method:  
     
   long LIS\_including(int \*a, long i){ //we find LIS of a[0...i], with a[i] at the end of the subsequence.

if (i==0)

return 1;

long lis\_including\_i=1; /\*at any given point, holds the length of the longest increasing

subsequence with a[i] at the end.\*/

for(long j=0; j < i; j++){

if (a[j] < a[i]){

long lis\_including\_j = LIS\_including(a, j);

lis\_including\_i = max( lis\_including\_i, lis\_including\_j +1 ); //updates if we find a longer increasing subsequence with a[i] at the end.

}

}

//cout<<"\ni="<<i<<" lis\_including\_i="<< lis\_including\_i;

return lis\_including\_i;

}  
  
Now, this method will get us the longest increasing subsequence with a[i] at the end. But it’s not always the case that, in an array a[0….n-1], a[n-1] is at the end of the longest increasing subsequence of the array. Eg: for a={10, 22, 9, 33, 21, 50, 41, 60, 80, 7, 36}, the longest increasing subsequence is {10, 22, 33, 50, 60, 80} of length 6. Thus, to get the true LIS, we must run the function for 0, 1 … n-1 in a loop, and take the max:  
long LIS(int \*a, long len){

long maximum=1;

for(long i=0; i<len; i++)

maximum=maxi(maximum, LIS\_including(a,i));

return maximum;

}  
  
Now, if in LIS\_including( ) you un-comment the line cout<<"\ni="<<i<<" lis\_including\_i="<< lis\_including\_i; you can see just how many times the function runs recursively, handling the same problems again and again (i.e. i=0, i=1, i=2 …). An array that memoizes the result would be very useful.   
In C++. This includes the code for the iterative solution also.   
From the iterative solution, it is quite clear that this algorithm is O(N^2) time with O(N) space, as for every element a[i], we are in the worst case checking all of its previous elements a[0..i-1].

1. \*\*\* **Largest Sum Contiguous Subarray** problem: For an array of elements (positive and negative), find the continuous subsequence with the largest sum.  
   eg: a = {-2, -3, 4, -1, -2, 1, 5, -3}, the largest sum contiguous subarray is {4, -1, -2, 1, 5} which gives the total of 7.   
     
   **Solution**: First, let’s recognize how a naïve implementation would do it: we generate all possible sets of the array (i.e. we generate the power set of the array) and then we pick the maximum element from that.  
   eg: a={1, -2, 3, 0},   
   power\_set(a) = { NULL, {1}, {-2}, {3}, {0}, {1, -2}, {1, 3}, {1, 0}, {-2, 3}, {-2, 0}, {3, 0}, {1, -2, 3}, {1, 3, 0}, {1, -2, 0}, {-2, 3, 0}, {1, -2, 3, 0} }  
   If ‘a’ has N elements, the power set of ‘a’ (which contains NULL for “no elements selected”) will have exactly 2^N elements.   
   There are NOT O(2^N) continuous subarrays, however.  
   Consider continuous subarrays of length 1: there are N  
   Consider continuous subarrays of length 2: there are N-1  
   Consider continuous subarrays of length 3: there are N-2  
   …  
   Consider continuous subarrays of length N: there is 1  
   Thus, there areN + (N-1) + (N-2) + … 3 + 2 + 1, i.e. O(N^2) continuous subarrays of size 1, 2, 3 … N respectively.   
   So, a loose upper bound for a naïve implementation is O(N^3) as any random continuous subsequence is of length O(N).  
     
   There is an algorithm which can find the answer in O(n) time and O(1) space. It is called **Kadane’s algorithm**. Basically, Kadane’s algorithm says two things:

1) Any subsequence of all positive numbers can only increase the maximum and thus *may* be part of the maximum.

2) Any negative subsequence can only be included in the maximum if it lies between two positive subsequences, both of which have greater (absolute) value than the negative subsequence.  
In Kadane, we can logically compress the array into a single file of alternating positive and negative numbers. {-2, -3, 4, -1, -2, 1, 5, -3} => {-5, 4, -3, 6, -3}.  
While iterating, we maintain a current\_sum variable. Now, we only add a group to our subarray if it increases the current sum. All positive numbers increase the sum, so include those. Adding negative numbers will increase the current sum, only if the negative number region lies between two positive number regions which are both greater than it.   
Eg: in {-5, 4, -3, 6, -3}, -3 lies between 4 and 6; the maximum sum subarray is {4, -3, 6} because |-3|<|4| and |-3|<|6|, so there is an overall gain for both 4 and 6 by being joined by -3. In another case, if we have {2, 4, -5, 6}, there is no reason to join 4 and 6…the sum of {4, -5, 6} is 5, which is less than just {6}. It makes no sense for {6} to join with {4}.  
  
One version of Kadane’s algorithm is as follows:  
  
maxSubArraySum(a, len){

max\_sum\_so\_far = 0

max\_ending\_here = 0;

for(i = 0; i < len; i++){

max\_ending\_here = a[i] + max\_ending\_here;

if (max\_ending\_here < 0)

max\_ending\_here = 0;

if (max\_sum\_so\_far < max\_ending\_here)

max\_sum\_so\_far = max\_ending\_here;

}

return max\_sum\_so\_far;

}  
  
You can see how it works on {-2, -3, 4, -1, -2, 1, 5, -3}: it starts adding numbers only from 4, and it adds -1 and -2 because the sum does not go less than zero (meaning, it is still worthwhile to use the negative subarray), and we add 1 and 5 at the end, giving us a max\_sum\_so\_far of 7, which is the value we return, as we don’t use -3.

The problem with this implementation is that it cannot deal with an array of only-negative values; it returns max\_sum\_so\_far as 0. You can modify it to work on an all-negative array by having a simple first-pass step that checks if all are negative and, if so, returns their max i.e. the least-negative.

A very compact implementation of Kadane is below. It works even for the case of an all-negative array:  
  
int max(int x, int y) {   
 return (y > x)? y : x; }

maxSubArraySum(a, len){

max\_sum = a[0];

curr\_sum = a[0];

for (i = 1; i < len; i++){

curr\_sum = max(a[i], curr\_sum + a[i]); //recursive definition of problem

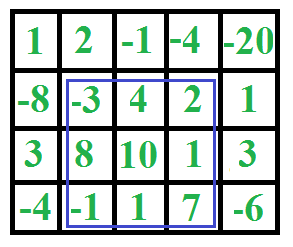
max\_sum = max(max\_sum, curr\_sum);

}

return max\_sum;

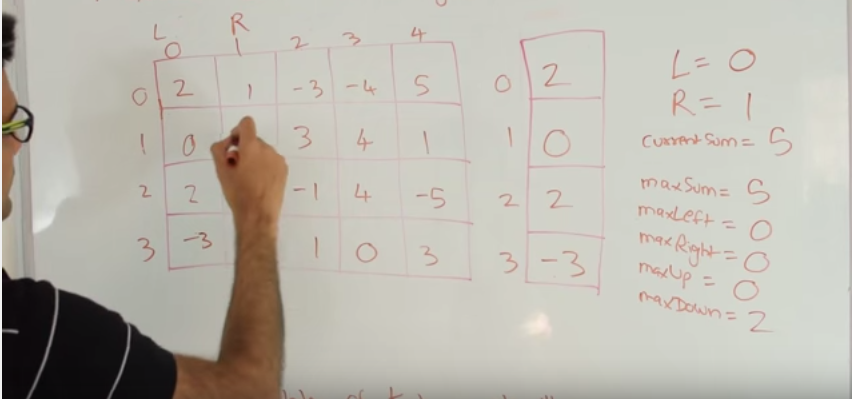
}

Mentally working on {-2, -3, -6, -1, -5} we can see how this works.   
Note: the recursive definition of this problem is this: curr\_sum = max(a[i], curr\_sum + a[i]), and we choose the best current\_sum.

1. Find the largest-sum subrectange in an MxN matrix.  
   Solution: this is an extension of the previous problem on largest contiguous sum subarray, but for a 2D array instead of 1D.  
     
     
     
   Think about the naïve solution: check every sub-matrix in the NxM matrix (i.e. N rows and M columns). How many are there? The total is the **sum of**:

* M.N [each element in a 1x1 box]
* M.(N-1) [2x1 col boxes]
* N.(M-1) [1x2 row boxes]
* M.(N-2) [3x1 col boxes]
* N.(M-2) [1x3 row boxes]
* …. [upto M.(1) and N.(1)]
* (M-1)(N-1) [2x2 square]
* (M-1)(N-2) [3x2 ‘tall’ matrix]
* (M-2)(N-1) [2x3 ‘wide’ matrix]
* ….. [Assorted]
* 1 [NxM matrix]

In the naïve, brute-force solution, generating all these matrices requires four nested loops; two which iterate from M to 1 and N to 1. We need two further nested loops to actually calculate the sum for each submatrix. Thus, this becomes an O(N^2.M^2) solution, or O(N^4) for square input matrices.  
  
There is a solution which works in O(N^3) time for a perfectly square matrix, with O(N) space. For a NxM matrix, it is O(N.M^2) with O(N) space or, O(M.N^2) with O(M) space…you can choose the smaller and optimize if you know N and M.   
This solution uses Kadane’s algorithm for a 1D array as a function to get the max sum (if you want to identify the borders the box, then it should return the max sum as well as the indices of the max sum subarray).   
A [video link](https://www.youtube.com/watch?v=yCQN096CwWM&feature=youtu.be) of the explanation of the solution. In brief:  
Note: The following explanation assumes we are using the O(M^2.N) version of the algorithm, where N=number of rows and M=number of columns:

* Start by initializing an auxiliary array which is the length of the number of rows, i.e. for each row of the NxM matrix, we have one cell in the aux array. Initialize the aux array to zero.
* Set L=0 and R=0, indicating we are selecting column matrices of different sizes from the first column.
* Copy this column into the aux array. Apply Kadane’s 1D algorithm on this aux array, to get the max sum and the indices of the max subarray. If the max sum is greater than our previous best max, update it.
* Now, make L=0 and R=1, i.e. we are selecting from the first two columns. Add the values of this second column to the values in the aux array. Thus, the aux array now contains the sum of the elements of the first two columns of the NxM matrix. Apply Kadane again to the aux array. We might get a different sum with completely different indices. The indices we get indicate that, from the first two columns, this is the top and bottom limits of the box with the largest sum. If the max sum is bigger than the max sum of any box we have seen before, update it.  
    
  

We continue this process for all valid values of L and R where L<=R. Thus, we essentially loop L and R in two nested loops, and whenever we update L, we copy the column into the aux array, overwriting the values present there. This way, we get all combinations of columns. By using Kadane’s algorithm on the rows, we know that we have considered all the possible boxes. Thus, we know we have considered all the possibilities we would if we had four nested loops, but we still only get an O(N^3) algorithm.  
Since we loop through the columns using two nested loops for L and R, we have O(M^2) iterations. For each of these iterations, we run Kadane, which is O(N), so we have an O(M^2.N) algorithm.  
We could do the same thing by maintaining an aux array for the columns instead, getting an O(M.N^2) algorithm as we iterated O(N^2) times through the rows. We can keep two separate functions and optimize if N>M or M>N.  
  
  
  
This solution does not work if all the numbers are negative, so you must place a precheck before running the solution to return the maximum element if all are negative.

findMaxSum (int\*\* M, ROW, COL){  
 maxSum = INT\_MIN;  
 int \*temp=new int[ROW];  
 for (left = 0; left < COL; ++left){  
 memset(temp, 0, sizeof(temp)); //sets all to zero  
 for (right = left; right < COL; ++right) {  
 for (i = 0; i < ROW; ++i) // Calculate sum between current L and R for every row 'i'  
 temp[i] += M[i][right];  
 sum = kadane(temp, &kadane\_start, &kadane\_finish, ROW);  
 if (sum > maxSum){  
 maxSum = sum;  
 finalLeft = left;  
 finalRight = right;  
 finalTop = kadane\_start;  
 finalBottom = kadane\_finish;  
 }  
 }  
 }   
 // Print final values if that is required  
 return maxSum  
}

1. **\*\*\*Rod cutting**: you run a steel company that sells rods. Different lengths of rods sell for different prices, eg. A 1’’ rod might sell for $1.5, a 2’’ rod might sell for $3, a 3’’ rod might sell for $4, rods of length 4’’ and 5’’ might not be sellable and hence a 6’’ rod might sell for $29, etc etc. Given this table of length-value pairs, find the optimal way to cut up and sell a rod of any given length.  
   **Solution:** first notice that if we are given a rod of length 3’’, which by the table we can sell for $4, it’s still more optimal to cut it into two pieces of 1’’ and 2’’ and sell them for $1.5 and $3 respectively, giving us $4.5 in total. Thus, just because a value is in the table, does not mean that is the most optimal value we can sell it for.  
   Second notice that depending on our table, some values are just not possible. Eg: if we have a table with just 5’’ and 9’’ in it, it is not possible to find the cost of rods of length 1’’, 2’’, 3’’, 4’’, 6’’, 7’’, 8’’, 11’’, 12’’, 13’’, 16’’, 17’’, etc. because we simply cannot make them.   
   We build the solution from the bottom up, using an array to store values of previously calculated lengths.  
   The rest of the solution is as follows: given an input length L, create an array a[0…L] of (L+1) spaces (we add the zero index to correspond to zero length for simplicity), which stores the optimum value for rods of length 0, 1, 2, 3….L. iterate through the array; for any rod of length ‘i’, the optimum way to cut it can be calculated from the previously calculated values, i.e. from a[0….i-1]. Concretely:  
   for j = 0 upto i/2:

possible\_max=max( possible\_max, a[j] + a[i-j] )  
Thus, our optimum solution for a[i], is the maximum possible\_max.   
You might be wondering why we only consider a pair of values, i.e. j and its complement with respect to i, when in fact, the best possible cut for a[i] might be many small little pieces. The answer is that that has been taken into account: both a[i-j-1] and a[j] are the optimums found when we have many small little pieces. So, in this solution, we are just combining two bags of small little pieces to make a[i].   
  
Eg: let our table be:

|  |  |
| --- | --- |
| Inches | $ |
| 1 | 3 |
| 2 | 9 |
| 3 | 4 |
| 5 | 11 |

Let input\_length=L=10. To solve this, we first have to find the optimals form 1 to 9.  
Start by mapping the table onto the array, upto at max L (because if the table has greater values, we can’t use them anyway).  
a=

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 3 | 9 | 4 | 0 | 11 | 0 | 0 | 0 | 0 | 0 |

* For i =0, we neither gain nor lose with a rod of 0’’, so we set it to zero.
* For i=1, i.e. 1’’, we have only one possible way to do it, so we don’t update the array.
* For i=2, i.e. 2’’ we consider the best possible\_max:  
  possible\_max=a[i]=a[2] = 9   
  for j=0 upto i/2:

possible\_max=max(possible\_max, a[j]+a[i-j])  
a[i]=possible\_max

i.e., we check if it is better to use 1’’ + 1’’ or 2’’ directly. Here, it is best to use 2’’ directly, and we get possible\_max of 9.   
a=

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 3 | 9 | 4 | 0 | 11 | 0 | 0 | 0 | 0 | 0 |

* For i=2, i.e. for 3’’,   
  possible\_max=a[i]=a[2] = 9   
  for j=0 upto i/2:

possible\_max=max(possible\_max, a[j]+a[i-j])  
a[i]=possible\_max

This process yields that it is better to use 1’’ and 2’’ rather than 3’’ directly, giving a best possible\_max of 12  
a=

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 3 | 9 | 12 | 0 | 11 | 0 | 0 | 0 | 0 | 0 |

* For i=4, i.e. for 4’’, we try:

possible\_max = 0’’ + 4’’ i.e. a[0]+a[4] = 0

possible\_max = 1’’ + 3’’ i.e. a[1]+a[3] = 12

possible\_max = 2’’ + 2’’ i.e. a[2]+a[2] = 18

Thus, we find that the best for i=4 is 2’’ + 2’’, with a total of 18.  
a=

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 3 | 9 | 12 | 18 | 11 | 0 | 0 | 0 | 0 | 0 |

**Note**: since the optimal of 3’’ is formed from 1’’ and 2’’, the 3’’ value we use to test for 4’’ is not a single rod piece, but a bag of rod pieces. So, we are considering all possibilities and using the previously-calculated solutions to the subproblems, i.e. dynamic programming.

* For i=5:  
  a=

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 3 | 9 | 12 | 18 | 11 | 0 | 0 | 0 | 0 | 0 |

* For i=6:  
  a=

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 3 | 9 | 12 | 18 | 11 | 0 | 0 | 0 | 0 | 0 |

* For i=7:  
  a=

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 3 | 9 | 12 | 18 | 11 | 0 | 0 | 0 | 0 | 0 |

* For i=8:  
  a=

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 3 | 9 | 12 | 18 | 11 | 0 | 0 | 0 | 0 | 0 |

* For i=9:  
  a=

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 3 | 9 | 12 | 18 | 11 | 0 | 0 | 0 | 0 | 0 |

* For i=10:  
  a=

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 3 | 9 | 12 | 18 | 11 | 0 | 0 | 0 | 0 | 0 |

Thus, using dynamic programming, we can solve the rod-cutting problem. This algorithm has O(L^2) time complexity, meaning it grows in terms of the size of the input number, as in the actual value. If our input has been L=1000 units, it would have taken a lot more time. Thus, our runtime is pseudo-polynomial. The space complexity is similarly O(L).

1. Edit Distance (or Levenshtein distance): find the minimum number of changes to be made to a string to convert it into another string.

**Alternate:** Given two strings of size M and N, convert one string into another. The set of operations you can perform are: insert and delete and replace, all having some cost, find minimum cost of edits (operations) required to convert one string into another.   
  
**Solution:**  
Levenshtein proposed this problem in 1966 with the operations insert, delete and substitute all having equal weight of 1. The most popular (and in fact, the best) solution was proposed by many people over the next decade, most notably by Wager and Fischer in 1974.  
We will discuss the Wagner-Fischer solution.  
  
First notice that the problem proposed by Levenshtein, i.e. “Find the minimum number of edits” is a subset of the problem “Find the minimum cost”, where the operations can all have different costs (in Levenshtein’s problem, cost of all operations must be equal). The Wagner-Fischer method easily allows for the costs of insertion, deletion and substitution to be made variable.   
Note that it is redundant to have substitution cost >= the sum of costs of insertion and deletion, as people would never use substitution if that was the case.  
  
A naïve, brute-force implementation to find the optimum is not very easy to implement, as we must trace all the paths possible, with all the possibilities of each path. It is an exponential algorithm that is not very easy to implement.  
  
  
**Wagner-Fischer solution:**  
Let the strings be A and B, of lengths N and M.  
Assume for now that cost of insertion = 1, deletion =1, and substitution = 1.5  
This solution uses an (N+1)x(M+1) matrix, E, with one extra row and column.   
Eg: When converting A=“SUNDAY” to B“SATURDAY”, N=6 thus we have 7 rows, and M=9 thus we have 10 columns.  
  
Every cell E[i][j] in this table holds the minimum edit distance for converting the substrings X[0…i] into Y[0…j].   
Eg: E[2,4] holds the Minimum Edit distance for converting “SU” into “SATU”.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **-** | **S** | **A** | **T** | **U** | **R** | **D** | **A** | **Y** |
|  | **i, j** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| **-** | **0** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **S** | **1** | 1 |  |  |  |  |  |  |  |  |
| **U** | **2** | 2 |  |  |  |  |  |  |  |  |
| **N** | **3** | 3 |  |  |  |  |  |  |  |  |
| **D** | **4** | 4 |  |  |  |  |  |  |  |  |
| **A** | **5** | 5 |  |  |  |  |  |  |  |  |
| **Y** | **6** | 6 |  |  |  |  |  |  |  |  |

Along the topmost row and left column of the matrix, we have filled the indices themselves. That is because, for those, one of the strings is “” (of length 0), so we must only perform insertions operations (or only deletion operations) in order to convert it into the other. Thus, we have our base cases.

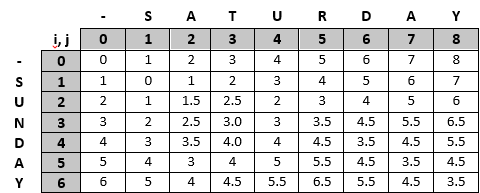
The other cells are calculated from the cells before them to the left, top, and top-left.  
There is a fixed rule when it comes to calculating the cell value E[i][j]:

* If A[i]==B[j], then E[i][j]=E[i-1][j-1].
* Else:
  + If coming from the **left**, we can just add B[j], so we add the **insertion cost.**
  + If coming from the **top**, we can just remove A[i], so we add the **deletion cost.**
  + If coming from the **top-left** cell i.e. E[i-1][j-1], then we always **substitute**.

Thus, in any scenario, we know what to do. We don’t know which one will give the minimum cost, so we calculate all of them and take the minimum.  
  
We can prove that the above operations are correct by taking an example:

To get E[2][5], i.e. “SU”->”SATUR”:

* From the left i.e. from the cost of going “SU”->”SATU”, we get the cost of getting to “SATUR”. This is the cost of getting from “SU”->”SATU”, plus the ‘insertion cost’ (for appending “R”).
* From the top: i.e. from the cost of going “S”->”SATU”, get the cost of getting from “SU” to “SATUR”. The answer is to delete U from “SU” to make it ”S”, then using the cost of “S”->”SATUR”. Thus, we add the ‘deletion cost’.
* From the top-left:   
  Now, if A[i]==B[j], there is no extra cost, eg: the cost of edits to be made to convert “SUND” to “SATURD” is the same as the cost of edits to convert “SUN” to “SATUR”, because there is no edit operation to be performed.   
  However, if A[i] != B[j], as in the current case while converting “SU” to “SATUR”, then we must add the substitution cost, because the cost is the cost of converting “S”->”SATU”, and then since the next alphabets are not the same, we must substitute A[i] with B[j], hence incurring the substitution cost.  
  i.e. “SU”->”SATUR” = “S”->”SATU” + sub(U,R), as we add the letter “U” (no charge) and then substitute it with “R”.



Let’s try filling the table for “SUNDAY”->”SATURDAY”, having already filled in the base cases E[0][0….M-1] and E[0….N-1][0]:

* E[1][1]: Cost to convert “S” into “S”. Since there is no change, the cost is 0.
* E[1][2]: Convert “S” into “SA”:
  + From left: insert “A”. Cost = 0 + 1 = 1
  + From top: Cost = cost( delete “S”) + cost of (“”->”SA”). Cost = 1 + 2 = 3.
  + From top-left: Cost (“”->”S”) + cost of replacement. Cost = 1 + 1.5 = 2.5.

Thus, the smallest cost is 1, which is what we put in E[1][2].

* E[4][3]: Convert “SUND”->”SAT”:
  + From left: 3.5 + 1 = 4.5
  + From top: 3.0 + 1 = 4.0
  + From top-left: 2.5 + 1.5 = 4.0
* E[5][7]: Convert “SUNDA” to “SATURDA”:
  + A[i]==B[j], thus = 3.5 + 0 = 3.5

We thus can get the entire table. The value we need to return is E[N-1][M-1].

Notice that for getting E[i][j], we assume that E[i-1][j-1], E[i][j-1] and E[i-1][j] have been found optimally. Hence, we only care about A[i] and B[j], i.e. the last alphabet of each of the words. We can be unconcerned with the rest.

A simple implementation follows:

int EditDistance(A, B, insert\_penalty=1, delete\_penalty=1, substitution\_penalty=1):  
 int \*\*E=new int [A.length + 1][B.length+1]; //no need to initialize  
 for (i=0; i < A.length+1; i++)  
 mat[i][0]= i;

for (j=0; j < B.length+1; j++)  
 mat[0][j]= j;

for (i=1; i < A.length+1; i++){  
 for (j=1; j < B.length+1; j++){  
 if(A[i-1]==B[j-1])

E[i][j]=mat[i-1][j-1];

else{  
 from\_left= E[i][j-1] + insert\_penalty; from\_top= E[i-1][j] + delete\_penalty; from\_top\_left= E[i-1][j-1] + substitution\_penalty ; E[i][j]=min(from\_left, min(from\_top, from\_top\_left);  
}

}

}

return E[A.length][B.length];

}

The recursive definition of the problem of Edit Distance is:

\begin{align}d_{i0} &= \sum_{k=1}^{i} w_\mathrm{del}(b_{k}), & & \quad  \text{for}\; 1 \leq i \leq m \\
d_{0j} &= \sum_{k=1}^{j} w_\mathrm{ins}(a_{k}), & & \quad \text{for}\; 1 \leq j \leq n \\
d_{ij} &= \begin{cases} d_{i-1, j-1} & \text{for}\; a_{j} = b_{i}\\ \min \begin{cases} d_{i-1, j} + w_\mathrm{del}(b_{i})\\ d_{i,j-1} + w_\mathrm{ins}(a_{j}) \\ d_{i-1,j-1} + w_\mathrm{sub}(a_{j}, b_{i}) \end{cases} & \text{for}\; a_{j} \neq b_{i}\end{cases} & & \quad  \text{for}\; 1 \leq i \leq m, 1 \leq j \leq n.\end{align}

With such a definition, we can use a recursive implementation to only calculate the final output, as we most likely don’t care about the others.  
  
Note: CLRS has a version of the Edit Distance problem with three additional operations:

* Copy, where you copy A[i] into B[j], i.e. B[j]=A[i], then set i++ and j++
* Twiddle, i.e. set B[j]=A[i+1] and B[j+1]=A[i], then set i+=2 and j+=2
* Kill, i.e. kill the remainder of the string A by not considering any more. The loop is broken.

The Edit Distance problem has many uses, and with different values/functions for the insertion, deletion and substitution cost, we can modify it to different applications. A common application is in autocorrect and fuzzy string searching.  
[Another explanation of Wagner-Fischer](https://jlordiales.wordpress.com/2014/03/01/dynamic-programming-edit-distance/).

1. AA