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. 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 for every element). 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 (Note: LIS= Longest\_increasing\_subsequence):  
   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. So, this naïve method will take at least O(2^N) time.  
     
   There is an algorithm which can find the answer in O(n) time and O(1) space. It is called **Kadane**’s algorithm.  
     
   Basically, notice this about the array: we can compress it logically into single positive numbers and negative numbers.   
   {-2, -3, 4, -1, -2, 1, 5, -3} => {-5, 4, -3, 6, -3}  
   now, we only add a group to our subarray if it increases a maximum. Adding negative numbers might increase the maximum only if the negative number region lies between two positive number regions which are both greater than it. In the above (compressed) example, -3 lies between 4 and 6; the longest subarray is {4, -3, 6} because the total sum is greater, because |-3|<|4| and |-3|<|6|, so there is an overall gain for both 4 and 6.  
   A very compact implementation of the algo is below. It works even for the case of an all-negative array:  
     
   int max(int x, int y) {   
    return (y > x)? y : x; }

int maxSubArraySum(int a[], int len){  
 int max\_so\_far = a[0];

int curr\_max = a[0];

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

curr\_max = max(a[i], curr\_max+a[i]); // check for {-2, -3, 4, -1, -2, 1, 5, -3}

max\_so\_far = max(max\_so\_far, curr\_max);

}

return max\_so\_far;

}