Lecture 6: Maximum Subarray Sum.

Lecturer: *Sundar Vishwanathan*
Computer Science & Engineering Indian Institute of Technology, Bombay

As input for the next problem, you are given an array A. We wish to find $i \leq j$ such that $A[i] + \cdots + A[j]$ is maximum.

You may have seen this algorithm already. Our objective is to look at the way these algorithms are designed. The bruteforce algorithm computes these sums for each i, j and outputs the maximum. A bad implementation will run in $O(n^3)$ time. First find an $O(n^2)$ algorithm.

Method 1. Induction. We begin the design with the simplest form of recursion. Recurse on the first $n-1$ elements and then augment the solution when the last element is considered. The only subarrays the last element is involved in are of the form $A[t]+A[t+1]+\cdots A[n-1]+A[n]$, for each t. These (suffix) sums can be found in $O(n)$ time. How? The recurrence is $T(n) = T(n-1)+n$. This yields an $O(n^2)$ time solution. Often, you will find yourself in this situation. In the sense that you try something new but end up with an obvious bound.

Now begins the thinking part. The extra information needed are the suffix sums for the last element, and in particular the maximum of these suffix sums. One way to improve the situation is to observe that we are recalculating an old value. Which one? Notice that while the principle by itself is just common sense, the difficulty lies in identifying when it is applicable. *Hint. Compare the suffix sums computed above during iteration $i-1$ and iteration i.*

Here are the two lists of suffix sums. For the $i-1$th iteration we need to compute the maximum of $A[1]+A[2]+\cdots A[i-1]; A[2]+A[3]+\cdots A[i-1]; \cdots ; A[i-2]+A[i-1]; A[i-1]$.

For the ith iteration we need to compute the maximum of $A[1]+A[2]+\cdots A[i-1]+A[i]; A[2]+A[3]+\cdots A[i-1]+A[i]; \cdots ; A[i-1]+A[i]; A[i]$.

Apart from the term $A[i]$ the others in the latter list are derived from the previous set by adding $A[i]$ to each. If we had only stored the result of the maximum of the prefix sums computed in the previous (ith) step, computing the maximum of the new ($i+1$th) prefix sums takes $O(1)$ time. The new recurrence is $T(n) = T(n-1)+c$ where c is a constant. This yields an $O(n)$ algorithm.

Here is another way to look upon this. The key question you ask is this: Can the recursive call return some more information which you can use? You progress when you notice that if you had the maximum of the suffix sums of the first $n-1$ elements of the array, in addition to the maximum subarray sum from the recursive call then finding the new maximum subarray sum takes just one extra addition and one comparison, yielding an $O(n)$ solution. This is our next design principle.

Design Principle 4. Soup up the inductive hypothesis if it is easier/faster to solve this more general problem.

By souping up the inductive hypothesis it usually means we ouput more than what is asked for. Sometimes the extra information will occur to you as you do the recursion. In our case we will design an algorithm that not only returns the maximum subarray sum, but also the maximum suffix sum. In the inductive step then, we have to compute *both* of these quantities, given these two quantities for the smaller array. It is not difficult to see that that both can be done in $O(1)$ additional time, finishing an $O(n)$ algorithm for the problem.

There are other ways of viewing this solution which yields other techniques. Let $M[i]$ denote the maximum sum amongst all intervals ending at i. That is it is the maximum suffix sum ending at i. If we could compute $M[i]$ for i from 1 to n then solving our original problem is easy and requires just one pass of M. Why? Now we notice that computing these values by induction is easy. In a way this is what we did in the previous attempt albeit implicitly. Here we do so explicitly. The main recursive step then is the following: if you had all M values upto $n - 1$ how do you compute $M[n]$? I will leave it to you to recall that this can be done in $O(1)$ time, again yielding an $O(n)$ solution.

Exercise: Write the recursive procedure for calculating M.

Design Principle 4a. If you wish to calculate certain quantity for an array which can be seen as the maximum or minimum of certain values calculated for every index, it may be easier to find this value for each index first.