CS 218 : Design and Analysis of Algorithms

Lecture 6: Maximum Subarray Sum. Counting Inversion Pairs

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. Assume that not all entries in A are negative.

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. Recursion. 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 elememnt is considered. The only subarrays the last element is involved in are of the form $A[t]+A[t+1]+\cdots A[n-1]+A[n]$. These (suffix) sums can be found in $O(n)$ time. Why? 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 thing to notice (though it is of no use to us) is that if the answer is known, we may not need to look up all the suffix sums. Why? However, in the worst case, this observation will not help us.

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 and iteration $i + 1$.*

Here are the two. For the $i-1$th iteration we 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 compute $A[1] + A[2] + \cdots A[i-1] + A[i], A[2] + A[3] + \cdots A[i-1] + A[i], \cdots, A[i-1] + A[i], A[i]$.

Apart from the sequence $A[i]$ the other sequences are derived from the previous set of sequences by adding $A[i]$ to each. In particular the maximum amongst them is known. If we had only stored the result of the maximum of the prefix sums computed in the previous step, computing the maximum of the new prefix sums takes $O(1)$ time. This yields a $O(n)$ 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. Usually you come up with this 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 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 minimum sum amongst all intervals ending at i. We could compute $M[i]$ for i from 1 to n in that order. Notice that computing all these values is not difficult. The induction is easy. We return the maximum of the M values. Notice that 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.

Design Principle 4a. If you wish to calculate certain quantity for an array, it may be easier to find this value for each prefix.

The second approach was to compute, first, $D(i) = A[1] + \cdots + A[i]$. Given the D values, can you see how to compute $M(i)$ for each i?

$$M[i] = D[i] - \min_{j \leq i} D[j]$$

The second term in the above difference can be computed in $O(n)$ time in one pass of the array; something you have seen before. So, the complete algorithm has four passes. Compute the D values in one pass. Compute the prefix minimums of the D values in the next. Compute the S values in the third and the answer in the fourth. Of course, you can do many passes in one pass.

This solution gives us the fifth design principle.

Design Principle 5. Reduction. Show that it is enough to solve a simpler problem. In other words *reduce* solving the first problem to solving a problem which you know how to solve, or which is easier to solve.

Here it was computing S to computing D.

The next problem is as following:

Input: Array A.

Output: Number of pairs $i < j$ such that $A[i] > A[j]$

The naive method makes $O(n^2)$ comparisons. We wish to do this faster. The first step is to try the simplest of inductions. Recursively find this value for the first $n-1$ elements in the array and then extend to the entire array. For the last element, we need to know the number of elements which are smaller than it. The naive way of comparing it with each element fails to give us improvements. This should look familiar by now.

However, notice that if we also had the prefix of the array sorted, we could now find this in $O(\log n)$ extra comparisons. We cannot afford to sort it at this juncture since that will be too expensive.

Exercise. Write the recurrence and solve it.

One idea is to build the sorting into the recursive call. That is, instead of just solving the problem, the procedure also sorts the array and returns it. While this takes care of the number, and also keeps the number of comparisons down to $O(n \log n)$, the sorting is insertion sort leading to $O(n^2)$ time.

There are two branches one can take at this point. Like insertion sort, we look at a data structures solultion or change the recursion.

We now enter a stage where the design is problem dependent. How did we solve this problem for sorting? There were two main techniques. One is to break the problem into two halves and recurse; that way insertion sort gives way to merge sort.

Before we proceed further recall **Principle 5** During Recursive calls if it is easier to solve a more general problem, then try and write a recursion to solve the more general problem.

Let us apply the divide and conquer paradigm blindly. Divide the array into two equal parts, solve each part recursively and put the solutions together. How do we put the solutions together? Note that putting together is an algorithmic problem. State this clearly first. Here is the problem: Given two arrays X and Y, determine the number of pairs $x, y; x \in X, y \in Y$ such that $x \leq y$. Again, it looks hopeless if you compare each element in X to each element in Y. However, if Y were sorted, one could find the required quantity for each element in X using binary search. At this point we can afford to sort Y and still get something non-trivial.

If the time taken were $T(n)$ the recurrence we get is:

$$T(n) \leq 2T(n/2) + cn \log n$$

For some constant c. Where the $n \log n$ time is to do binary search for each element in X and sorting Y. The running time comes out to be $O(n(\log n)^2)$. Note that this is already much better than the brute force method.

The next step is a problem dependent observation that if the two parts were already sorted then one could find the number of pairs in $O(n)$ time. How?

Hint. Do this while merging. When you insert an element from X into the sorted array, can you find the number of inversion pairs this element has with elements of Y in $O(1)$ time?

Design Tip: If there are many steps in the algorithm which take time, work with each and see if each of them can be improved.

But what about the sorting? If you unroll the recursion and check, you will notice the unnecessary sorting at each step and should be able to figure out the fix! The solution is to use the principle we mentioned above which is to soup up the induction. In other words, we solve a more general problem. Not only does the procedure InvPairs count the number of inversion pairs, it also recursively returns the array *sorted.* We can find the number of inversions in an extra $O(n)$ time. To return the sorted array, we only need to merge the two sorted arrays an additional $O(n)$ time.

The running time now follows the following familiar recursion with a satisfactory result.

$$T(n) = 2T(n/2) + cn$$

Exercise. Write the entire procedure formally. Work out a few examples and see that you understand how it all works.

Note. When you soup up the induction, make sure that your procedure outputs everything needed at the next level.

Let us go back to the original problem. Here is the other path. Recall that you have recursively solved the problem on the first $n - 1$ elements of the array and would like to extend the solution to the nth element. It is sufficient to have a data structure that supports the following operations in $O(\log n)$ time.

- Insert(x).
- FindinverseRank(x): which returns the number of elements greater than an element x in the data structure.

Again, the key is to recognize what you require. If we had such a data structure, we could scan the array from left to right, insert each element and use findinverserank each time to update a count. If we could do each of these operations in $O(\log n)$ time we would have a $O(n \log n)$ algorithm. *Exercise.* Modify the balanced binary search tree data structure to implement the above. The big question is what extra information do you store so that findrank can be computed in $O(\log n)$ time. By the way, thinking balanced binary search tree is natural-this was the other way to fix insertion sort. *Hint.* What you store is not the rank. This is yet another example of reduction. We have reduced the problem to a data structure design problem.

First a design principle for problems on arrays which we have used repeatedly.
I will not number it since it is covered by the others. **Principle. Array scan.** Scan the array from left to right, use (or design one if needed) an appropriate data structure and update/compute information.

Finally the meta tip for the day. Be overconfident when you begin the design. Once you think you have the algorithm in place be very critical and check if the solution is correct.