CS 218 : Design and Analysis of Algorithms

Lecture 11: Dynamic Programming: Order and Method. Weighted Interval Scheduling.

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

1 Weighted Interval Scheduling

Consider the following problem. **Input:** A set S of n intervals given by their left and right end-points and a positive integral weight for each interval.
Output: A subset T of S of maximum weight such that no two intervals in T overlap.

A typical problem that we will encounter many times is to find a subset of a set of maximum or minimum weight so that some constraints are satisfied.

Step 1: Find a recursive solution.

Either a subset contains the first element or it does not. We will consider both cases separately and put them together. Suppose the original set of intervals is S. Let $MWNOSI(S)$ denote a procedure which returns a maximum weight non overlapping subset of intervals. The procedure then returns the maximum of

$$MWNOSI(S \setminus \{I\})$$

and

$$MWNOSI(S') + W_I$$

Here I is any interval in S, S' is the set of all intervals that do not overlap with I. Note that correctness can be easily proved by induction. Note also the brute force nature of the algorithm we have just designed. We do not know whether the first interval is chosen or not; we try both and choose the best.

There are more than one recursive calls (here two). Also, the input to the two calls have overlap. If we unroll the recursion, we see that we may repeat calls.

Step 2 is to determine what the distinct calls to MWNOSI are. Distinct in terms of distinct values of the input. Note that any subset of the intervals is possible depending on the input and the way we choose I at each stage. So the best we can say about the number of recursive calls is 2^n. This running time is not acceptable.

Step 3. If the number of distinct calls is large, try ordering the input. In fact, we recommend ordering anyway. One may just order the input arbitrarily or use some parameters from the problem. By ordering them arbitrarily we mean considering the input as $I_1, \ldots, I_n$, say the order in which the intervals are presented. There are other orders possible, for instance order by weight, by the left end-points, etc.

We begin with the lexicographic ordering.

$$MWNOSI(I_1, \ldots, I_n)$$

. In the recursive step we return the maximum of:

$$MWNOSI(I_2, \ldots, I_n)$$

and

$$MWNOSI(I_p, \ldots, I_q) + MWNOSI(I_r, \ldots, I_s) + W_1.$$

We need to explain what p, q, r, s are. Suppose the first interval is (a, b). Then $I_p, \ldots, I_q$ are all intervals which end before a in the same order in which they appear in $I_1, \ldots, I_n$. Similarly the latter term are intervals which start after b. *Exercise.* Prove that the number of distinct recursive calls here is a polynomial in n.

This is already an improvement. We emphasize that if we had not split the second call into two disjoint parts, and worked on just the intervals which did not intersect with I_i then we would not have a polynomial number of distinct calls. So the trick here is that if the input splits into disjoint pieces such that the solution to one does not affect the solution to others, and the solutions to the pieces can be put together to get the final solution then use different recursive calls for each piece. In general, for any problem, we recommend starting with any ordering. Lexicographic hi sahi.

Now note that if I_1 were the interval that started first then we have only one term in the second case. It also simplifies how we handle distinct recursive calls. So, we order the intervals by their left end-points. This is what most books do, and yields the fastest procedure for the problem. In this case, we see that the recursive calls are of the form given below and we pick the maximum solution amongst them.

$$MWNOSI(I_2, \ldots, I_n)$$

and

$$MWNOSI(I_j, I_{j+1}, \ldots, I_n) + W_1.$$

This is because the first interval intersects with the first few intervals, say the first $j-1$. Here is the key fact. Each call is a prefix of the input order. This yields n possible distinct calls. This is something we will aspire for for now: see if we can keep recursive calls to prefixes or suffixes.

For completeness we need to define a generic procedure call. $MWNOSI(I_j, I_{j+1}, \ldots, I_n)$. Write a recurrence for this. For the final output we will call this procedure with $j = 1$. Also clearly define the base cases.

Now define a table $T(j)$, that will store $MWNOSI(I_j, I_{j+1}, \ldots, I_n)$.

In the procedure, we will first initialize the table. Then when invoked, we first check if the corresponding table entry is filled. If yes, we return it. If not, we compute this and then fill the table entry.

How much time does this procedure take? The key is to charge operations (additions, comparisons) to table entries.

2 Adding to the Key Steps from last time.

1. Give the procedure a name. Clearly describe the input and output.
2. Write a recursive procedure.
3. Determine the number of distinct calls.
4. If the number of distinct calls is large, try ordering the input. Find an order such that the number of distinct calls is as small as possible. Split the input into distinct calls during the recursion if possible.
5. Allocate a table, with one entry per distinct call.

We will add one more step to this list and that about takes care of this design paradigm. Note that the key steps are evaluation of the number of distinct calls and finding a suitable order.