Chapter 6 Dynamic Programming

We began our study of algorithmic techniques with greedy algorithms, which in some sense form the most natural approach to algorithm design. Faced with a new computational problem, we've seen that it's not hard to propose multiple possible greedy algorithms; the challenge is then to determine whether any of these algorithms provides a correct solution to the problem in all cases.

6.1 Weighted Interval Scheduling: A Recursive Procedure

We have seen that a particular greedy algorithm produces an optimal solution to the Interval Scheduling Problem, where the goal is to accept as large a set of nonoverlapping intervals as possible. The weighted Interval Scheduling Problem is a strictly more general version, in which each interval has a certain value (or weight), and we want to accept a set of maximum value.

Designing a Recursive Algorithm

Since the original Interval Scheduling Problem is simply the special case in which all values are equal to 1, we know already that most greedy algorithms will not solve this problem optimally. But even the algorithm that worked before (repeatedly choosing the interval that ends earliest) is no longer optimal in this more general setting.

Indeed, no natural greedy algorithm is known for this problem, which is what motivates our switch to dynamic programming. As discussed above, we will begin our introduction to dynamic programming with a recursive type of algorithm for this problem, and then in the next section we'll move to a more iterative method that is closer to the style we use in the rest of this chapter.

We use the notation from our discussion of Interval Scheduling. We have n requests labeled $1, ..., n$, with each request i specifying a start time s_i and a finish time f_i. Each interval i now also has a value, or weight v_i. Two intervals are compatible if they do not overlap. The goal of our current problem is to select a subset $S \subseteq \{1, ..., n\}$ of mutually compatible intervals, so as to maximize the sum of the values of the selected intervals,

$$\sum_{i \in S} v_i$$

Let's suppose that the requests are sorted in order of nondecreasing finish time:

$f_1 \leq f_2 \leq ... \leq f_n$. We'll say a request i comes before a request j if $i < j$. This will be the natural left-to-right order in which we'll consider intervals. To help in talking about this order, we define $p(j)$, for an interval j, to be the largest index $i < j$ such that intervals i and j are disjoint. In other words, i is the leftmost interval that ends before j begins. We define $p(j) = 0$ if no request $i < j$ is disjoint from j.

Now, given an instance of the Weighted Interval Scheduling Problem, let's consider an optimal solution $\mathcal{O}$, ignoring for now that we have no idea what it is. Here's something completely obvious that we can say about $\mathcal{O}$: either interval n (the last one) belongs to $\mathcal{O}$, or it doesn't. Suppose we explore both sides of this dichotomy a little further. If $n \in \mathcal{O}$, then clearly no interval indexed strictly between $p(n)$ and n can belong to $\mathcal{O}$, because by the definition of $p(n)$, we know that intervals $p(n) + 1, p(n) + 2, ..., n - 1$ all overlap interval n. Moreover, if $n \in \mathcal{O}$, then $\mathcal{O}$ must include an optimal solution to the problem consisting of requests $\{1, ..., p(n)\}$ - for if it didn't, we could replace $\mathcal{O}$'s choice of requests from $\{1, ..., p(n)\}$ with a better one, with no danger of overlapping request n.

On the other hand, if $n \notin \mathcal{O}$, then $\mathcal{O}$ is simply equal to the optimal solution to the problem consisting of requests $\{1, ..., n - 1\}$. This is by completely analogous reasoning: we're assuming that $\mathcal{O}$ does not include request n; so if it does not choose the optimal set of requests from $\{1, ..., n - 1\}$, we could replace it with a better one.

All this suggests that finding the optimal solution on intervals $\{1, 2, ..., n\}$ involves looking at the optimal solutions of smaller problems of the form $\{1, 2, ..., j\}$. Thus, for any value of j between 1 and n, let $\mathcal{O}_j$ denote the optimal solution to the problem consisting of requests $\{1, ..., j\}$, and let $OPT(j)$ denote the value of this solution. (We define $OPT(0) = 0$, based on the convention that this is the optimum over an empty set of intervals.) The optimal solution we're seeking is precisely $\mathcal{O}_n$, with value $OPT(n)$. For the optimal solution $\mathcal{O}_j$ on $\{1, 2, ..., j\}$, our reasoning above (generalizing from the case in which $j = n$) says that either $j \in \mathcal{O}_j$, in which case $OPT(j) = v_j + OPT(p(j))$, or $j \notin \mathcal{O}_j$, in which case $OPT(j) = OPT(j - 1)$. Since these are precisely the two possible choices ($j \in \mathcal{O}_j$ or $j \notin \mathcal{O}_j$), we can further say that.

$$OPT(j) = max(v_j + OPT(p(j)), OPT(j - 1)) \quad (1)$$

And how do we decide whether n belongs t the optimal solution $\mathcal{O}_j^2$. This too is easy: it belongs to the optimal solution if and only if the first of the options above is at least as

good as the second; in other words,

Request j belongs to an optimal solution on the set $\{1, 2, ..., j\}$ if and only if

$$v_j + OPT(p(j)) \geq OPT(j-1) \quad (2)$$

These facts form the first crucial component on which a dynamic programming solution is based: a recurrence equation that expresses the optimal solution (or its value) in terms of the optimal solutions to smaller subproblems.

Despite the simple reasoning that led to this point, (1) is already a significant development. It directly gives us a recursive algorithm to compute $OPT(n)$, assuming that we have already sorted the requests by finishing time and computed the values of $p(j)$ for each j.

$Compute - Opt(j)$
If $j = 0$ then
  Return 0
Else
  Return $max(v_j + Compute - Opt(p(j)), Compute - Opt(j-1))$
Endif

The correctness of the algorithm follows directly by induction on j:

$Compute - Opt(j)$ correctly computes $OPT(j)$ for each $j = 1, 2, ..., n$.

Proof. By definition $OPT(0) = 0$. Now, take some $j > 0$, and suppose by way of induction that $Compute - Opt(j)$ correctly computes $OPT(i)$ for all $i < j$. By the induction hypothesis, we know that $Compute - Opt(p(j)) = OPT(p(j))$ and $Compute - Opt(j-1) = OPT(j-1)$; and hence from (1) it follows that

$$\begin{aligned} OPT(j) &= max(v_j + Compute - Opt(p(j)), Compute - Opt(j-1)) \\ &= Compute - Opt(j). \end{aligned} \quad (1)$$

Unfortunately, if we really implemented the algorithm $Compute - Opt$ as just written, it would take exponential time to run in the worst case.

Memoizing the Recursion

In fact, though, we're not so far from having a polynomial-time algorithm. A fundamental observation, which forms the second crucial component of a dynamic programming solution, is that our recursive algorithm $Compute - Opt$ is really only solving $n + 1$ different subproblems: $Compute - Opt(0), Compute - Opt(1), ..., Compute - Opt(n)$. The fact that it runs in exponential time as written is simply due to the spectacular redundancy in the number of times it issues each of these calls.

How could we eliminate all this redundancy? We could store the value of $Compute - Opt$ in a globally accessible place the first time we compute it and then simply use this precomputed value in place of all future recursive calls. This technique of saving values that have already been computed is referred to as *memoization*.

We implement the above strategy in the more "intelligent" procedure $M - Compute - Opt$. This procedure will make use of an array $M[0...n]$; $M[j]$ will start with the value "empty", but will hold the value of $Compute - Opt(j)$ as soon as it is first determined. To determine $OPT(n)$, we invoke $M - Compute - Opt(n)$.

$M - Compute - Opt(j)$
  If $j = 0$ then
    Return 0
  Else if $M[j]$ is not empty then
    Return $M[j]$
  Else
    Define $M[j] = max(v_j + M - Compute - Opt(p(j)), M - Compute - Opt(j-1))$
    Return $M[j]$
  Endif

Analyzing the Memoized Version

Clearly, this looks very similar to our previous implementation of the algorithm; however, memoization has brought the running time way down.

The running time of $M - Compute - Opt(n)$ is $O(n)$ (assuming the input intervals are sorted by their finish times).