CMSC 451: Lecture 8
Dynamic Programming: Weighted Interval Scheduling

Dynamic Programming: In this lecture we begin our coverage of an important algorithm design technique, called *dynamic programming* (or *DP* for short). The technique is among the most powerful for designing algorithms for optimization problems. Dynamic programming is a powerful technique for solving optimization problems that have certain well-defined clean structural properties. (The meaning of this will become clearer once we have seen a few examples.)

There is a superficial resemblance to divide-and-conquer, in the sense that it breaks problems down into smaller subproblems, which can be solved recursively. However, unlike divide-and-conquer problems, in which the subproblems are disjoint, in dynamic programming the subproblems typically overlap each other, and this renders straightforward recursive solutions inefficient.

Dynamic programming solutions rely on two important structural qualities, *optimal substructure* and *overlapping subproblems*.

Optimal substructure: This property (sometimes called the *principle of optimality*) states that for the global problem to be solved optimally, each subproblem should be solved optimally. While this might seem intuitively obvious, not all optimization problems satisfy this property. For example, it may be advantageous to solve one subproblem sub-optimally in order to conserve resources so that another, more critical, subproblem can be solved optimally.

Overlapping Subproblems: While it may be possible subdivide a problem into subproblems in exponentially many different ways, these subproblems overlap each other in such a way that the number of distinct subproblems is reasonably small, ideally *polynomial* in the input size.

An important issue is how to generate the solutions to these subproblems. There are two complementary (but essentially equivalent) ways of viewing how a solution is constructed:

Top-Down: A solution to a DP problem is expressed recursively. This approach applies recursion directly to solve the problem. However, due to the overlapping nature of the subproblems, the same recursive call is often made many times. An approach, called *memoization*, records the results of recursive calls, so that subsequent calls to a previously solved subproblem are handled by table look-up.

Bottom-up: Although the problem is formulated recursively, the solution is built iteratively by combining the solutions to small subproblems to obtain the solution to larger subproblems. The results are stored in a table.

In the next few lectures, we will consider a number of examples, which will help make these concepts more concrete.

Weighted Interval Scheduling: Let us consider a variant of a problem that we have seen before, the Interval Scheduling Problem. Recall that in the original (unweighted) version we are given a set $R = \{r_1, \ldots, r_n\}$ of n *requests* to be scheduled on an exclusive resource (e.g., a picnic

table at a local park). Each request has a start-finish time interval, $[s_i, f_i]$. The objective is to compute any maximum sized subset of non-overlapping intervals (see Fig. 1(a)).

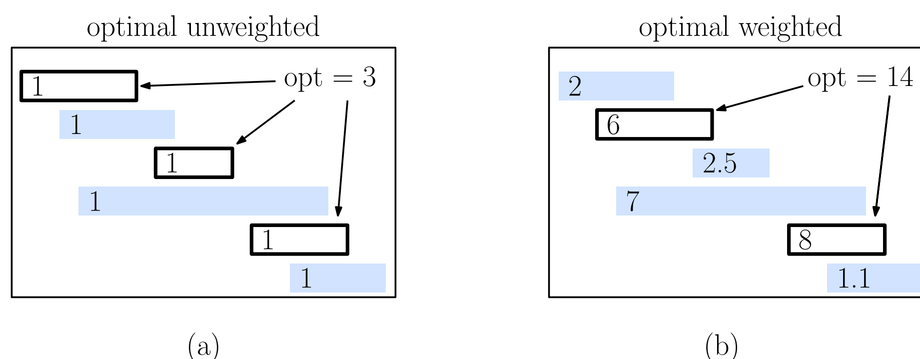

Fig. 1: Weighted and unweighted interval scheduling.

In *weighted interval scheduling*, we assume that in addition to the start and finish times, each request is associated with a numeric *weight* or *value*, call it v_i, and the objective is to find a set of non-overlapping requests such that sum of values of the scheduled requests is maximum (see Fig. 1(b)). The unweighted version can be thought of as a special case in which all weights are equal to 1. Although a greedy approach works fine for the unweighted problem, no greedy solution is known for the weighted version. We will demonstrate a method based on dynamic programming.

Recursive Formulation: Dynamic programming solutions are based on a decomposition of a problem into smaller subproblems. Let us consider how to do this for the weighted interval scheduling problem. As we did in the greedy algorithm, it will be convenient to sort the requests in non-decreasing order of finish time, so that $f_1 \leq \ldots \leq f_n$.

Here is the idea behind DP in a nutshell. Consider the *last* request $[s_n, f_n]$. There are two possibilities. If this request *is not* in the optimum schedule, then we can safely ignore it, and recursively compute the optimum solution of the first $n-1$ requests. Otherwise, this request *is* in the optimal solution. We will schedule this request (receiving the profit of v_n) and then we must eliminate all the requests whose intervals overlap this one. Because requests have been sorted by finish time, this involves finding the largest index p such that $f_p < s_n$. Thus, we solve the problem recursively on the first p requests.

But we don't know the optimum solution, so how can we select among these two options? The answer is that we will compute the cost of both of them recursively, and take the better of the two.

Let's now implement this idea. Recall that the requests are sorted by finish times. For the sake of generality, let's assume that we want to solve the problem on requests 1 through j, where $0 \leq j \leq n$. If $j = 0$, there is nothing to do. Otherwise, given any request j, define $\text{prior}(j)$ to be the largest integer such that $f_{\text{prior}(j)} < s_j$, that is, $\text{prior}(j)$ is latest request by finish times that does *not* overlap request j. If no such i exists (that is, all the preceding intervals overlap), let $\text{prior}(j) = 0$ (see Fig. 2). This means that if request j is put in the schedule, we are free to include request $\text{prior}(j)$ and an earlier ones.

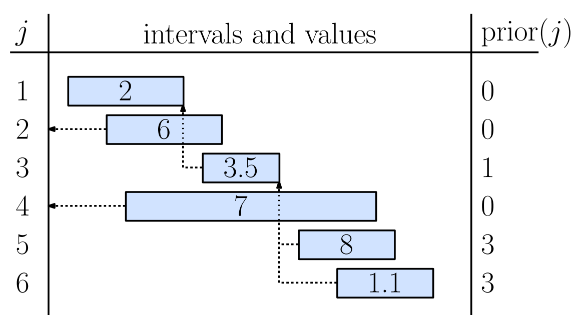

Fig. 2: Weighted interval scheduling input and prior-values.

For now, let's just concentrate on computing the *optimum total value*. Later we will consider how to determine which *requests* produce this value. A natural idea would be to define a function that gives the optimum value for just the first i requests.

Definition: For $0 \leq j \leq n$, $W(j)$ denotes the maximum possible value achievable if we consider just requests $\{1, \ldots, j\}$ (assuming that requests are given in order of finish time).

As a basis case, observe that if we have no requests, then we have no value. Therefore, $W(0) = 0$. If we can compute $W(j)$ for each value of j, then clearly, the final desired result will be the maximum value using *all* the requests, that is, $W(n)$.

Summarizing our earlier observations, in order to compute $W(j)$ for an arbitrary j, we observe that there are two possibilities:

Request j is not in the optimal schedule: If j is not included in the schedule, then we should do the best we can with the remaining $j-1$ requests. Therefore, $W(j) = W(j-1)$.

Request j is in the optimal schedule: If we add request j to the schedule, then we gain v_j units of value, but we are now limited as to which other requests we can take. We cannot take any of the requests following prior(j). Thus we have $W(j) = v_j + W(\text{prior}(j))$.

Ignoring the basis case ($j = 0$) there are two options. Clearly, we take the better of the two.

$$W(j) \;=\; \begin{cases} 0 & \text{if } j = 0 \text{ (basis)} \\ \max \left\{ \begin{array}{ll} W(j-1) & \text{(reject } j\text{)} \\ v_j + W(\text{prior}(j)) & \text{(accept } j\text{)} \end{array} \right\} & \text{if } j > 0 \end{cases}$$

The optimal solution for all the requests is $W(n)$. Note that the principle of optimality applies here. When we make the recursive invocations to $W(j-1)$ and $W(\text{prior}(j))$, we should solve these problems optimally. (Their solutions are independent of the choices we have made that brought us here, so there is no harm in doing the best we can in solving them.)

How do we know which of the these two options (accept or reject) to select? We will see in the next section that evaluating both of them in a straightforward manner will lead to a high running time. While it is tempting to try to determine which option is better (e.g., higher value or shorter duration), the basic law of dynamic programming is to *act stupid*—just try both and take the better option.

Recursive Implementation (Slow!): The simplest implementation would be to express the recursive formulation as a recursive function. See the following code block. We first sort the requests by finish time and precompute the values of prior(j), which we store in an array. We then invoke the recursive function `rec-WIS(n)` to compute the best total cost.

Recursive Weighted Interval Scheduling (Inefficient!)

```
WIS(s[1..n], f[1..n], v[1..n]) {          // recursive WIS (slow)
    Sort requests by finish time
    Compute prior[j] for j = 1, ..., n (Exercise)
    return rec-WIS(n)                      // total value of all requests
}

rec-WIS(j) {                               // total value of 1 .. j
    if (j == 0) return 0                   // basis
    else return max(
        rec-WIS(j-1),                      // reject j
        v[j] + rec-WIS(prior[j]) )         // accept j
}
```

The correctness of this procedure follows from the previous discussion. The running time will be a problem, however. To make this concrete, let us suppose that prior$(j) = j - 2$, for all j. (Convince yourself that you can construct a set of intervals to make this happen.) Let $T(j)$ be the number of recursive function calls to `rec-WIS`(0) that result from a single call to `rec-WIS`(j). Clearly, $T(0) = 1$, $T(1) = T(0) + T(0)$, and for $j \geq 2$, we have $T(j) = T(j-1) + T(j-2)$ (see Fig 3). If you start expanding this recurrence, you will find that the resulting series is essentially a Fibonacci series:

j	0	1	2	3	4	5	6	7	8	...	20	30	50
$T(j)$	1	2	3	5	8	13	21	34	55	...	17,711	2,178,309	32,951,280,099

It is well known that the Fibonacci series $F(j)$ grows exponentially as a function of j, in particular $F(n) \approx \phi^n$, where $\phi \approx 1.618$. Our running time is at least $T(n)$, which is exponential in n.

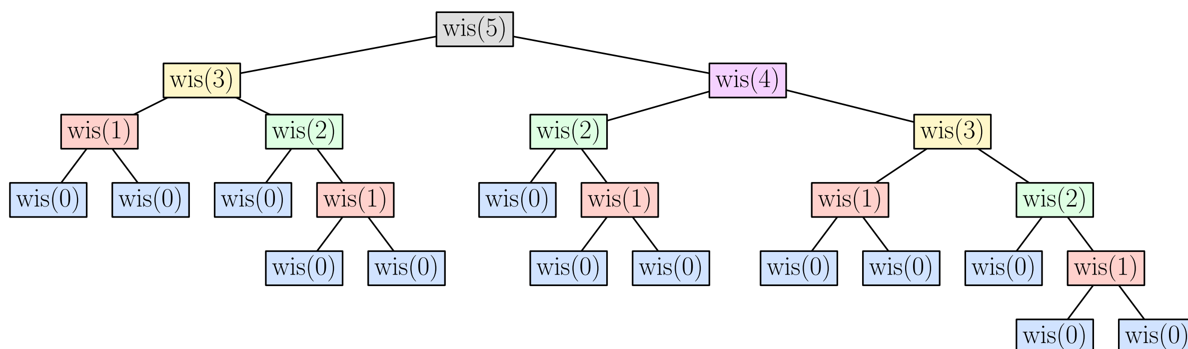

Fig. 3: The exponential nature of recursive-WIS.

Memoized Formulation (and Extracting the Schedule): The problem with the simple recursive formulation is that it repeatedly calls the same function. To save time, let's just cache (or save) the value, and look it up later if needed. This process is called *memoization*. (Intuitively, we are making a "memo" to ourselves of what the value is for future reference.) The algorithm below saves the values of $W(j)$ in an array $W[j]$. This results in significantly fewer recursive calls (see Fig. 4), with only one recursive call for each j.

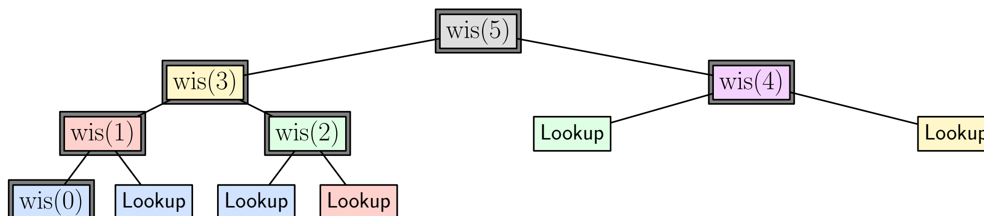

Fig. 4: Recursive calls in memoized WIS. (Highlighted squares are recursive calls, and the others are table lookups.)

Initially, we set $W[j] = -1$ for $0 \leq j \leq n$, so we know that its value has not been initialized. The modified algorithm is presented in the code-block below. We will add one additional piece of information, which will help in reconstructing the final schedule. Whenever a choice is made between two options, we'll save a flag that indicates whether we accepted or rejected the current request. In particular, we maintain a boolean array accept, and we set accept$[j]$ to true if j was accepted and otherwise we set it to false. The resulting algorithm is presented in the following code block. As before, assume that requests have been sorted by finish time and the array prior$[1..n]$ has been computed.

Memoized Weighted Interval Scheduling

```
memo-WIS(j) {                                    // memoized WIS implementation
    if (j == 0) return 0                         // basis case - no requests
    else if (W[j] has been computed) return W[j]
    else {
        rejectVal = memo-WIS(j-1)                // value if we reject j
        acceptVal = v[j] + memo-WIS(prior[j])    // value if we accept j
        if ( rejectVal > acceptVal ) {           // better to reject
            W[j] = rejectVal
            accept[j] = true                     // remember our choice
        } else {                                 // better to accept
            W[j] = acceptVal
            accept[j] = false                    // remember our choice
        }
        return W[j]                              // final value
    }
}
```

An example is shown in Fig. 5. The memoized version runs in $O(n)$ time. To see this, observe that each invocation of memo-WIS either returns in $O(1)$ time (with no recursive calls), or it computes one new entry of W. Since there are n entries in the table, the latter can occur at

most n times.

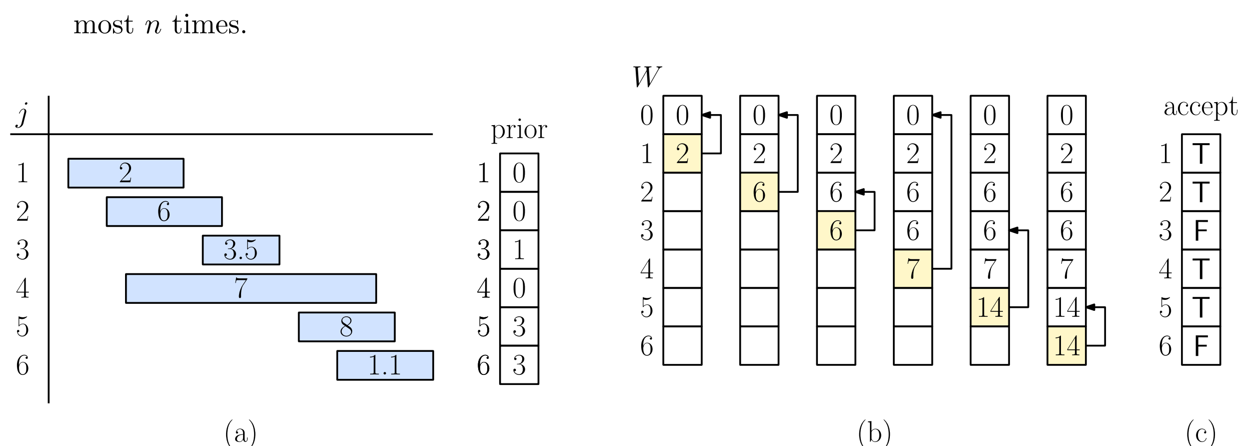

Fig. 5: (a) The input intervals and prior values, (b) the bottom-up construction of the table and accept flags, (c) the final accept values.

Bottom-up Construction: (Optional) Yet another method for computing the values of the array, is to dispense with the recursion altogether, and simply fill the table up, one entry at a time. We need to be careful that this is done in such an order that each time we access the array, the entry being accessed is already defined. This is easy here, because we can just compute values in increasing order of j. As before, we include the computation of the accept flags.

Bottom-Up Weighted Interval Scheduling

```
bottom-up-WIS() {                                    // bottom-up WIS implementation
    W[0] = 0                                         // basis
    for (i = 1 to n) {
        rejectVal = W[j-1]                           // value if we reject j
        acceptVal = v[j] + W[prior[j]]               // value if we accept j
        if ( rejectVal > acceptVal ) {               // better to reject
            W[j] = rejectVal
            accept[j] = true                         // remember our choice
        }
        else {                                       // better to accept
            W[j] = acceptVal
            accept[j] = false                        // remember our choice
        }
    }
    return W[n]
}
```

Do you think that you understand the algorithm now? If so, answer the following question. Would the algorithm be correct if, rather than sorting the requests by finish time, we had instead sorted them by start time? How about if we didn't sort them at all?

Computing the Final Schedule: So far we have seen how to compute the value of the optimal schedule, but how do we compute the schedule itself? This is a common problem that arises

in many DP problems, since most DP formulations focus on computing the numeric optimal value, without consideration of the object that gives rise to this value.

We use the accept[j] values to address this. We start with $W[n]$ and work backwards. We know that value of $W[j]$ arose from two distinct possibilities, accept or reject. If we accepted request j, then we add j to the schedule and continue with prior[j]. Otherwise, we add nothing to the schedule and we continue with $j-1$. The algorithm for generating the schedule is given in the code block below.

Computing Weighted Interval Scheduling Schedule

```
get-schedule() {                                // get the WIS schedule
    j = n                                       // start with the last request
    sched = empty
    while (j > 0) {
        if (accept[j]) {                        // accepted request j?
            prepend j to the front of sched
            j = prior[j]
        } else
            j = j-1
    }
    return sched                                // return the final schedule
}
```

The computation of the final schedule is illustrated in Fig. 6.

- Since accept[6] $=$ F, we *reject* 6 and continue with $6 - 1 = 5$.
- Since accept[5] $=$ T, we *accept* 5 and continue with prior[5] $= 3$.
- Since accept[3] $=$ F, we *reject* 3 and continue with $3 - 1 = 2$.
- Since accept[2] $=$ T, we *accept* 2 and continue with prior[2] $= 0$ (and terminate since $j = 0$).

We obtain the final schedule $\langle 2, 5 \rangle$.

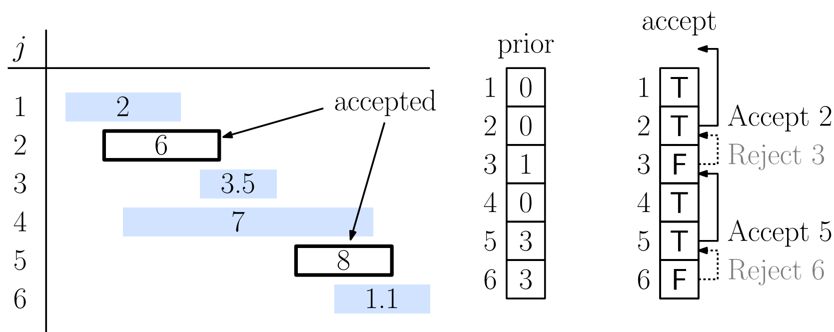

Fig. 6: Using the accept flags to compute the final schedule.