Algorithms & Complexity: Lecture 9, Dynamic Programming

Sam Barrett

March 1, 2021

Dynamic programming is very different from greedy algorithms, greedy algorithms follow a rule *blindly* whereas DP algorithms are more careful.

The general way a dynamic programming algorithm works is by building up a final solution from the solutions of multiple sub-problems. But how do we know which sub-problems to consider? And how do they contribute to the final solution?

1 Fibonacci Numbers

We can define the set of fibonacci numbers recursively as follows:

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_n &= F_{n-1} + F_{n-2}, \forall n \geq 2
\end{aligned}
$$

This recursive definition can easily be interpreted into an algorithm:

Algorithm 1: RecursiveFibonacci

```
1 if n = 0 then
2 |  return 0
3 if n = 1 then
4 |  return 1
5 else
6 |  return RecursiveFibonacci(n − 1) + RecursiveFibonacci(n − 2)
7 end
```

Intuitively, you can see that the path of this algorithm will form a (recursive) tree with the final solution being the root.

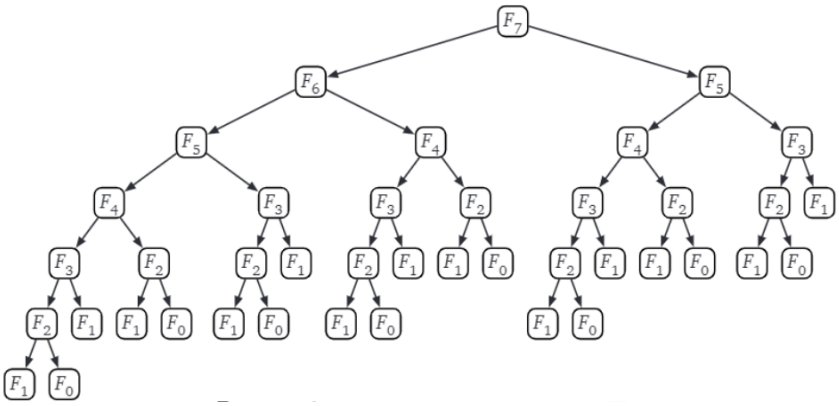

Figure 1: Recursive tree to compute F_7

1.1 Memo(r)isation of recursion

This method looks at the previous algorithm and asks whether it would be more efficient to store values on the first time they are computed so that they may be retrieved from a lookup table on subsequent recursions, for instance you can see that F_2 is recalculated in every sub-tree in Figure 1, If we were to store it we could remove the need for this calculation to be repeated.

Algorithm 2: MemoisationFibonacci

```
1 if n = 0 then
2 |  return 0
3 if n = 1 then
4 |  return 1
5 if F[n] is undefined then
6 |  F[n] ← MemoisationFibonacci(n-1) + MemoisationFibonacci(n-2)
7 end
```

Using this algorithm we can trim or *prune* the tree that needs to be generated.

1.2 Iterative approach

At this point we are already maintaining an array, so why do we not fill it up iteratively? Recursion acts as a layer of abstraction, making our process closer to the formal (recursive) definition of the set. We can remove this and instead write our algorithm as:

What is the running time of such an algorithm?

- we store n items in our array F

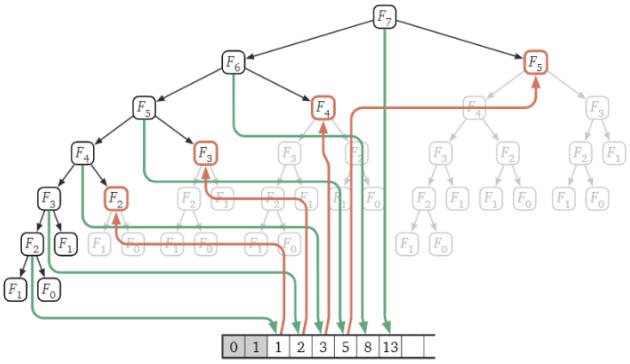

Figure 2: Computing F_7 via memoisation

Algorithm 3: IterativeFibonacci(n)

```
1 F[0] ← 0
2 F[1] ← 1
3 for i ← 2 to n do
4 |   F[i] ← F[i − 1] + F[i − 2]
5 end
```

- Computing each new entry needs 2 lookups and one addition.
  Therefore, the total running time to compute F_n is $O(n)$

2 Interval Scheduling Problem

2.1 Without weights (lecture 8 recap)

- We are given a set of n requests $R = \{\texttt{Req}(1), \texttt{Req}(2), \ldots, \texttt{Req}(i), \ldots, \texttt{Req}(n)\}$
- $\texttt{Req}(i)$ has a start time of $\texttt{Start}(i)$ and a finish time of $\texttt{Finish}(i)$
- There is a machine which can handle one request at a time
- Two requests **conflict** if they overlap

The interval scheduling problem asks:

Problem 1 *(Interval Scheduling) Select a set $C \subseteq R$ of requests such that $|C|$ is maximised and no two requests from C conflict.*

2.2 Weighted

- We are given a set of n requests $R = \{\texttt{Req}(1), \texttt{Req}(2), \ldots, \texttt{Req}(i), \ldots, \texttt{Req}(n)\}$

- $\texttt{Req}(i)$ has a start time of $\texttt{Start}(i)$ and a finish time of $\texttt{Finish}(i)$
- **Additionally, each request $\texttt{Req}(i)$ has a weight given by $\texttt{Weight}(i)$**
- There is a machine which can handle one request at a time
- Two requests **conflict** if they overlap

The weighted interval scheduling problem asks:

Problem 2 *(Weighted interval scheduling problem) Select a set $C \subseteq R$ of requests such that $\sum_{i \in C} \texttt{Weight}(i)$ is maximised and no two requests from C conflict.*

I.e. maximise the weight of all chosen requests.
The algorithm seen for unweighted ISP does not hold. (Algorithm)

Algorithm 4: Select requests by increasing order to finish times

```
1 Let R = {Req(1), Req(2), ..., Req(i), ..., Req(n)} be the set of all
    requests
2 Let C denote the set of requests that we select, initialise it as C = ∅
3 while R ≠ ∅ do
4 |   Find the request Req(i) ∈ R which has the smallest finish time.
5 |   Add Req(i) to C
6 |   Delete from R all requests that conflict with Req(i)
7 end
```

Remember: This is an example of a greedy algorithm. It does not take the newly added weights into account, therefore often finds a sub-optimal solution.
Instead to solve the weighted interval scheduling problem we:

Algorithm 5: Weighted interval scheduling algorithm

```
1 Begin by ordering requests in increasing order of finishing time.
2 M[0] = 0
3 for each request j ∈ 1..n do
4 |   M[j] = max{Weight(j) + M[Last(j)], M[j − 1]}
5 end
```

Where $\texttt{Last}(i)$ is given by:

$$Last(i) = \begin{cases} i & \text{the largest index } i \text{ s.t. } i \text{ is disjoint from } j \\ 0 & \text{if there is no request } i < j \text{ that is disjoint from } j \end{cases}$$

and is the last compatible request with i.

In this algorithm, $M[j]$ stores the value of the set of requests of maximum weight which can be chosen for the sub-instance containing requests $\{1, 2, \ldots, j\}$. Our goal is then to compute $M[n]$ from the entries $M[1], M[2], \ldots, M[n-1]$ (**Note** these have all been computed previously)

Our recurrence of $\max\{\texttt{Weight}(j) + M[\texttt{Last}(j)], M[j-1]\}$ is basically deciding whether request j is worth including in the final solution, if we had a higher total weight previously without j then we can ignore it (keeping our previous value), but if not, our result is the weight of j along with the result of the sub-instance concerned with the last compatible request with j ($\texttt{Last}(j)$).

2.2.1 Correctness

We can prove the correctness of this algorithm by induction on j. Our base case will be $j = 0$ and our inductive step essentially argues the correctness of our recurrence.

2.2.2 Running time

The running time of this algorithm can be broken down into:

- Sorting the requests in increasing order of finishing time, this can be done in $O(n \log n)$ time.
- We can now find $\texttt{Last}(j)$ for $1 \leq j \leq n$ in $O(n)$ time.
- Filling $M[j]$ requires a comparison between two existing entries from M along with an addition and a max operation. This can be done in $O(1)$ time.

  Therefore filling the entire array M can be done in $O(n)$ time

3 Bellman-Ford algorithm for finding shortest paths

We have previously looked at Dijkstra's algorithm for finding shortest paths. However, this algorithm has one major flaw: it does not work when we have negative edge-lengths.

One might think a simple solution to this problem is to add a constant of the largest negative length to all edges in a graph. I.e. if the lowest edge value in a graph G is -2, by adding 2 to all edges there are no more negative edge lengths. This does not work in practise as it affects what paths are the shortest, preferring paths with fewer edges. (Change in path cost is equal to number of edges multiplied by the constant added).

An alternative algorithm that deals with this is the **Bellman-Ford** algorithm.

This algorithm assumes that there are no **negative cycles**. As this would imply that the optimal route is infinite in number of edges!

By assuming no negative cycles, we can say that for any two vertices s and t, there is a shortest $s \to t$ path which has **at most** $n-1$ edges. We can show this to be correct:

- Let P be a shortest $s \to t$ path with the fewest number of edges.
- If a vertex x repeats on P, then delete the $x \to x$ cycle from P
  - Therefore the number of edges in P decreases
  - The length of P cannot increase as there as no negative edges.

3.1 Defining our algorithm

We can say, for each vertex $v \in V$ and each $0 \leq i \leq n-1$, let $\texttt{OPT}[i, v]$ denote the shortest $s \to v$ path having at most i edges.

We can now set up our recurrence:

1. If the shortest $s \to v$ path actually uses at most $i-1$ edges out of the original i then the value is $\texttt{OPT}[i-1, v]$
2. Otherwise, the last (i^{th}) edge has to be (w, v) for some $w \in V$ as our shortest $s \to v$ path uses all i edges.
   - The cost of this path is $\texttt{OPT}[i-1, w] + \texttt{cost}(w, v)$
   - We need to minimise this over all w s.t. (w, v) is an edge in G

We must take a minimum of these two choices:

$$\texttt{OPT}[i, v] = \min \left\{ \texttt{OPT}[i-1, v], \min_{w \in V} \left(\texttt{length}(w, v) + \texttt{OPT}[i-1, w] \right) \right\}$$

We can now formulate our algorithm: (See Algorithm 6)

3.1.1 Running time

- $\texttt{OPT}$ has $O(n^2)$ entries
- Computing each entry needs to lookup (and compute)$O(n)$ entries.
  - Computing $\texttt{OPT}[i, v]$ needs knowledge of $\texttt{OPT}[i-1, x], \forall x \in V$
  - Needs to perform $O(n)$ min operations and $O(n)$ additions
- Total running time $O(n^3)$

Algorithm 6: Shortest path from a vertex s to all other vertices

We maintain a $n \times n$ table indexed by $0 \leq i \leq (n-1)$ and $v \in V$ where $\texttt{OPT}[i, v]$ stores the length of the shortest $s \to v$ path which uses at most i edges.
Define $\texttt{OPT}[0, s] = 0$ and $\texttt{OPT}[0, v] = \infty$ for each $v \in V, v \neq s$
for $i = 1..n$ **do**
    for $v \in V$ **do**
        $$\texttt{OPT}[i, v] = \min \left\{ \texttt{OPT}[i-1, v], \min_{w \in V} \left(\texttt{length}(w, v) + \texttt{OPT}[i-1, w] \right) \right\}$$
    end
end
return $\texttt{OPT}[n-1, v]$ for each $v \in V$