Algorithm Design, Analysis & Complexity

Lecture 3 - Dynamic Programming

Koushik Pal

University of Toronto

May 18, 2021

Fibonacci Series

Definition (Fibonacci Series)

$F_0 = F_1 = 1$
$F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$.

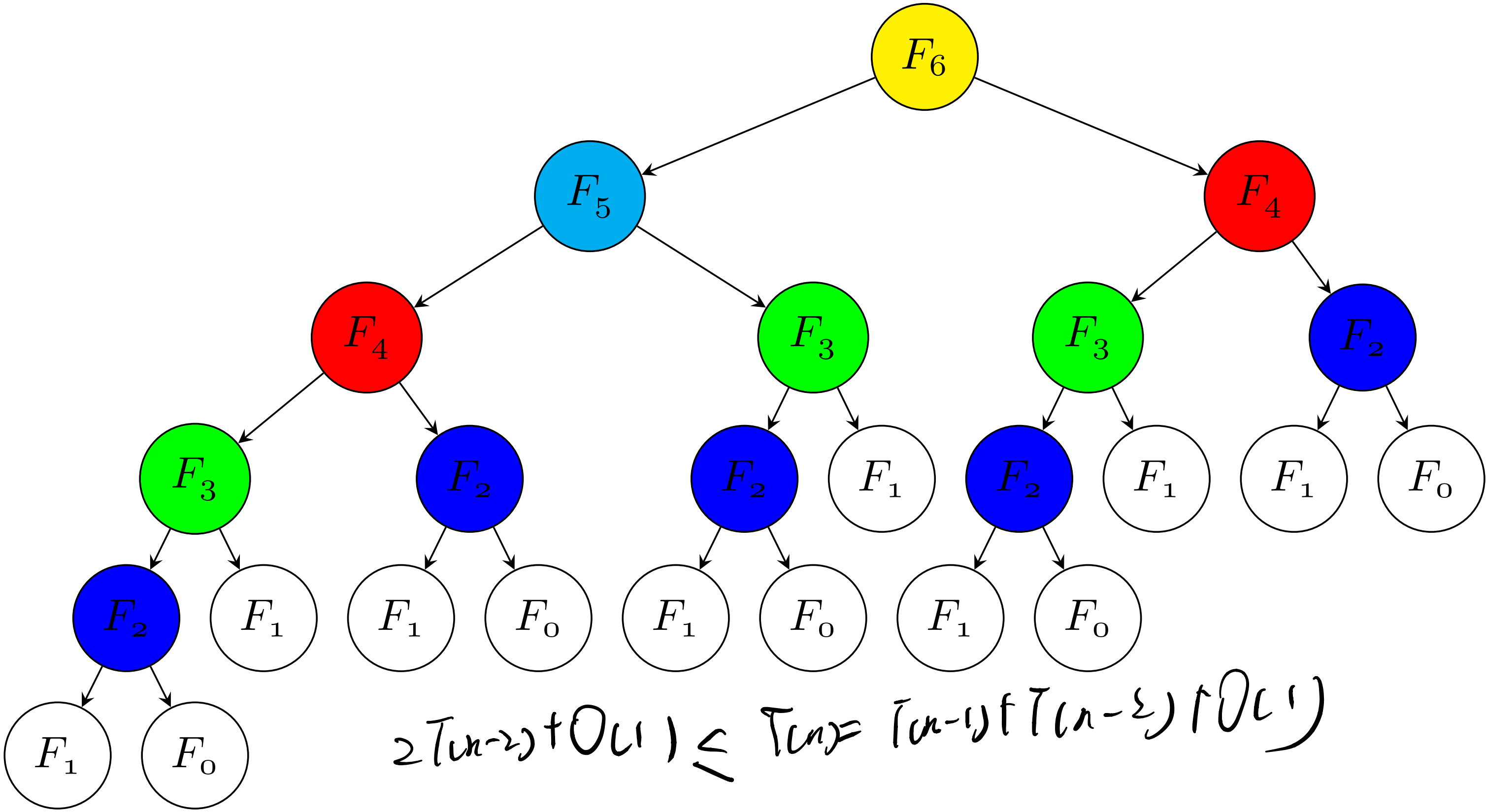
Goal: Compute the n^{th} Fibonacci number F_n.

---

procedure $\textsc{Fibonacci}(n)$
    if $n == 0$ or $n == 1$ **then**
        return 1
    else
        return $\textsc{Fibonacci}(n-1) + \textsc{Fibonacci}(n-2)$

Complexity: ???

Exponential Explanation



$$2T(n-2)+O(1) \le T(n) = T(n-1)+T(n-2)+O(1)$$

$$\le 2T(n-1)+O(1)$$

Complexity: $\Theta(2^n)$.

This happens simply because we repeat our calculations!

To make it efficient, we store all computed results.

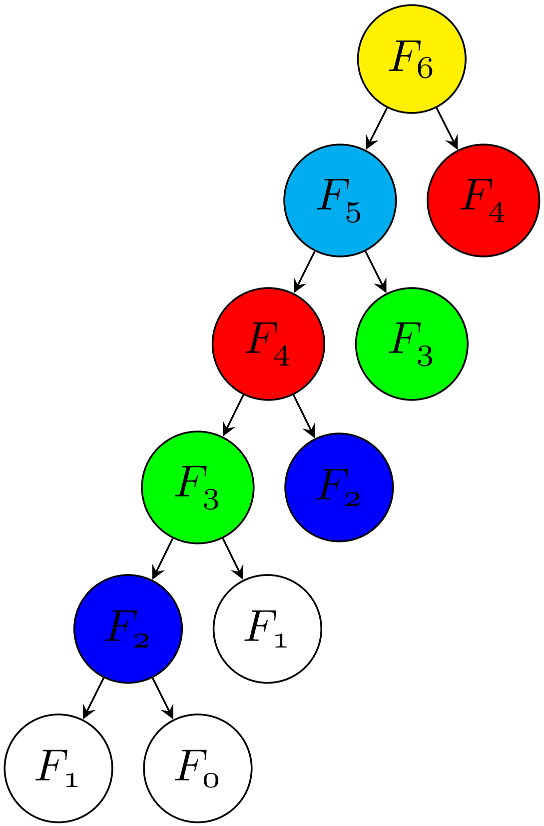
Memoization

procedure $\textsc{FibonacciMemoizedWrapper}(n)$
    $M = []$
    $M[0] = M[1] = 1$
    return $\textsc{FibonacciMemoized}(n, M)$

procedure $\textsc{FibonacciMemoized}(n, M)$
    if $M[n]$ is defined **then**
        return $M[n]$
    else
        $M[n-1] = \textsc{FibonacciMemoized}(n-1, M)$
        $M[n-2] = \textsc{FibonacciMemoized}(n-2, M)$
        $M[n] = M[n-1] + M[n-2]$
        return $M[n]$

Complexity: ???

Improved Complexity



Complexity: $\Theta(n)$ (each call to the recursive function computes and fills one value of M and there are only n values to fill).

Iterative Approach

Now turn this recursive top-down algorithm into an iterative bottom-up algorithm.

procedure $\textsc{FibonacciIterative}(n)$
    $M = []$
    $M[0] = M[1] = 1$
    for $k = 2$ to n **do**
        $M[k] = M[k-1] + M[k-2]$
    return $M[n]$

Complexity: $\Theta(n)$.

Dynamic Programming

A dynamic programming algorithm for an optimization problem is basically a recursive solution that uses memoization to solve repeated calls to the same subproblems.

Indication that dynamic programming might apply to a problem:

Definition

A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems.

Definition

A problem is said to have overlapping subproblems if subproblems share subsubproblems, or equivalently, if the problem can be broken down into subproblems which are reused several times.

Weighted Interval Scheduling

Problem (Weighted Interval Scheduling)

Given n *requests* $\{r_1, r_2, \ldots, r_n\}$*, with each request specifying a start time* s_i*, a finish time* t_i*, and having a value or weight* v_i*, select a subset* $S \subseteq \{r_1, r_2, \ldots, r_n\}$ *of* **mutually compatible** *requests so as to* **maximize** *the sum of the values of the selected requests* $\sum_{r_i \in S} v_i$*.*

Note

Interval scheduling is a special case of weighted interval scheduling with $v_i = 1$ for all $1 \leq i \leq n$.

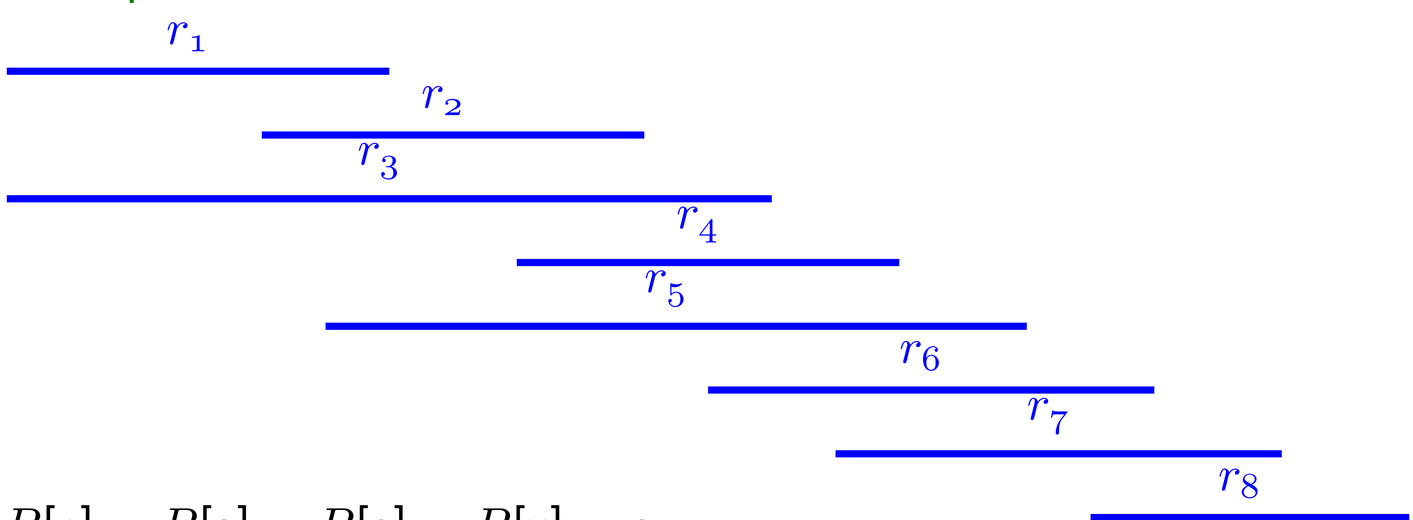
Weighted Interval Scheduling

Preprocessing: Assume the requests $\{r_1, r_2, \ldots, r_n\}$ are sorted by their **finish times**, i.e., $t_1 \leq t_2 \leq \cdots \leq t_n$.

Also, define $P[j]$, for a request r_j, as follows:

$$P[j] = \begin{cases} i, & \text{where } i < j \text{ is largest s.t. } r_i \text{ is compatible with } r_j \\ 0, & \text{if no request } r_i \text{ is compatible with } r_j \text{ for } i < j. \end{cases}$$

Example



$P[1] = P[2] = P[3] = P[5] = 0$,
$P[4] = 1, P[6] = 2, P[7] = 3, P[8] = 5$.

Weighted Interval Scheduling

Exercise. Compute P efficiently (in $\Theta(n \lg n)$ steps)!

Observation

Let $\mathcal{O}$ be an optimal solution. Then there are two possibilities:

1. $r_n \in \mathcal{O}$*: then the remaining requests in* $\mathcal{O}$ *form a subset of* $\{r_1, r_2, \ldots, r_{P[n]}\}$.
2. $r_n \notin \mathcal{O}$*: then the remaining requests in* $\mathcal{O}$ *form a subset of* $\{r_1, r_2, \ldots, r_{n-1}\}$.

Weighted Interval Scheduling

Definition

Let $\mathcal{O}_j$ denote an optimal solution to the problem consisting of requests $\{r_1, \ldots, r_j\}$, and OPT_j denote the value of this solution.

Goal

To find $\mathcal{O}_n$ and OPT_n.

By the previous observation,

1. if $r_j \in \mathcal{O}_j$, then $OPT_j = v_j + OPT_{P[j]}$
2. if $r_j \notin \mathcal{O}_j$, then $OPT_j = OPT_{j-1}$

Consequently, $OPT_j = \max\{v_j + OPT_{P[j]}, OPT_{j-1}\}$.

Moreover, request r_j belongs to an optimal solution if and only if

$$v_j + OPT_{P[j]} \geq OPT_{j-1}.$$

Non-DP Solution

```
procedure Compute-Opt-Wrapper(n)
    return Compute-Opt(n)
```

```
procedure Compute-Opt(j)
    if j == 0 then
        return 0
    else
        return max{v_j + Compute-Opt(P[j]),
                        Compute-Opt(j−1)}
```

Proof of Correctness

Proof.

Proof by mathematical induction.

Base Case: $OPT_0 = 0 = \textsc{Compute-Opt}(0)$ (no requests, no value).

Ind. Hyp.: Assume $\textsc{Compute-Opt}(k) = OPT_k$ for $1 \le k < j$.

Ind. Step: Show the result holds for j.

Proof.

$$
\begin{aligned}
OPT_j &= \max\{v_j + OPT_{P[j]}, OPT_{j-1}\} \\
&= \max\{v_j + \textsc{Compute-Opt}(P[j]), \\
&\qquad\qquad \textsc{Compute-Opt}(j-1)\} \quad \textit{(by Ind. Hyp.)} \\
&= \textsc{Compute-Opt}(j)
\end{aligned}
$$

□

Complexity

Complexity: $T(n) = T(n-1) + T(P[n])$.

If $P[j] = j-1$ for all j, then $T(n) = 2T(n-1)$, which yields $T(n) = 2^n$.

It is exponential because we are repeating our calculations.

Solution: Memoization!

DP Solution

procedure M-COMPUTE-OPT-WRAPPER(n)
    $M = []$
    $M[0] = 0$
    return M-COMPUTE-OPT(n, M)

procedure M-COMPUTE-OPT(j, M)
    if $M[j]$ is defined **then**
        return $M[j]$
    else
        $M[P[j]] =$ M-COMPUTE-OPT($P[j], M$)
        $M[j-1] =$ M-COMPUTE-OPT($j-1, M$)
        $M[j] = \max\{v_j + M[P[j]], M[j-1]\}$
        return $M[j]$

Complexity: $\Theta(n)$ (each function call computes and fills one value of M, and M has only n values to fill).

Iterative Approach

Now turn this recursive top-down algorithm into an iterative bottom-up algorithm.

---

procedure $\textsc{M-Bottom-Up-Compute-Opt}(n)$
    Let $M[0, \ldots, n]$ be initialized to $-\infty$
    $M[0] = 0$
    for $j = 1$ to n **do**
        $M[j] = \max\{v_j + M[P[j]], M[j-1]\}$
    return $M[n]$

Complexity: $\Theta(n)$.

Overall Complexity: Since sorting of the requests by their finish times takes $\Theta(n \lg n)$ and computing the array P takes $\Theta(n \lg n)$ time, the overall complexity of the whole algorithm is $\Theta(n \lg n)$.

Optimal Solution

Finally, use M to compute an optimal solution, i.e., a subset of compatible requests that has the maximum sum of values.

```
procedure FIND-SOLUTION-WRAPPER(n, M)
    S = []
    return FIND-SOLUTION(n, M, S)
```

```
procedure FIND-SOLUTION(j, M, S)
    if j == 0 then
        return S
    else if v_j + M[P[j]] > M[j-1] then
        S = S.append(r_j)
        return FIND-SOLUTION(P[j], M, S)
    else
        return FIND-SOLUTION(j-1, M, S)
```

Complexity: $\Theta(n)$.

Five steps of Dynamic Programming

1. Defining the optimal substructure / recursive structure
2. Array definition for memoization
3. Defining the recurrence relation in terms of the array
4. Bottom-up iterative algorithm
5. Find one optimum solution using the array values.

Brute Force vs. Greedy vs. DP

Brute force algorithm explores every possible solution from the solution space.

Dynamic Programming algorithm first finds optimal solutions to subproblems and then makes an informed choice. It's important to make sure that there are only a polynomial number of subproblems.

Greedy algorithm first makes a "greedy" choice — the choice that looks best at the time — and then solves a resulting subproblem, without bothering to solve all possible related smaller subproblems.