# Dynamic Programming

COMP 6651 – Algorithm Design Techniques

Denis Pankratov

# Motivating example

The Fibonacci sequence:

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

The first few terms of the sequence are

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

The associated computational problem:

**Input**: $n \geq 0$

**Output**: $F_n$

# Simple recursive solution

```
$Fib(n)$
    $\boldsymbol{if}\ n \leq 1$
        $\boldsymbol{return}\ 1$
    $\boldsymbol{else}$
        $\boldsymbol{return}\ Fib(n-1) + Fib(n-2)$
```

Let $T(n) =$ number of addition operations

$T(n) = T(n-1) + T(n-2) + 1$ if $n \geq 2$
$T(0) = T(1) = 0$

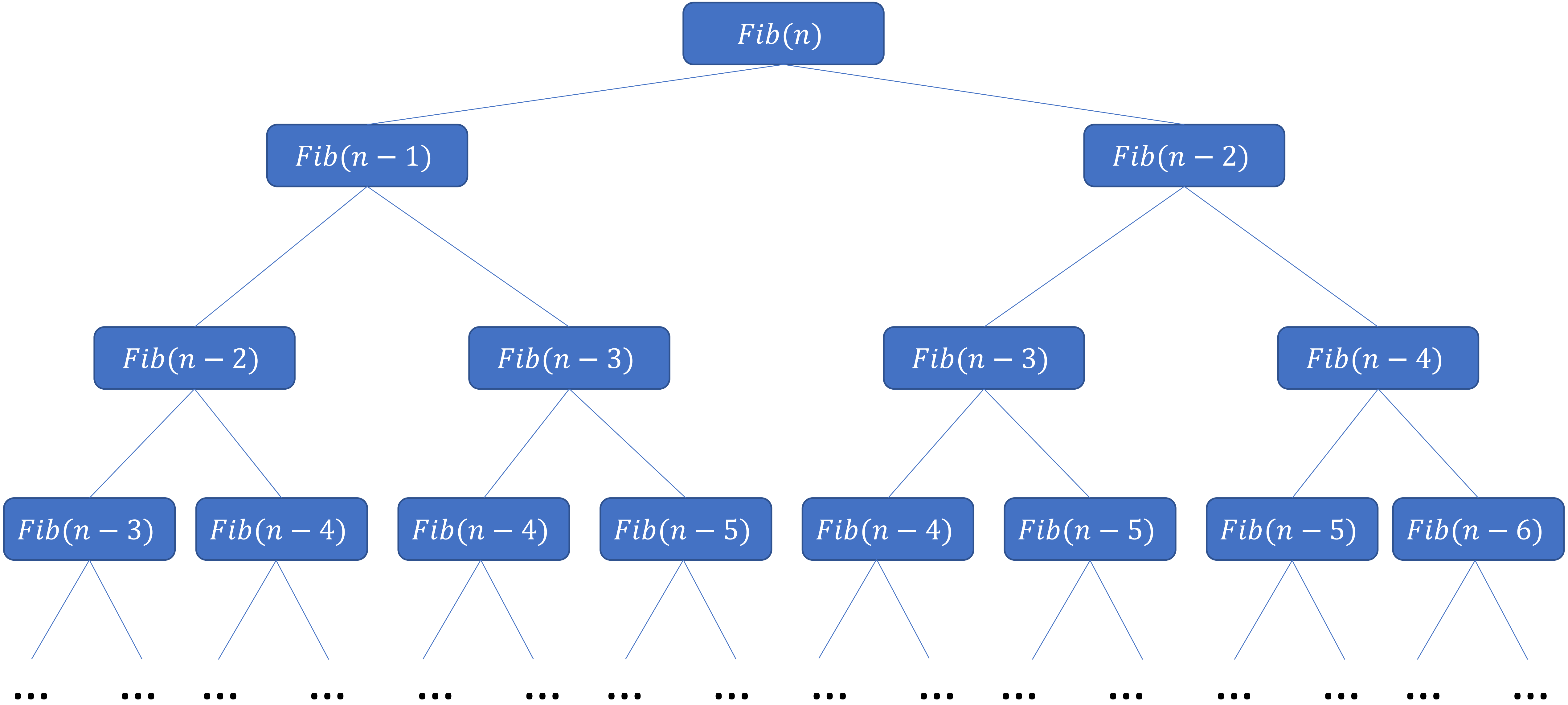
How large is $T(n)$?

Notice that $T(n)$ is monotone, therefore

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ &\geq T(n-2) + T(n-2) + 1 \\ &= 2T(n-2) + 1 \end{aligned}$$

Every two steps the value of $T(n)$ doubles, therefore $T(n) = \Omega\left(2^{\frac{n}{2}}\right)$

This is exponential! We should be able to do better



$Fib(n)$

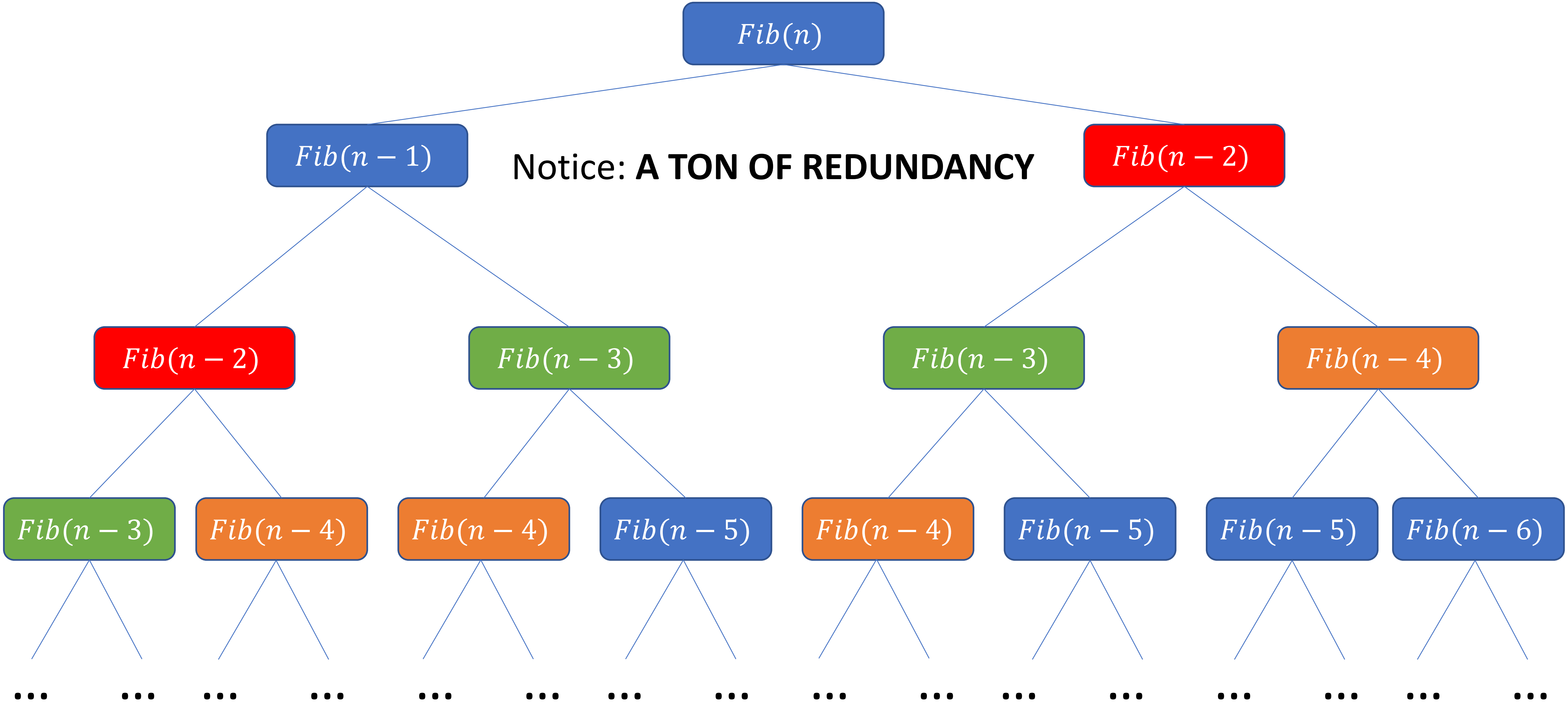
$Fib(n-1)$ $Fib(n-2)$

$Fib(n-2)$ $Fib(n-3)$ $Fib(n-3)$ $Fib(n-4)$

$Fib(n-3)$ $Fib(n-4)$ $Fib(n-4)$ $Fib(n-5)$ $Fib(n-4)$ $Fib(n-5)$ $Fib(n-5)$ $Fib(n-6)$

… … … … … … … … … … … … … … … …

Notice: **A TON OF REDUNDANCY**

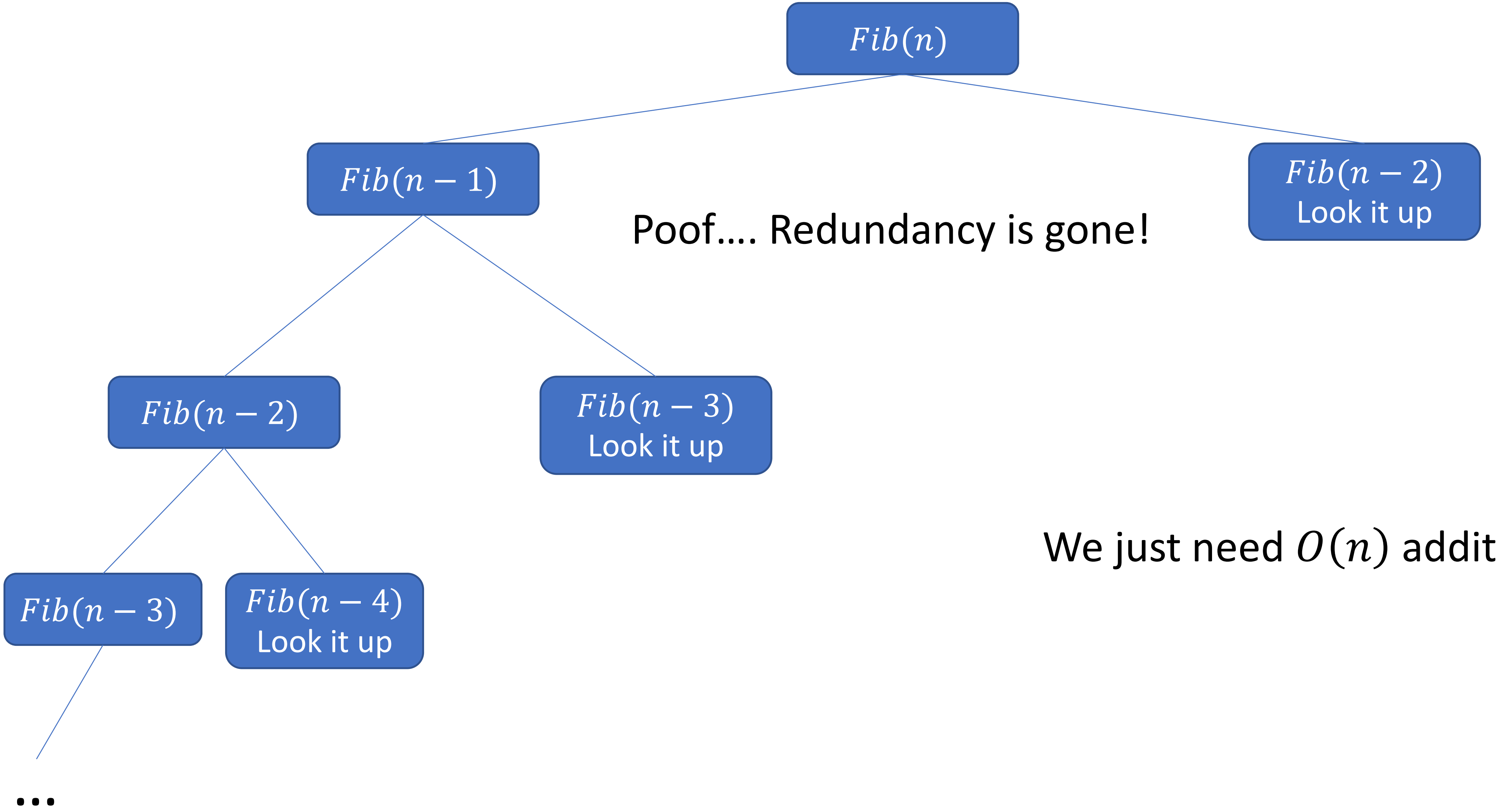


In fact, there will be an **EXPONENTIAL AMOUNT** of redundant calculations

Obvious solution: remember Fibonacci numbers that you computed before and look them up when you need them!

```
$Fib(n)$
    $F[0..n]$ – initialize all entries of a global array to $-1$ (indicating “not computed yet”)
    $F[0] \leftarrow F[1] \leftarrow 1$
    $\boldsymbol{return}\ RecFib(n)$

$RecFib(n)$ // has access to global array $F[0..n]$
    $\boldsymbol{if}\ F[n] \neq -1$
        $\boldsymbol{return}\ F[n]$
    $\boldsymbol{else}$
        $\boldsymbol{return}\ F[n] \leftarrow RecFib(n-1) + RecFib(n-2)$
```



Poof.... Redundancy is gone!

We just need $O(n)$ additions

This is the essence of **dynamic programming**!

There are two approaches:

1. Recursion with overlapping subproblems + a table or a map to store solutions to subproblems
2. Similar to (1), but often you can get rid of recursion altogether and populate the table iteratively

Approach (1) is called **memoization** (NOT memoRization) in the context of dynamic programming.

Approach (2) is sometimes referred to as iterative dynamic programming.

# Approach (2) to computing the Fibonacci sequence

```
$IterativeFib(n)$
    $F[0..n]$ – initialize all entries of a global array to $-1$ (indicating “not computed yet”)
    $F[0] \leftarrow F[1] \leftarrow 1$
    $\boldsymbol{for}\ i = 2\ \boldsymbol{to}\ n$
        $F[i] \leftarrow F[i-1] + F[i-2]$
    $\boldsymbol{return}\ F[n]$
```

# Iterative approach vs memoization

- Iterative approach has a benefit of avoiding recursive function calls and function calls may be expensive in real-life programming.

- Memoization is easier because sometimes the pattern of recursive calls is not easy to understand.

- Memoization may use less memory if not all entries in the table need to be filled in

# Dynamic Programming approach

1. Find a structure of sub-problems parameterized by one or more variables

   *example*: $F_i$ ($i < n$) is a sub-problem of $F_n$, variable – index $i$
2. Optimal solution to a sub-problem should be reconstructable from optimal solutions to sub-problems (**optimal substructure property**)
3. Since sub-problems rely on sub-problems, which rely on sub-sub-problems, and so on, many of the sub-sub-…-sub-problems must be shared in order to achieve savings (**overlapping sub-problems property**)

Usually (3) follows from (1) and bounds on values that variables can achieve

*example*: index $i$ can range from $0$ to $n$, each sub-problem is solved only once, so total number of sub-problems is $n + 1$

# A simple way to state and analyze dynamic programming solutions and avoid errors

- “Semantics” – “meaning”

In stating dynamic programming solutions we can define two array:

**Semantic array**

describes in plain English the meaning of an entry in the array

example: $F[i]$ = the $i$th Fibonacci number

**Computational array**

gives a mathematical formula how to compute an entry in the array

example: $F[i] = F[i-1] + F[i-2]$ if $i \geq 2$ and $F[0] = F[1] = 1$

Pseudocode: implementation details of computational array

Correctness argument: proof that two arrays are equivalent

Correctness argument: semantic array = computational array

*“what you are meaning to compute and what you actually compute are the same thing”*

Often can be carried out by induction on the variables defining sub-problems

In case of Fibonacci sequences it is obvious

Sometimes, we can save space by “forgetting” sub-sub-…-sub-problems that are not going to be used.

```
$IterativeFib(n)$
    $F[0..n]$ – initialize all entries of a global array to $-1$ (indicating “not computed yet”)
    $F[0] \leftarrow F[1] \leftarrow 1$
    $\boldsymbol{for}\ i = 2\ \boldsymbol{to}\ n$
        $F[i] \leftarrow F[i-1] + F[i-2]$
    $\boldsymbol{return}\ F[n]$
```

$\Theta(n)$ space assuming each value fits in a single memory cell

```
$IterativeFib2(n)$
    $F_{-1} \leftarrow 1$
    $F_{-2} \leftarrow 1$
    $\boldsymbol{for}\ i = 2\ \boldsymbol{to}\ n$
        $F_{cur} \leftarrow F_{-1} + F_{-2}$
        $F_{-2} \leftarrow F_{-1}$
        $F_{-1} \leftarrow F_{cur}$
    $\boldsymbol{return}\ F_{-1}$
```

$\Theta(1)$ space assuming each value fits in a single memory cell

# Aside

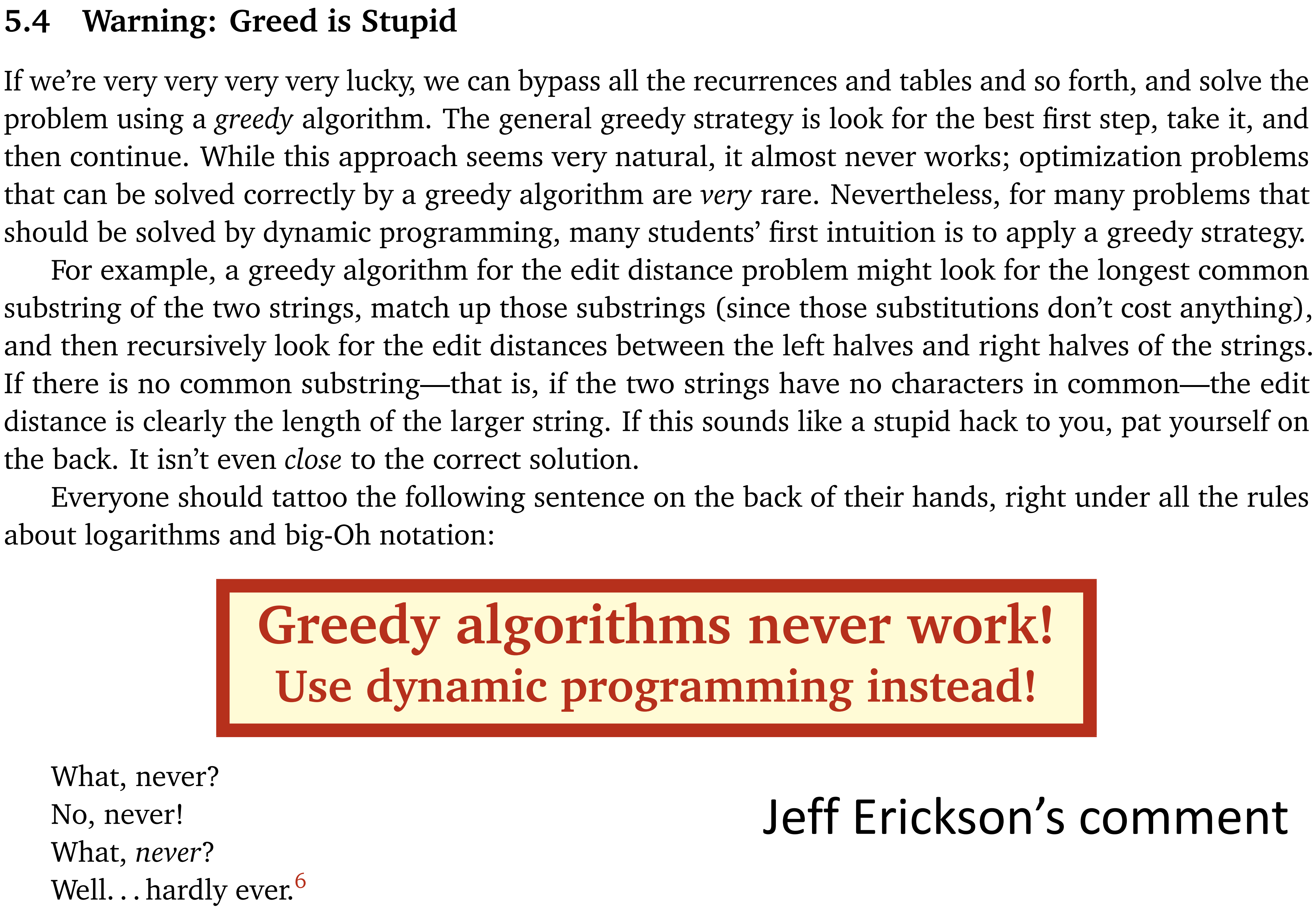
Why the word “**programming**”?

The term “dynamic **programming**” has very little to do with writing code

It was coined by Bellman Ford in 1950s (computer programming was rare)

**Programming** in this context has same meaning as “planning” as in

“yo dude, you should check out my workout **program**, it’s bench press on Mondays, biceps on Tuesdays, bench press on Wednesday, biceps on Thursday”



## 5.4 Warning: Greed is Stupid

If we're very very very very lucky, we can bypass all the recurrences and tables and so forth, and solve the problem using a *greedy* algorithm. The general greedy strategy is look for the best first step, take it, and then continue. While this approach seems very natural, it almost never works; optimization problems that can be solved correctly by a greedy algorithm are *very* rare. Nevertheless, for many problems that should be solved by dynamic programming, many students' first intuition is to apply a greedy strategy.

For example, a greedy algorithm for the edit distance problem might look for the longest common substring of the two strings, match up those substrings (since those substitutions don't cost anything), and then recursively look for the edit distances between the left halves and right halves of the strings. If there is no common substring—that is, if the two strings have no characters in common—the edit distance is clearly the length of the larger string. If this sounds like a stupid hack to you, pat yourself on the back. It isn't even *close* to the correct solution.

Everyone should tattoo the following sentence on the back of their hands, right under all the rules about logarithms and big-Oh notation:

**Greedy algorithms never work!**
**Use dynamic programming instead!**

What, never?
No, never!
What, *never*?
Well. . . hardly ever.[6]

Jeff Erickson's comment

# My view of greedy algorithms

- Previous comments are in the context of emphasizing DP algorithms and were a deliberate overstating of the point.
- Greedy algorithms may not always be optimal or as good as more sophisticated algorithms
- They work well enough either in terms of provable approximations or in practice in many cases
- Serve as a starting point for approaching a problem, serve as benchmarks
- Coming up with DP is difficult and often end up with a rather non-obvious solution. Greedy algorithms tend to be “conceptually simple”
- In some applications, e.g., auctions, conceptual simplicity is extremely important

# Longest Increasing Subsequence (LIS)

Input: $A[1..n]$ – an array of $n$ integers

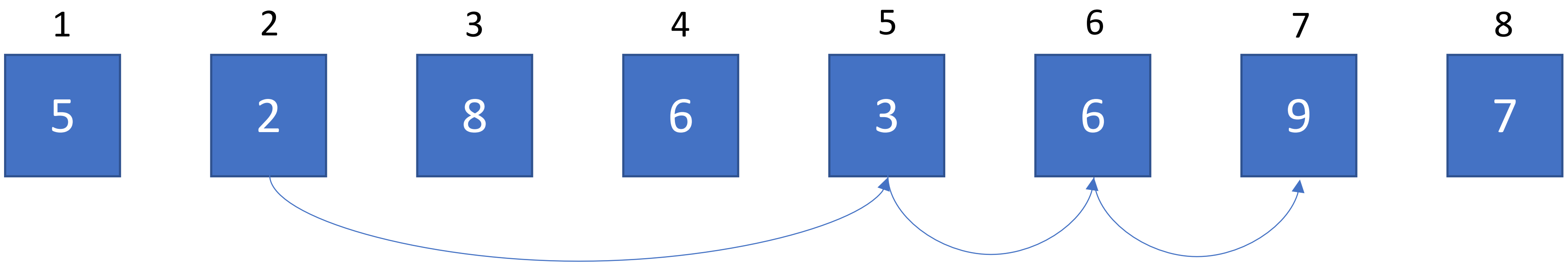
Output: length of a longest increasing subsequence, i.e.,

$m$ such that there exist indices $1 \leq i_1 < i_2 < \cdots < i_m \leq n$ and $A[i_1] < A[i_2] < \cdots < A[i_m]$ and there does not exist $m + 1$ indices with this property.

Notes: subsequence does not have to be contiguous, i.e., indices can skip over numbers

array elements need to get strictly larger

# Example



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 2 | 8 | 6 | 3 | 6 | 9 | 7 |

$i_1 = 2, i_2 = 5, i_3 = 6, i_4 = 7$

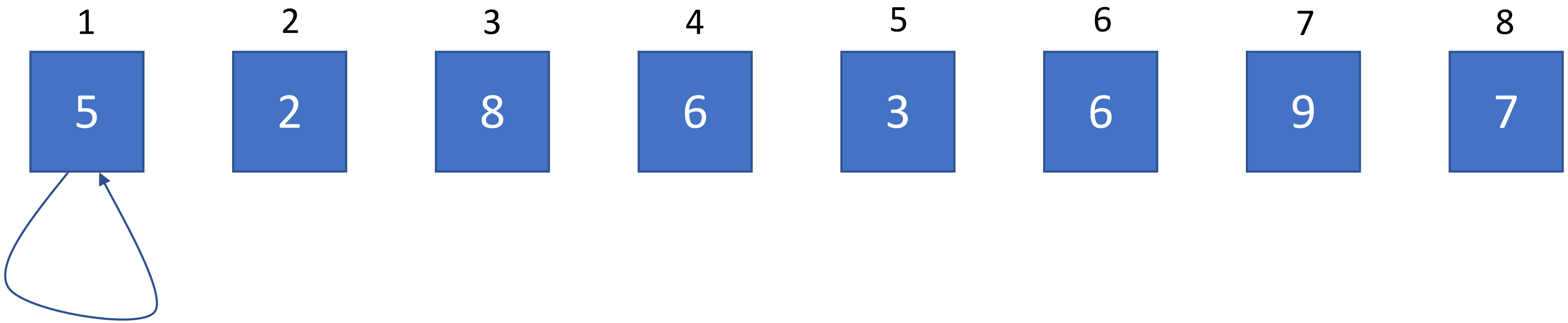
$A[i_1] = 2, A[i_2] = 3, A[i_3] = 6, A[i_4] = 9$

This is, in fact, a longest increasing subsequence

Its length is $4$, so the output should be $4$

# Semantic array

$D[i]$ = length of a longest increasing subsequences ending in element $A[i]$ in sub-array $A[1..i]$

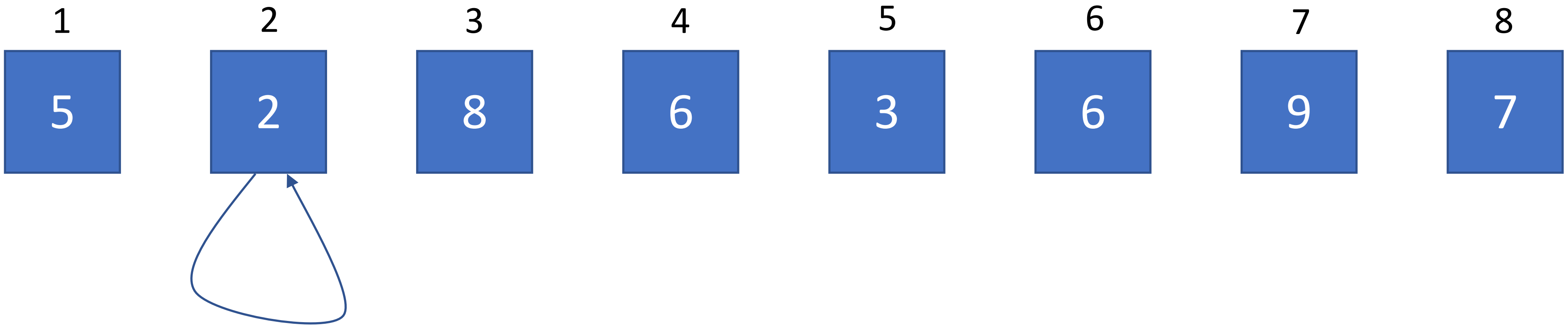


| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 2 | 8 | 6 | 3 | 6 | 9 | 7 |

$$D[1] = 1$$

Semantic array

$D[i]$ = length of a longest increasing subsequences ending in element $A[i]$ in sub-array $A[1..i]$

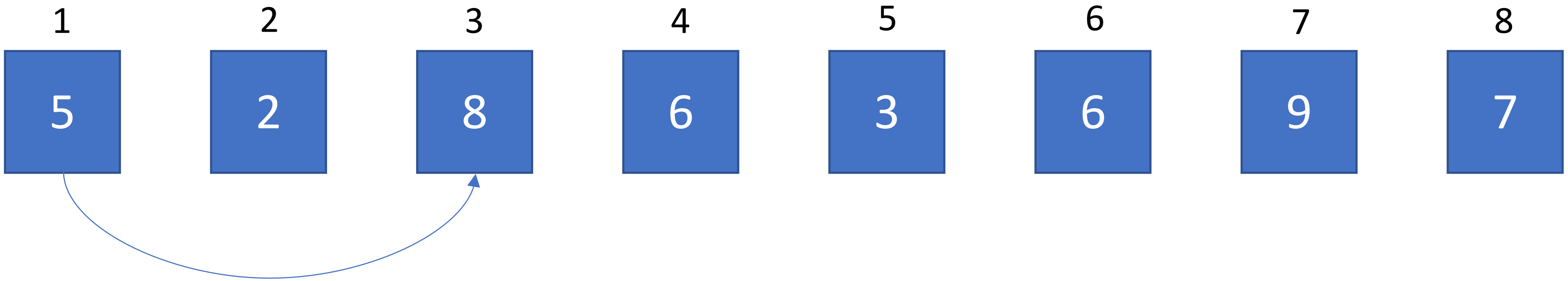


| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 2 | 8 | 6 | 3 | 6 | 9 | 7 |

$D[1] = 1 \qquad D[2] = 1$

Semantic array

$D[i]$ = length of a longest increasing subsequences ending in element $A[i]$ in sub-array $A[1..i]$

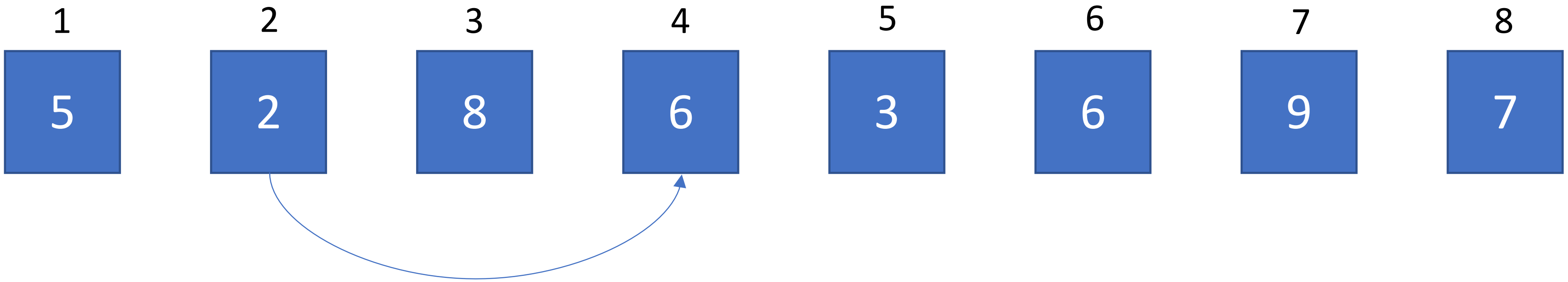


| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 2 | 8 | 6 | 3 | 6 | 9 | 7 |

$D[1] = 1 \quad D[2] = 1 \quad D[3] = 2$

## Semantic array

$D[i]$ = length of a longest increasing subsequences ending in element $A[i]$ in sub-array $A[1..i]$

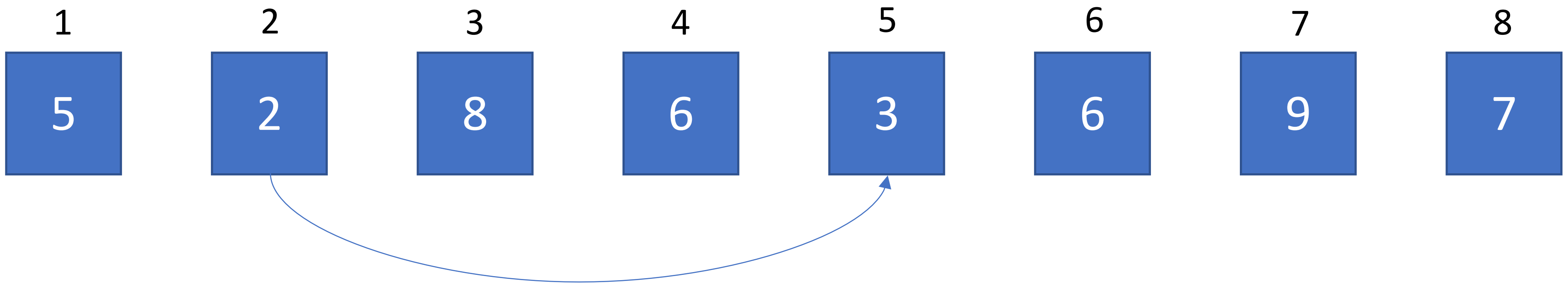


| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 2 | 8 | 6 | 3 | 6 | 9 | 7 |

$D[1] = 1 \quad D[2] = 1 \quad D[3] = 2 \quad D[4] = 2$

## Semantic array

$D[i]$ = length of a longest increasing subsequences ending in element $A[i]$ in sub-array $A[1..i]$

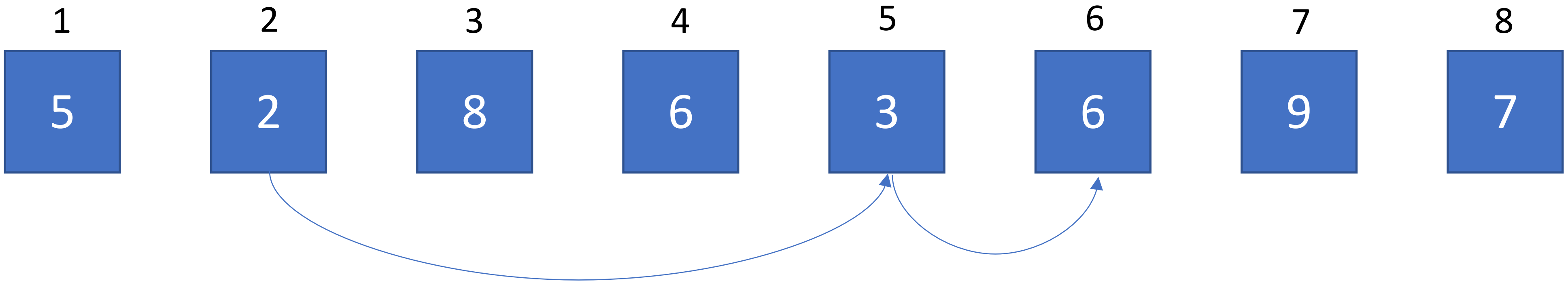


| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 2 | 8 | 6 | 3 | 6 | 9 | 7 |

$D[1] = 1 \quad D[2] = 1 \quad D[3] = 2 \quad D[4] = 2 \quad D[5] = 2$

Semantic array

$D[i]$ = length of a longest increasing subsequences ending in element $A[i]$ in sub-array $A[1..i]$

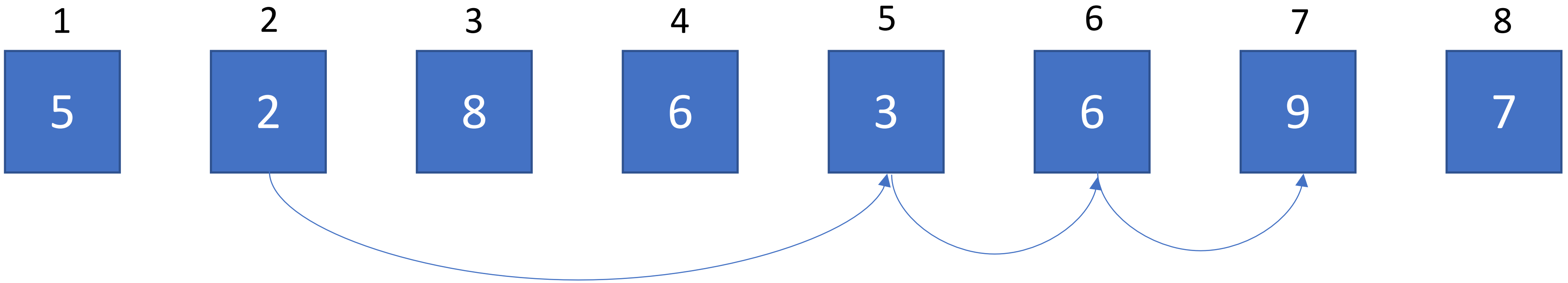


$D[1] = 1$ $D[2] = 1$ $D[3] = 2$ $D[4] = 2$ $D[5] = 2$

$D[6] = 3$

# Semantic array

$D[i]$ = length of a longest increasing subsequences ending in element $A[i]$ in sub-array $A[1..i]$



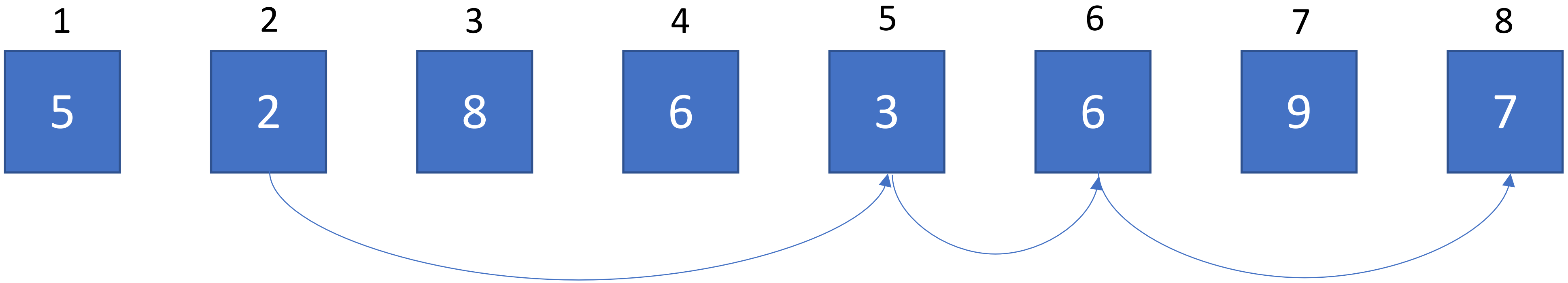
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 2 | 8 | 6 | 3 | 6 | 9 | 7 |

$D[1] = 1 \quad D[2] = 1 \quad D[3] = 2 \quad D[4] = 2 \quad D[5] = 2$

$D[6] = 3 \quad D[7] = 4$

## Semantic array

$D[i]$ = length of a longest increasing subsequences ending in element $A[i]$ in sub-array $A[1..i]$



$D[1] = 1$ $D[2] = 1$ $D[3] = 2$ $D[4] = 2$ $D[5] = 2$

$D[6] = 3$ $D[7] = 4$ $D[8] = 4$

Overall result is $\max_i D[i]$

Semantic array

$D[i]$ = length of a longest increasing subsequences ending in element $A[i]$ in sub-array $A[1..i]$

Computational array

$$D[i] = 1 + \max\{D[j] : j < i \; and \; A[j] < A[i]\}$$

Computational array = semantic array

Longest increasing subsequence ending in element $A[i]$ consists of an element $A[i]$ + longest increasing subsequence ending in element $A[j]$ for some $j < i$ and $A[j] < A[i]$

# Another way of looking at it

**Optimal substructure property:**

An optimal solution for subsequence $A[1..i]$ that includes $A[i]$ has to be 1 (for $A[i]$) + optimal solution for some $j < i$ with $A[j] < A[i]$

**Simple proof by contradiction ("cut and paste" argument):**

Suppose that an optimal solution for $A[1..i]$ that included $A[i]$ consisted of

1 (for $A[i]$) + sub-optimal solution for $j < i$ (*)

Since it is a feasible solution then $A[j] < A[i]$.

This is a contradiction! We can obtain a better solution than (*):

1 (for $A[i]$) + optimal solution for $j < i$.

# Pseudocode

```
$LIS(A[1..n])$
    instantiate $D[1..n]$
    $for\ i = 1\ to\ n$
        $m \leftarrow 0$
        $for\ j = 1\ to\ i - 1$
            $if\ A[j] < A[i]\ and\ D[j] > m$
                $m \leftarrow D[j]$
        $D[i] \leftarrow 1 + m$
    $return \max_{1 \le i \le n} D[i]$
```

$O(n)$

$O(n)$

Overall running time is $O(n^2)$

# Finding a longest increasing subsequence itself

What if we wanted not the length, but the subsequence itself?

We can do this by simply keeping track of choices resulting in computational array value

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 2 | 8 | 6 | 3 | 6 | 9 | 7 |

$$D[i] = 1 + \max\{D[j] : j < i \text{ } and \text{ } A[j] < A[i]\}$$

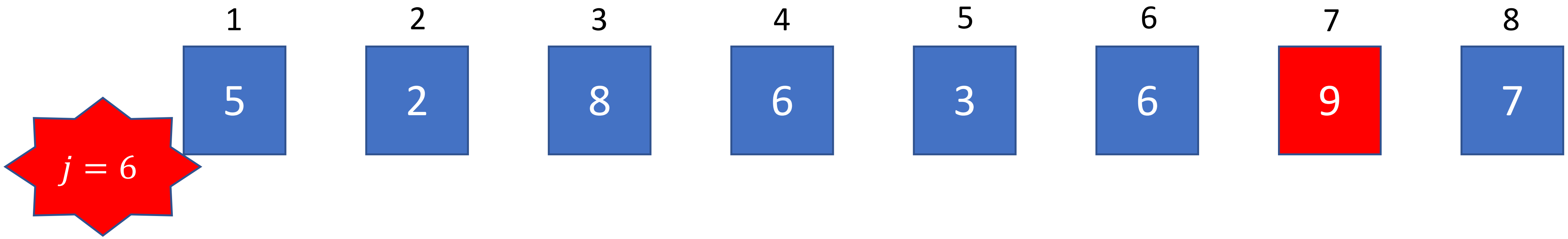
$D[1] = 1 \quad D[2] = 1 \quad D[3] = 2 \quad D[4] = 2 \quad D[5] = 2$

$D[6] = 3 \quad D[7] = 4 \quad D[8] = 4$

# Finding a longest increasing subsequence itself

What if we wanted not the length, but the subsequence itself?

We can do this by simply keeping track of choices resulting in computational array value



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 2 | 8 | 6 | 3 | 6 | 9 | 7 |

$j = 6$

What value of $j$ was responsible for this?

$$D[i] = 1 + \max\{D[j] : j < i \; and \; A[j] < A[i]\}$$

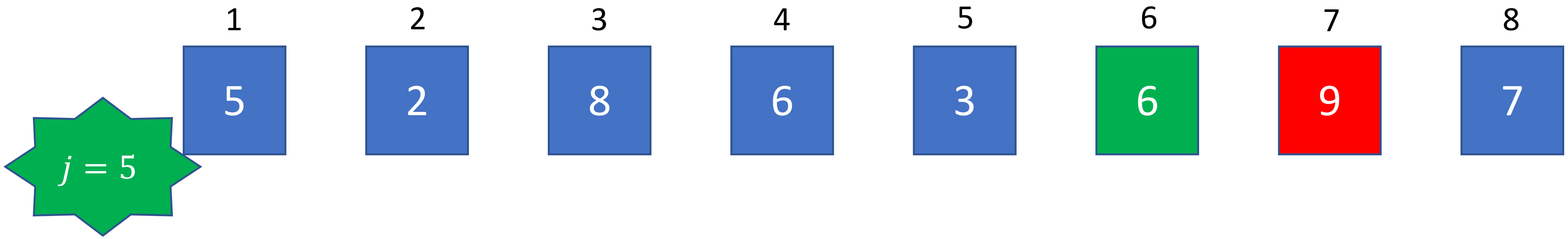
$D[1] = 1$  $D[2] = 1$  $D[3] = 2$  $D[4] = 2$  $D[5] = 2$

$D[6] = 3$  $D[7] = 4$  $D[8] = 4$

# Finding a longest increasing subsequence itself

What if we wanted not the length, but the subsequence itself?

We can do this by simply keeping track of choices resulting in computational array value



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 2 | 8 | 6 | 3 | 🟩 6 | 🟥 9 | 7 |

$j = 5$

What value of $j$ was responsible for this?

$$D[i] = 1 + \max\{D[j] : j < i \mathit{\ and\ } A[j] < A[i]\}$$

$D[1] = 1$ $D[2] = 1$ $D[3] = 2$ $D[4] = 2$ $D[5] = 2$

$D[6] = 3$ $D[7] = 4$ $D[8] = 4$

# Finding a longest increasing subsequence itself

What if we wanted not the length, but the subsequence itself?

We can do this by simply keeping track of choices resulting in computational array value

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 2 | 8 | 6 | 3 | 6 | 9 | 7 |

$j = 2$

What value of $j$ was responsible for this?

$$D[i] = 1 + \max\{D[j] : j < i \text{ } and \text{ } A[j] < A[i]\}$$

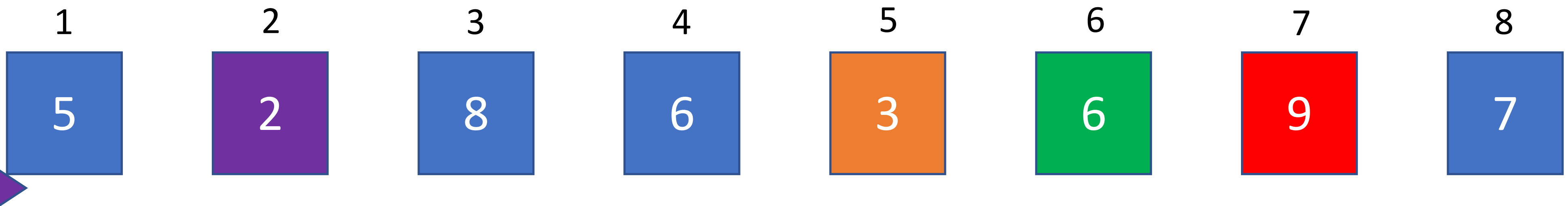
$D[1] = 1$ $D[2] = 1$ $D[3] = 2$ $D[4] = 2$ $D[5] = 2$

$D[6] = 3$ $D[7] = 4$ $D[8] = 4$

# Finding a longest increasing subsequence itself

What if we wanted not the length, but the subsequence itself?

We can do this by simply keeping track of choices resulting in computational array value



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 2 | 8 | 6 | 3 | 6 | 9 | 7 |

No *j* DONE

What value of *j* was responsible for this?

$$D[i] = 1 + \max\{D[j] : j < i \; and \; A[j] < A[i]\}$$

$D[1] = 1$ $D[2] = 1$ $D[3] = 2$ $D[4] = 2$ $D[5] = 2$

$D[6] = 3$ $D[7] = 4$ $D[8] = 4$

# Pseudocode

```
$LIS(A[1..n])$
    instantiate $D[1..n], Prev[1..n]$
    $for\ i = 1\ to\ n$
        $m \leftarrow 0$
        $Prev[i] \leftarrow i$
        $for\ j = 1\ to\ i - 1$
            $if\ A[j] < A[i]\ and\ D[j] > m$
                $m \leftarrow D[j]$
                $Prev[i] \leftarrow j$
        $D[i] \leftarrow 1 + m$
```

```
....
    $result \leftarrow \emptyset$
    $ind \leftarrow argmax_i\ D[i]$
    $result.push\_front(ind)$
    $while(Prev[ind] \neq ind)$
        $ind = Prev[ind]$
        $result.push\_front(ind)$
    $return\ result$
// returns indices corresponding to LIS
// if LIS is needed, push $A[ind]$ instead
```

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 2 | 8 | 6 | 3 | 6 | 9 | 7 |

$D[1] = 1 \quad D[2] = 1 \quad D[3] = 2 \quad D[4] = 2 \quad D[5] = 2$

$D[6] = 3 \quad D[7] = 4 \quad D[8] = 4$

$Prev[1] = 1, Prev[2] = 2, Prev[3] = 2, Prev[4] = 2, Prev[5]$
$= 2, Prev[6] = 5, Prev[7] = 6, Prev[8] = 6$

$ind\colon \quad 7 \qquad Prev[ind] = 6$

$result\colon (7)$

```
$result \leftarrow \emptyset$
$ind \leftarrow argmax_i\ D[i]$
$result.push\_front(ind)$
$while(Prev[ind] \neq ind)$
   $ind = Prev[ind]$
   $result.push\_front(ind)$
$return\ result$
```

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 2 | 8 | 6 | 3 | 6 | 9 | 7 |

$D[1] = 1 \quad D[2] = 1 \quad D[3] = 2 \quad D[4] = 2 \quad D[5] = 2$

$D[6] = 3 \quad D[7] = 4 \quad D[8] = 4$

$Prev[1] = 1, Prev[2] = 2, Prev[3] = 2, Prev[4] = 2, Prev[5] = 2, Prev[6] = 5, Prev[7] = 6, Prev[8] = 6$

$ind$: 6 $\qquad Prev[ind] = 5$

$result$: $(6,7)$

```
$result \leftarrow \emptyset$
$ind \leftarrow argmax_i\ D[i]$
$result.push\_front(ind)$
$while(Prev[ind] \neq ind)$
    $ind = Prev[ind]$
    $result.push\_front(ind)$
$return\ result$
```

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 2 | 8 | 6 | 3 | 6 | 9 | 7 |

$D[1] = 1 \quad D[2] = 1 \quad D[3] = 2 \quad D[4] = 2 \quad D[5] = 2$
$D[6] = 3 \quad D[7] = 4 \quad D[8] = 4$

$Prev[1] = 1, Prev[2] = 2, Prev[3] = 2, Prev[4] = 2, Prev[5]$
$= 2, Prev[6] = 5, Prev[7] = 6, Prev[8] = 6$

$ind\text{:} \quad 5 \qquad Prev[ind] = 2$
$result\text{:} (5,6,7)$

```
$result \leftarrow \emptyset$
$ind \leftarrow argmax_i\ D[i]$
$result.push\_front(ind)$
$while(Prev[ind] \neq ind)$
   $ind = Prev[ind]$
   $result.push\_front(ind)$
$return\ result$
```

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 5 | 2 | 8 | 6 | 3 | 6 | 9 | 7 |

$D[1]=1 \quad D[2]=1 \quad D[3]=2 \quad D[4]=2 \quad D[5]=2$

$D[6]=3 \quad D[7]=4 \quad D[8]=4$

$Prev[1]=1, Prev[2]=2, Prev[3]=2, Prev[4]=2, Prev[5]=2, Prev[6]=5, Prev[7]=6, Prev[8]=6$

$ind$: 2 $\qquad Prev[ind]=2$

$result$: $(2,5,6,7)$

```
$result \leftarrow \emptyset$
$ind \leftarrow argmax_i\ D[i]$
$result.push\_front(ind)$
$while(Prev[ind] \neq ind)$
   $ind = Prev[ind]$
   $result.push\_front(ind)$
$return\ result$
```

# Interval Scheduling Revisited

Recall Interval Scheduling

**Input**: $I[1..n]$ - a collection of half-open intervals where

$I[j].s$ – is the left-point of interval $j$ and

$I[j].f$ – is the right-point of interval $j$

In other words, interval $j$ is $[I[j].s, I[j].f)$

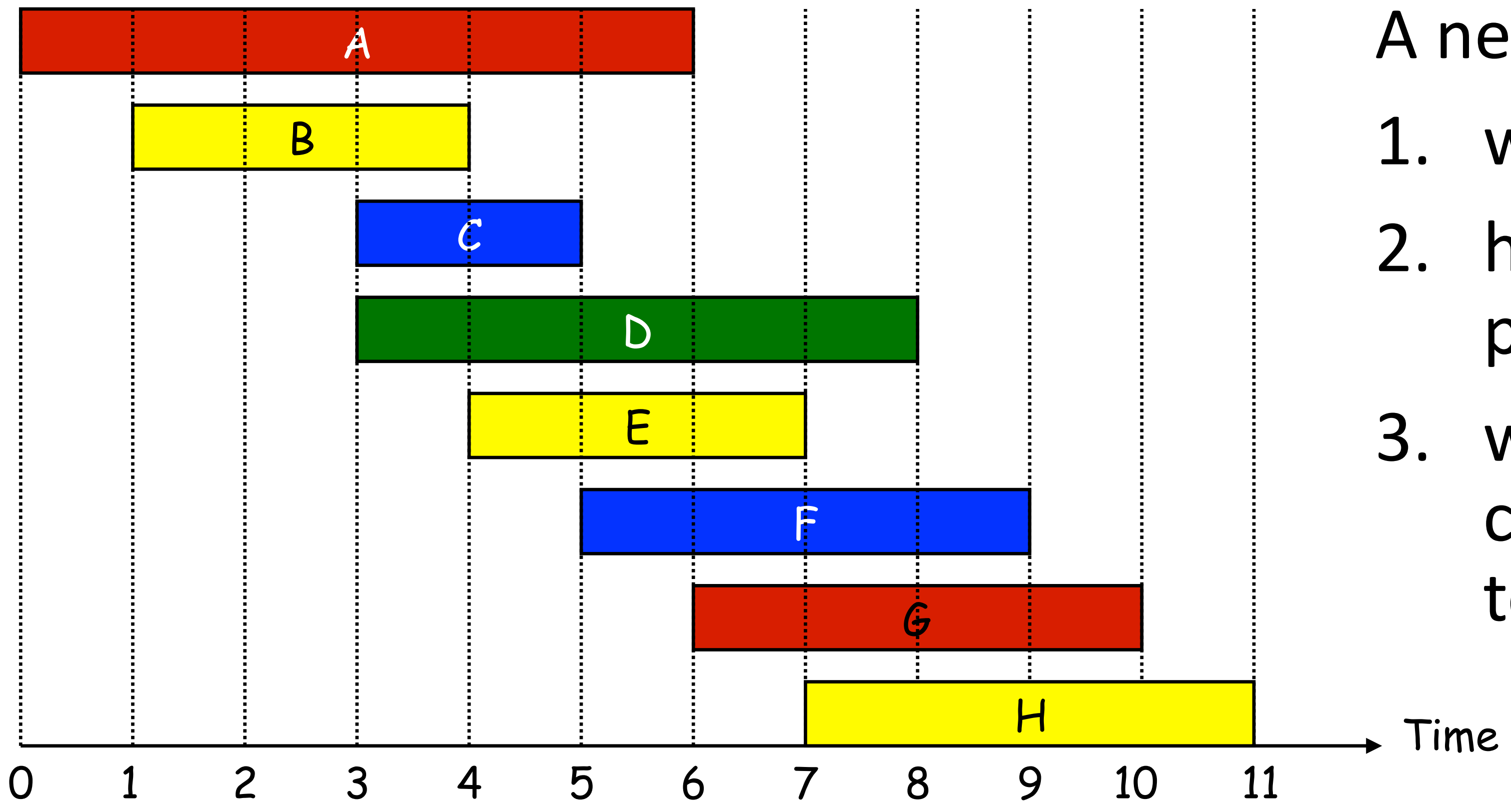
**Output**: A largest subcollection of non-overlapping intervals from $I[1..n]$

We saw that Earliest Finishing Time (EFT) greedy algorithm solves it optimally

Interpretation in scheduling:

intervals are execution times of jobs

find a largest subset of jobs that can be scheduled on a single machine



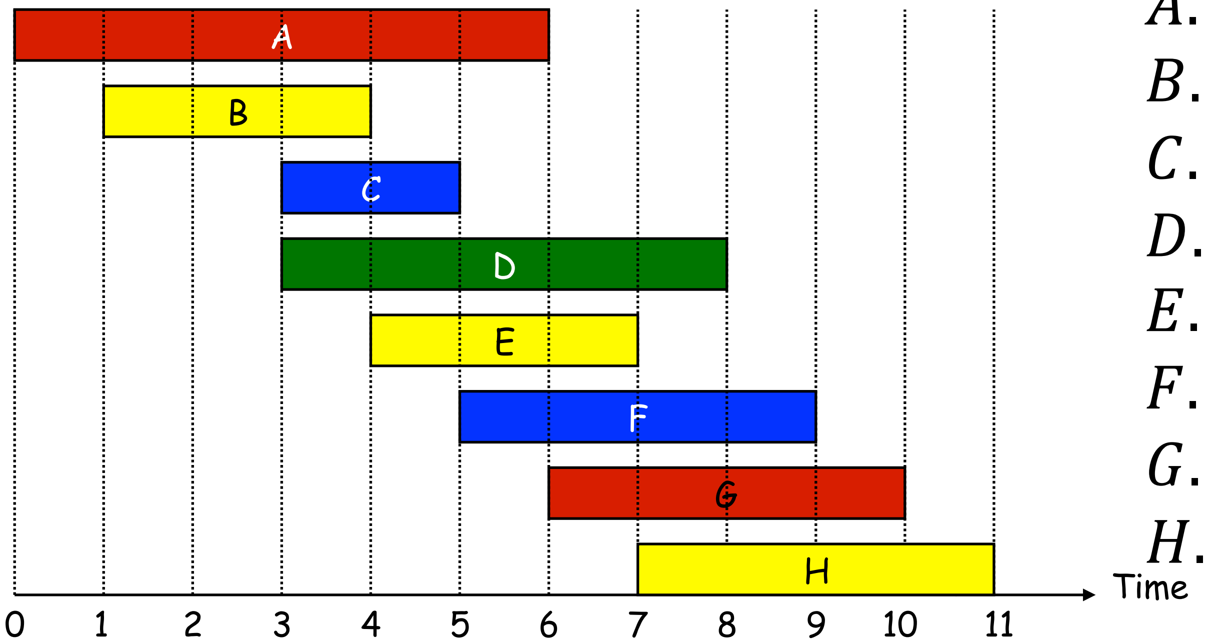
A new twist on the problem

1. what if jobs have weights?
2. higher weight means higher priority
3. wish to choose a subset of compatible jobs of maximum total weight

# Weighted Interval Scheduling

Input: $I[1..n]$ – a collection of half-open *weighted* intervals where interval $j$ is $[I[j].s, I[j].f)$ and its *weight* is $I[j].w$

Output: a subcollection of non-overlapping intervals of largest total weight



Weights:

$A.w = 10$

$B.w = 20$

$C.w = 10000$

$D.w = 25$

$E.w = 30$

$F.w = 4000$

$G.w = 5$

$H.w = 15$

Some feasible solutions:

$\{A, G\}$

$\{B, E, H\}$

$\{C, F\}$

$\{D\}$

Optimal solution: $\{C, F\}$

# Why not use greedy for Weighted Interval Scheduling?

Some possible orderings: by non increasing weight, by non increasing weight/interval length.

All the possible ways of ordering the input items that we can think of will not only fail to be optimal but can produce arbitrarily bad solutions for some instances

Moreover, for a general greedy formalization it can be proven that no greedy algorithm can provide a good solution (in worst case)

# The DP approach

Assume that intervals have been sorted by non-decreasing finishing times

$$I[1].f \leq I[2].f \leq \cdots \leq I[i].f$$

**Optimal substructure property:**

Optimal solution $OPT$ for the above either includes $I[i]$ or does not

if $I[i] \notin OPT$ then $OPT =$ optimal solution for $I[1..i-1]$

if $I[i] \in OPT$ then no interval in $OPT$ can end after $I[i].s$

moreover, the intervals ending by $I[i].s$ must be chosen optimally

**Proof**: standard “cut and paste” argument.

# Computing optimal **value**

**Semantic array:**

$D[i] =$maximum total weight obtainable by a non-overlapping set of intervals which are a subset of the first $i$ intervals $I[1..i]$

Solution to the whole problem is $D[n]$

For computational array, it is useful to define

$$before[i] = \text{the largest index } j \text{ such that } I[j].f \leq I[i].s$$

**Computational array:**

$$D[i] = \max(I[i].w + D\big[before[i]\big], D[i-1]) \text{ for } i > 0$$
$$D[0] = 0$$

```
$WeightedIntervalScheduling(I[1..n])$
    sort $I$ by non-decreasing attribute $I[j].f$
    instantiate arrays $D[0..n]$ and $before[0..n]$
    $D[0] \leftarrow 0$
    $before[0] \leftarrow 0$
    $\boldsymbol{for}\ i = 1\ \boldsymbol{to}\ n$
        $before[i] \leftarrow 0$
        $\boldsymbol{for}\ j = 1\ \boldsymbol{to}\ i - 1$                                  $O(n)$
            $\boldsymbol{if}\ I[j].f \leq I[i].s$                          $O(n)$
                $before[i] \leftarrow j$
        $D[i] \leftarrow \max(I[i].w + D[before[i]], D[i-1])$
    $\boldsymbol{return}\ D[n]$
```

Overall running time is $O(n^2)$

# Computing optimal **solution**

So far, we computed optimal value of a solution, i.e., maximum sum of weights of non-overlapping set of intervals

What about the set of intervals itself?

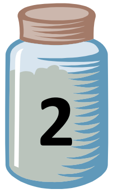
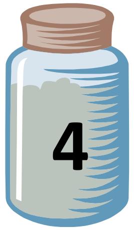
As before, keep track of which option resulted in the $D[i]$ entry:

```
$\boldsymbol{if}\ I[i].w + D\left[before[i]\right] > D[i-1]$
    $decision[i] \leftarrow$ ”$take\ interval$”
$\boldsymbol{else}$
    $decision[i] \leftarrow$ ”$skip\ interval$”
```

# Reconstructing solution from $decision[1..n]$

```
$result \leftarrow \emptyset$
$ind \leftarrow n$
$\boldsymbol{while}(ind \neq 0)$
    $\boldsymbol{if}\ decision[ind] = \text{“}take\ interval\text{”}$
        $result.insert(ind)$
        $ind \leftarrow before[ind]$
    $\boldsymbol{else}$
        $ind \leftarrow ind - 1$
$\boldsymbol{return}\ result$
```

# Recall, Fractional Knapsack

| | | | | | |
|---|---|---|---|---|---|
| Items: | |  | |  | |
| Weight: | 40 kg | 25 kg | 10 kg | 35 kg | 5 kg |
| Total Value: | $120 | $125 | $200 | $70 | $1000 |
| Value per unit weight: | $3 | $5 | $10 | $2 | $200 |

"knapsack"
Capacity: 100 kg

Solution:

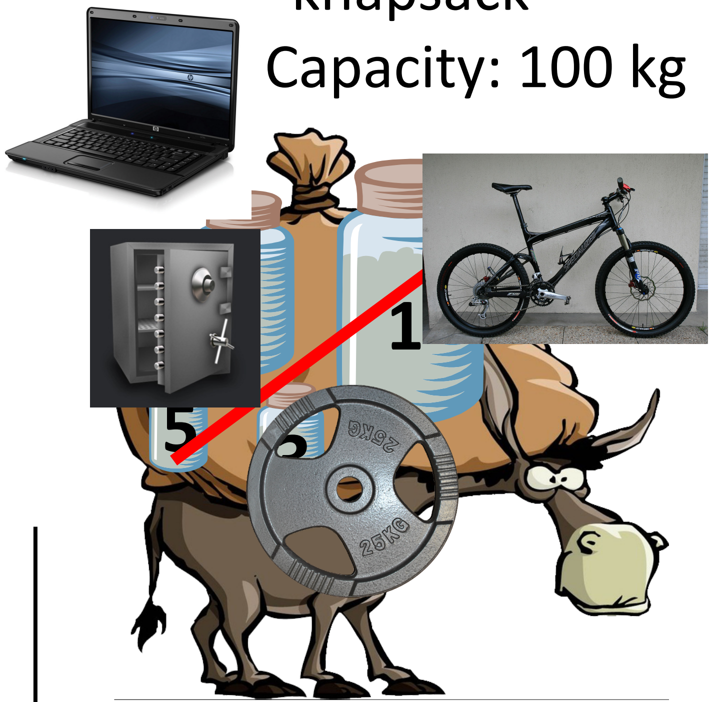
- 5 kg of 5
- 10 kg of 3
- 25 kg of 2
- 40 kg of 1
- 20 kg of 4

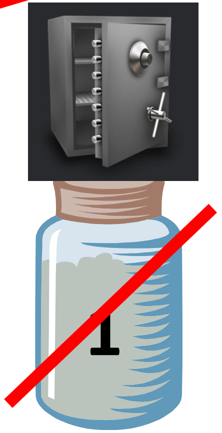
Empty 4 out a bit until it is 20 kg and then put it on donkey

# Integral (0-1)

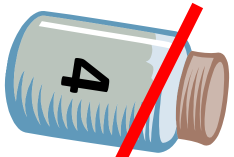
## ~~Recall, Fractional Knapsack~~

“knapsack”
Capacity: 100 kg



| | | | | | |
|---|---|---|---|---|---|
| Items: |  1 | 2 | 3 | 4 | 5 |
| Weight: | 40 kg | 25 kg | 10 kg | 35 kg | 5 kg |
| Total Value: | $120 | $125 | $200 | $70 | $1000 |
| Value per unit weight: | $3 | $5 | $10 | $2 | $200 |

~~Solution:~~
- ~~5 kg of 5~~
- ~~10 kg of 3~~
- ~~25 kg of 2~~
- ~~40 kg of 1~~
- ~~20 kg of 4~~



~~Empty 4 out a bit until it is 20 kg and then put it on donkey~~

Solution:
1, 2, 3, 5

# Fractional Knapsack formally

**Input**: $n$ items described by two parameters given as arrays of numbers:

$W[1..n]$ – where $W[i]$ is the weight of the $i$th item

$V[1..n]$ - where $V[i]$ is the total value of the $i$th item

$C$ – capacity of the knapsack

**Output**: $X[1..n]$ – array of fractions such that

$X[i] \in [0, 1]$

$\sum_{i=1}^{n} X[i] \cdot W[i] \leq C$

$\sum_{i=1}^{n} X[i] \cdot V[i]$ is as large as possible

# Integral (0-1)
# ~~Fractional~~ Knapsack formally

**Input**: $n$ items described by two parameters given as arrays of numbers:

$W[1..n]$ – where $W[i]$ is the weight of the $i$th item

$V[1..n]$ - where $V[i]$ is the total value of the $i$th item

$C$ – capacity of the knapsack

**Output**: $X[1..n]$ – array of fractions such that

~~$X[i] \in [0,1]$~~ $X[i] \in \{0,1\}$

$\sum_{i=1}^{n} X[i] \cdot W[i] \leq C$

$\sum_{i=1}^{n} X[i] \cdot V[i]$ is as large as possible

There are no known greedy algorithms for this problem

Greedy algorithms are also unlikely to work for it

In fact, there are no known truly efficient algorithms for this problem (polynomial runtime in the input length) and they are unlikely to exist

We can solve this problem in time $O(n \cdot C)$ with dynamic programming

This is called **pseudo-polynomial time** (more on this later)

# Sub-problems for pseudo-polynomial DP for Knapsack

- We can reduce the problem size by reducing capacity $C$
- We can reduce the problem size by restricting to the first $i$ inputs $W[1..i], V[1..i]$
- Turns out we need both

- Assume $W[i], V[i], C \in \mathbb{N}$ for simplicity

# Optimal substructure property

Consider items $W[1..i], V[1..i]$ and capacity $c$

Optimal solution $OPT$ for this instance either:

- Doesn’t include item $i$: it must be an optimal solution to

$$W[1..i-1], V[1..i-1] \text{ and capacity } c$$

- Includes item $i$: then it collects value $\underline{v[i]}$ and must contain an optimal solution to $W[1..i-1], V[1..i-1]$ and capacity $c - w[i]$ (only possible if $w[i] \leq c$)

**Proof**: “cut and paste” argument

# Sub-problems for pseudo-polynomial DP for Knapsack

Semantic array:

$D[i, c] =$ maximum value possible using only the first $i$ items and not exceeding the capacity bound $c$

Value of the solution to the whole problem $D[n, C]$

Computational array:

$$D[i, c] = \max\Big(D[i-1, c], \mathbb{I}(w[i] \leq c)\big(D\big[i-1, c-w[i]\big] + v[i]\big)\Big)$$

$D[0, c] = D[i, 0] = 0$ for $i \in \{0, 1, \ldots, n\}, c \in \{0, 1, \ldots, C\}$

# Computing optimal **value**

```
$DPKnapsack(W[1..n], V[1..n], C)$
    instantiate array $D[0..n, 0..C]$
    $\boldsymbol{for}\ i = 0\ \boldsymbol{to}\ n\quad D[i,0] \leftarrow 0$
    $\boldsymbol{for}\ c = 0\ \boldsymbol{to}\ C\quad D[0,c] \leftarrow 0$
    $\boldsymbol{for}\ i = 1\ \boldsymbol{to}\ n$
        $\boldsymbol{for}\ c = 1\ \boldsymbol{to}\ C$
            $D[i,c] \leftarrow D[i-1,c]$
            $\boldsymbol{if}\ w[i] \le c$
                $D[i,c] \leftarrow \max\big(D[i,c], D\big[i-1, c-w[i]\big] + v[i]\big)$
    $\boldsymbol{return}\ D[n,C]$
```

Exercise: modify this code to compute an actual optimal set of items!

$O(C)$

$O(n)$

Overall running time is $O(nC)$

# Why is running time $O(nC)$ not polynomial?

Observe that to write down value $C$ as part of the input we only need $\log C$ bits

Therefore, input length is expressed in terms of $n$ and $C$

Polynomial running time refers to polynomial in the input length

Therefore, polynomial running time for Knapsack would be expressed as a polynomial in terms of $n$ and $\log C$

Our running time is $n \cdot 2^{bitlength(C)}$, which is exponential in input length

Running times which are polynomial in numerical values are called pseudo-polynomial

## Example

Suppose that $n = 10000, W[1..10000], V[1..10000]$ and

$$C = 999999999999$$

Observe that this value of $C$ requires only 12 digits to specify. So it is a very short input, although it stands for a very large number

Our algorithm would run in time proportional to

$$nC = 9999999999990000$$

That’s a long time to wait!

# You should now be able to…

- Explain the dynamic programming paradigm

- State the difference between a semantic array and a computational array

- Understand drawbacks and benefits of memoization approach vs iterative approach to dynamic programming

- Understand drawbacks and benefits of greedy algorithms as compared with dynamic programming algorithms

# Review Questions

- Define optimal substructure property
- Define overlapping subproblems property
- Define semantic and syntactic arrays for Longest Increasing Subsequence, Weighted Interval Scheduling, and Knapsack
- Write down pseudocode for each of the above problems (computing value of the solution as well as the solution itself). Analyze runtime and correctness