# CS 310

*"To iterate is human, to recurse divine."*

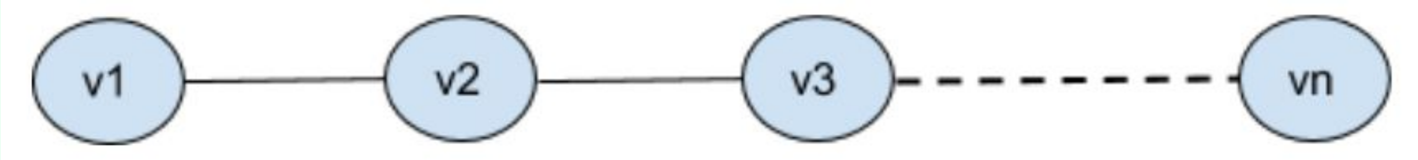
*- L. Peter Deutsch*

## Dynamic programming

- There is an **ordering** on the subproblems, and
- A recurrence relation that shows how to solve a subproblem given the answers to "smaller" subproblems that appear earlier in the ordering.
- In dynamic programming the **DAG** is implicit. Its nodes are the subproblems we define, and its edges are the dependencies between the subproblems.

# An example

Undirected graph G(V, E) where the vertices are connected in a **chain** as shown below.




The vertices represent chemicals and the edges between them represent interactions between pair of chemicals. **Each of the chemicals have a price ($price_i$) in rupees.** You have to pack a subset of chemicals in **one box** such that the **total price is maximized**. Chemicals that interact with each other cannot be placed together in the box.

**You are given a chain of five chemicals in the following order:**

**c1, c2, c3, c4, c5. Their prices are**

**1, 8, 2, 1, 7 respectively.**

Basis: $f(0)=0$, $f(1)=price_1$

Recurrence: $f(n) = \max(\ price_n+f(n-2),\ f(n-1)\ )$

| Chemicals | c1 | c2 | c3 | c4 | c5 |
|---|---|---|---|---|---|
| Price | 1 | 8 | 2 | 1 | 7 |

$f(0) = 0$, $f(1) = 1$

$f(2) = \max(\ 8+f(0)\ ,\ \ f(1)\ ) = \max(\ 8+0,\ 1) = 8$

$f(3) =$

$f(4) =$

$f(5) =$

Basis: $\texttt{f(0)=0, f(1)=price}_1$

Recurrence: $\texttt{f(n) = max( price}_n\texttt{+f(n-2), f(n-1) )}$

| Chemicals | c1 | c2 | c3 | c4 | c5 |
|---|---|---|---|---|---|
| Price | 1 | 8 | 2 | 1 | 7 |

$f(0) = 0, f(1) = 1$

$f(2) = \max(8+f(0), f(1)) = \max(8+0, 1) = 8$

$f(3) = \max(2+f(1), f(2)) = \max(2+1, 8) = 8$

$f(4) = \max(1+f(2), f(3)) = \max(1+8, 8) = 9$

$f(5) = \max(7+f(3), f(4)) = \max(7+8, 9) = 15$

Basis: `f(0)=0, f(1)=price`$_1$

Recurrence: `f(n) = max( price`$_n$`+f(n-2), f(n-1) )`

| Chemicals | c1 | c2 | c3 | c4 | c5 |
|---|---|---|---|---|---|
| Price | 1 | 8 | 2 | 1 | 7 |

f(0) = 0, f(1) = 1

f(2) = max ( **8**+**f(0)** , f(1) ) = max ( 8+0, 1) = 8

f(3) = max ( 2+f(1) , **f(2)** ) = max ( 2+1, 8) = 8

f(4) = max ( 1+f(2) , f(3) ) = max ( 1+8, 8) = 9

f(5) = max ( **7**+**f(3)** , f(4) ) = max ( 7+8, 9) = 15

# Memoized version - chemical chain

```
/* initialization */
for j= 1 to n
   Memo[j] = -1
Memo[0] = 1
Memo[1] = price[1]

chain(n) {
 if (Memo[n] < 0)
   Memo[n]= max(price[n]+chain(n-2), chain(n-1));
   return Memo[n];
}
```

# How do we find the set of chemicals?

Basis: $\mathtt{f(0)=0,\ f(1)=price_1}$

Recurrence: $\mathtt{f(n) = max(\ price_n+f(n-2),\ f(n-1)\ )}$

# How do we find the set of chemicals?

Basis: `f(0)=0, f(1)=price`$_1$

Recurrence: `f(n) = max( price`$_n$`+f(n-2), f(n-1) )`

---

```
FindSolution(n) {
  if (n<=0) return null
  if(price[n]+Memo[n-2] > Memo[n-1]) {
     /* Store n in a set*/
     FindSolution(n-2);
  }
  else
     FindSolution(n-1);
}
```

**Complexity of FindSolution()?**

# How do we find the set of chemicals?

Basis: $\mathtt{f(0)=0,\ f(1)=price_1}$

Recurrence: $\mathtt{f(n) = max(\ price_n+f(n-2),\ f(n-1)\ )}$

---

```
FindSolution(n) {
   if (n<=0) return null
   if(price[n]+Memo[n-2] > Memo[n-1]) {
      /* Store n in a set*/
      FindSolution(n-2);
   }
   else
      FindSolution(n-1);
}
```

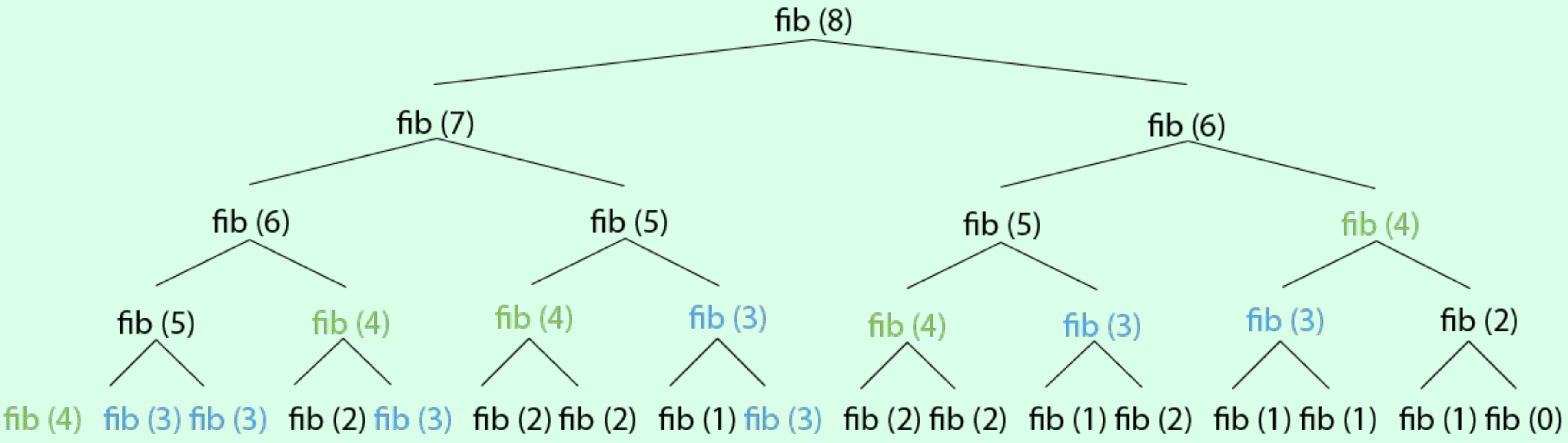
**Complexity of FindSolution(): O(n)**

# Dynamic Programming

Optimization problems must have the following two key ingredients in order for dynamic programming to apply.

- optimal substructure
  - a problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems.
- overlapping subproblems
  - **When a recursive algorithm revisits the same problem repeatedly**, we say that the optimization problem has overlapping subproblems.
    - *Typically, the total number of distinct subproblems is a polynomial in the input size.*

# Example of overlapping subproblems



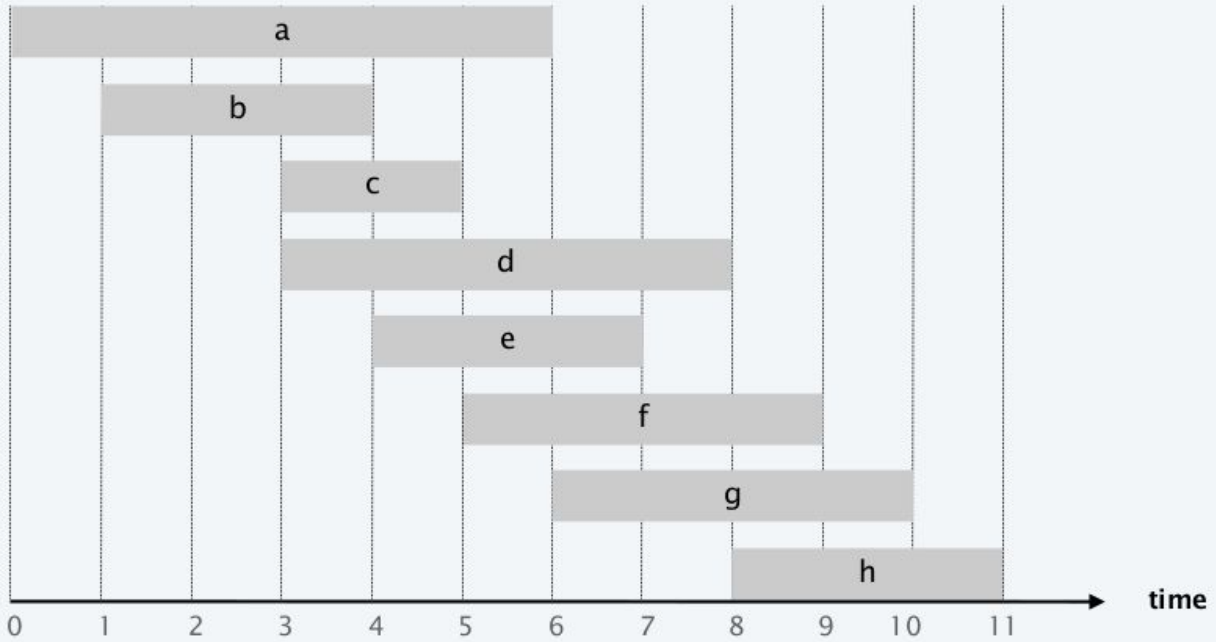
# Three Steps to Dynamic Programming

1. Formulate the answer as a recurrence relation or recursive algorithm.
2. Space of subproblems must be "small", typically bounded by a polynomial (i.e., show that the **number of different arguments** to your recursive function isn't large, so that we can benefit from storing the results).
3. **Specify an order** of evaluation for the recurrence so you always have what you need.

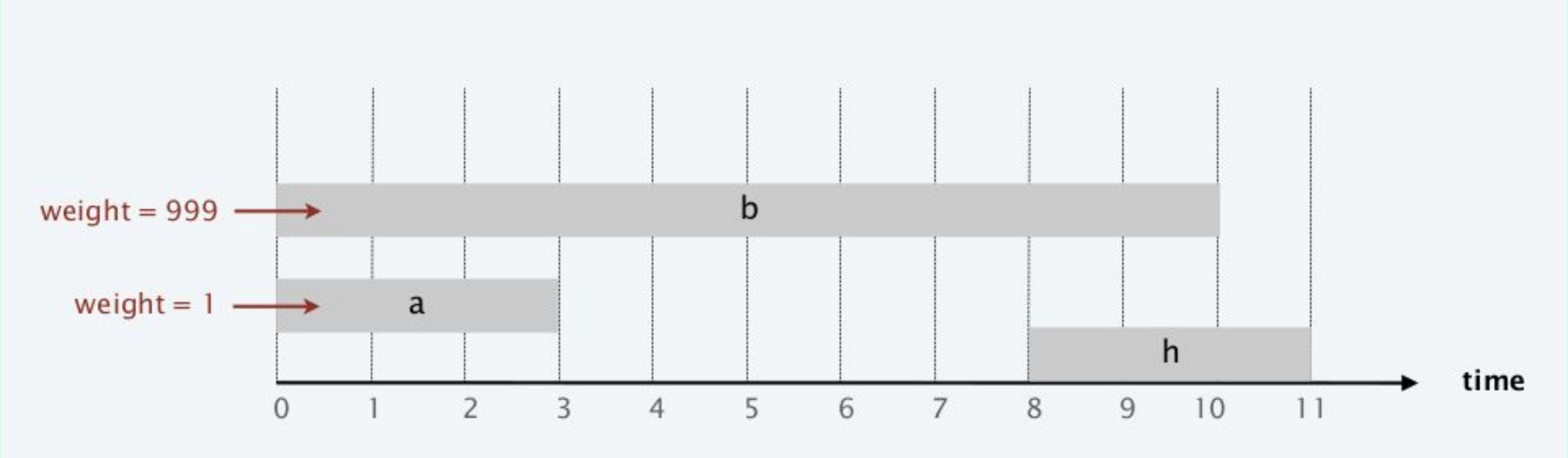
# Weighted interval scheduling

## Weighted interval scheduling problem.

- Job $j$ starts at $s_j$, finishes at $f_j$, and has weight or value $v_j$.
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.



# Earliest Finish Time First

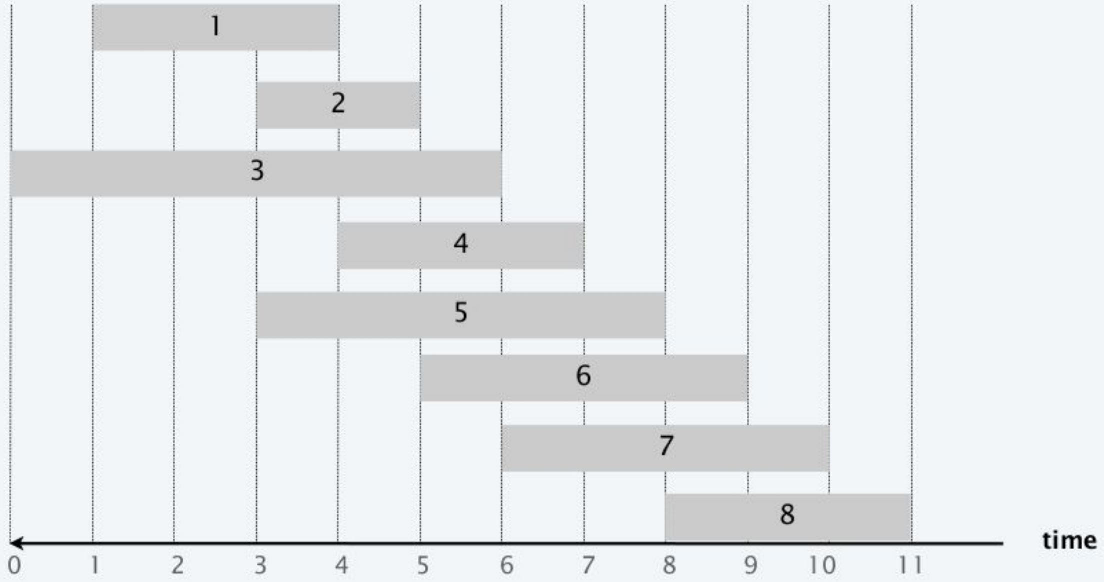


# Weighted interval scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Def. $p(j)$ = largest index $i < j$ such that job $i$ is compatible with $j$.

Ex. $p(8) = \square$, $p(7) = \square$, $p(2) = \square$.

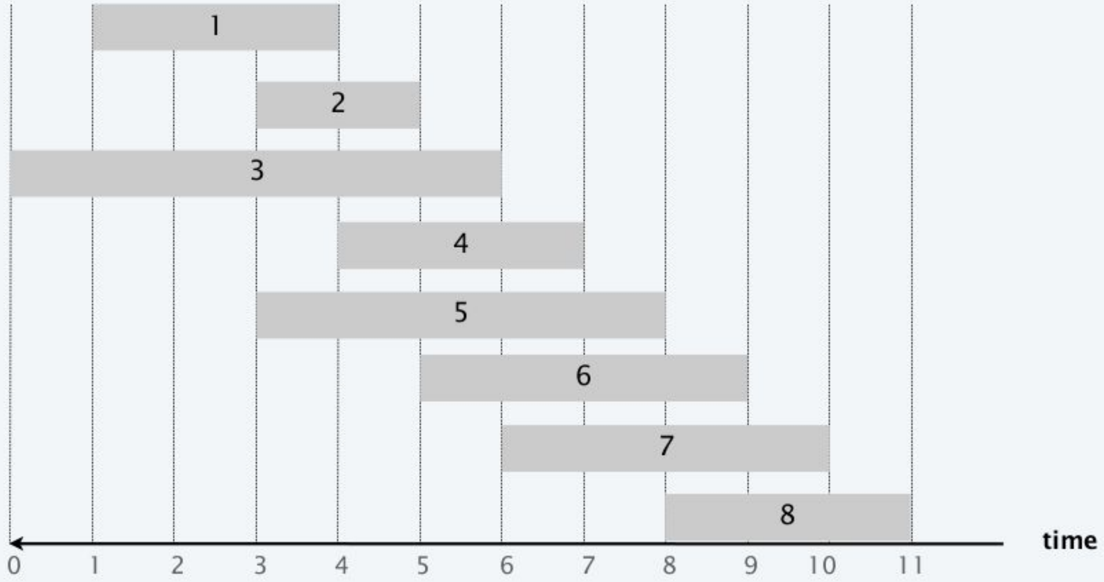


# Weighted interval scheduling

Notation. Label jobs by finishing time: $f_1 \le f_2 \le \ldots \le f_n$.

Def. $p(j)$ = largest index $i < j$ such that job $i$ is compatible with $j$.

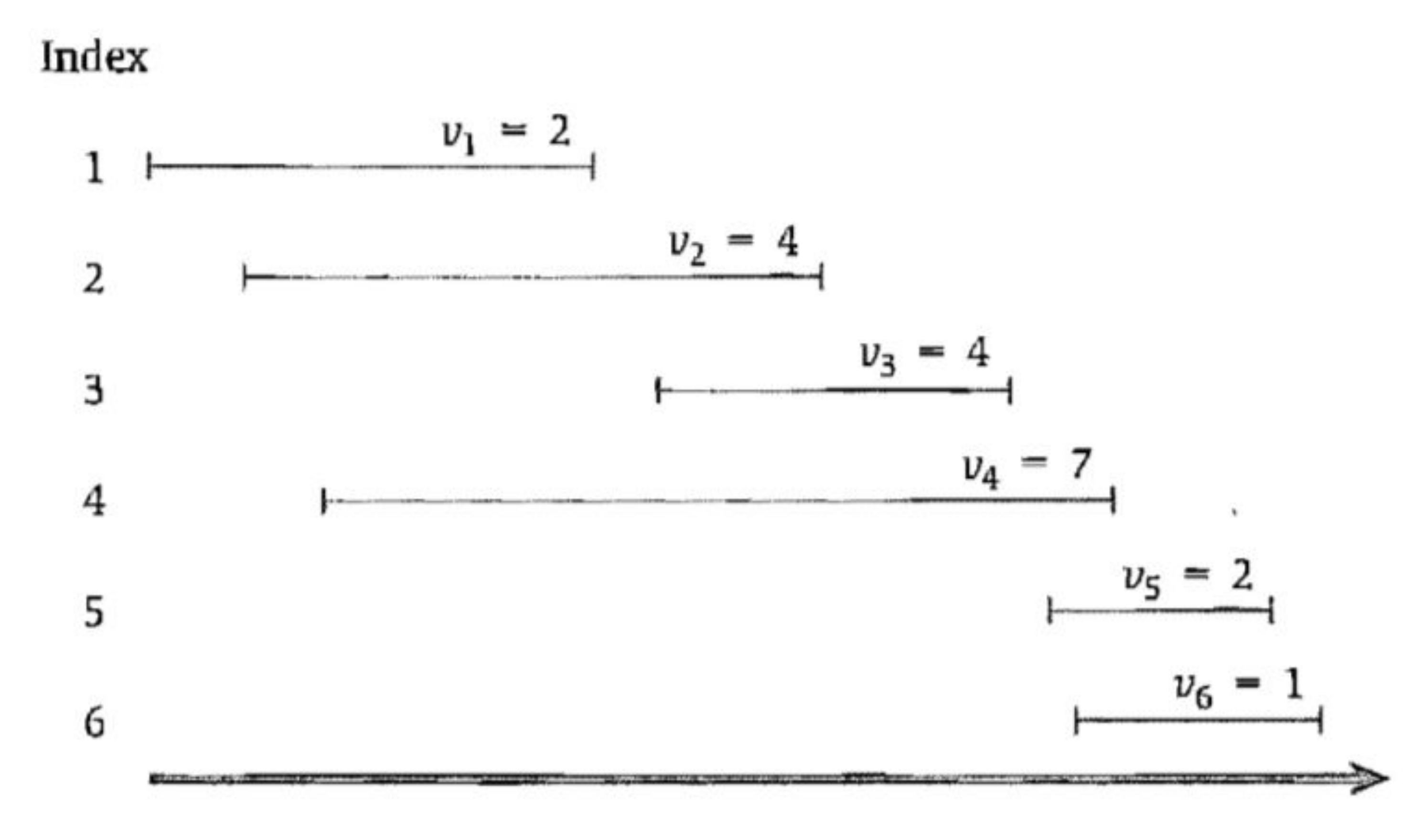
Ex. $p(8) = 5, p(7) = 3, p(2) = 0.$



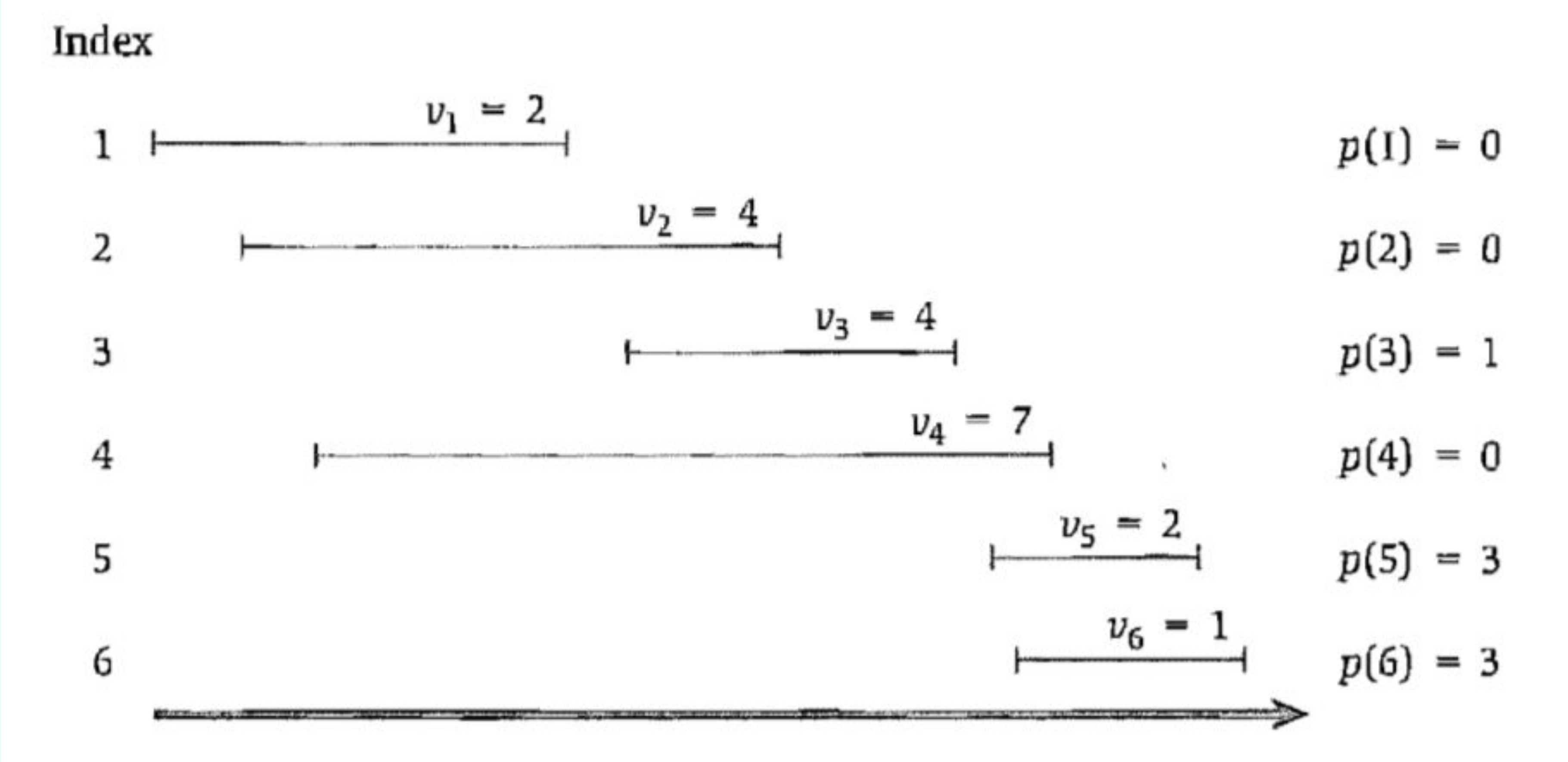
# Weighted Interval Scheduling

**Indices are names of intervals.**

**And $v_i$ are weights of intervals.**



# Weighted Interval Scheduling

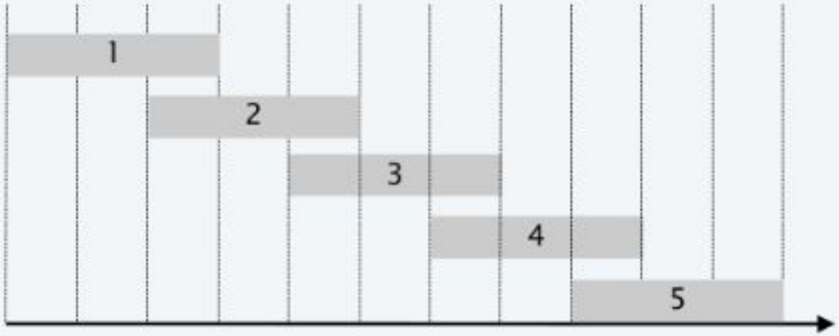


# Weighted Interval Scheduling

$$OPT(j) = \begin{cases} 0 & \text{if } \ j = 0 \\ \max\left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

```
Input:  n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n].
Compute p[1], p[2], …, p[n].

Compute-Opt(j)
if j = 0
   return 0.
else
   return max(v[j] + Compute-Opt(p[j]), Compute-Opt(j-1)).
```

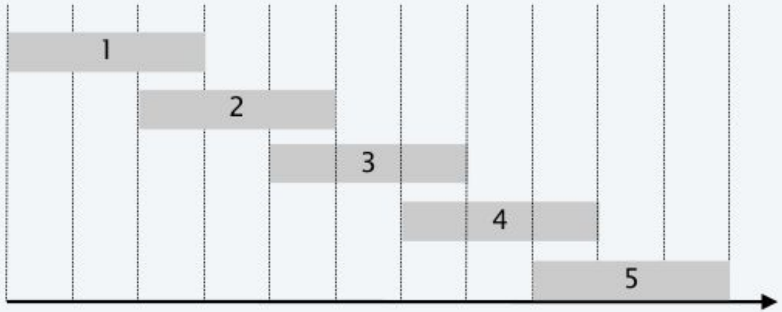



p(1) = 0, p(j) = j-2

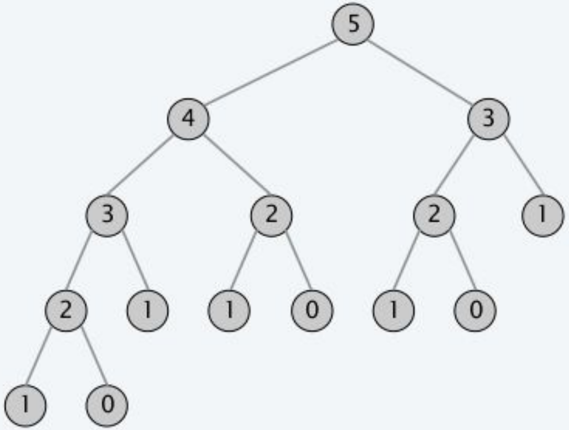
# Weighted interval scheduling: brute force

Observation. Recursive algorithm fails spectacularly because of redundant subproblems $\Rightarrow$ exponential algorithms.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.




p(1) = 0, p(j) = j–2




recursion tree

# Weighted interval scheduling: memoization

Memoization. Cache results of each subproblem; lookup as needed.

```
Input:  n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n].
Compute p[1], p[2], …, p[n].

for j = 1 to  n
   M[j] ← empty.
M[0] ← 0.

M-Compute-Opt(j)
if M[j] is empty
   M[j] ← max(v[j] + M-Compute-Opt(p[j]), M-Compute-Opt(j – 1)).
return M[j].
```

# Running time?

```
Input:  n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n].
Compute p[1], p[2], …, p[n].

for j = 1 to  n
   M[j] ← empty.
M[0] ← 0.

M-Compute-Opt(j)
_______________
if M[j] is empty
   M[j] ← max(v[j] + M-Compute-Opt(p[j]), M-Compute-Opt(j – 1)).
return M[j].
```

# Running time?

```
Input:  n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n].
Compute p[1], p[2], …, p[n].

for j = 1 to  n
   M[j] ← empty.
M[0] ← 0.

M-Compute-Opt(j)
________________
if M[j] is empty
   M[j] ← max(v[j] + M-Compute-Opt(p[j]), M-Compute-Opt(j - 1)).
return M[j].
```

For sorting: O(n log(n))

For M-Compute-Opt(n): O(n)

# Weighted interval: Unwind recursion

BOTTOM-UP $(n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n)$

Sort jobs by finish time so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p(1), p(2), \ldots, p(n)$.

$M[0] \leftarrow 0$.

FOR j = 1 TO $n$

$\quad M[j] \leftarrow \max \{ v_j + M[p(j)], M[j-1] \}$.

Sorting: O(n log(n))
For-loop: O(n)

# Weighted interval scheduling: finding a solution

Q. DP algorithm computes optimal value. How to find solution itself?

A. Make a second pass.

```
Find-Solution(j)
if j = 0
   return ∅.
else if (v[j] + M[p[j]] > M[j-1])
   return {j } ∪ Find-Solution(p[j]).
else
   return Find-Solution(j-1).
```

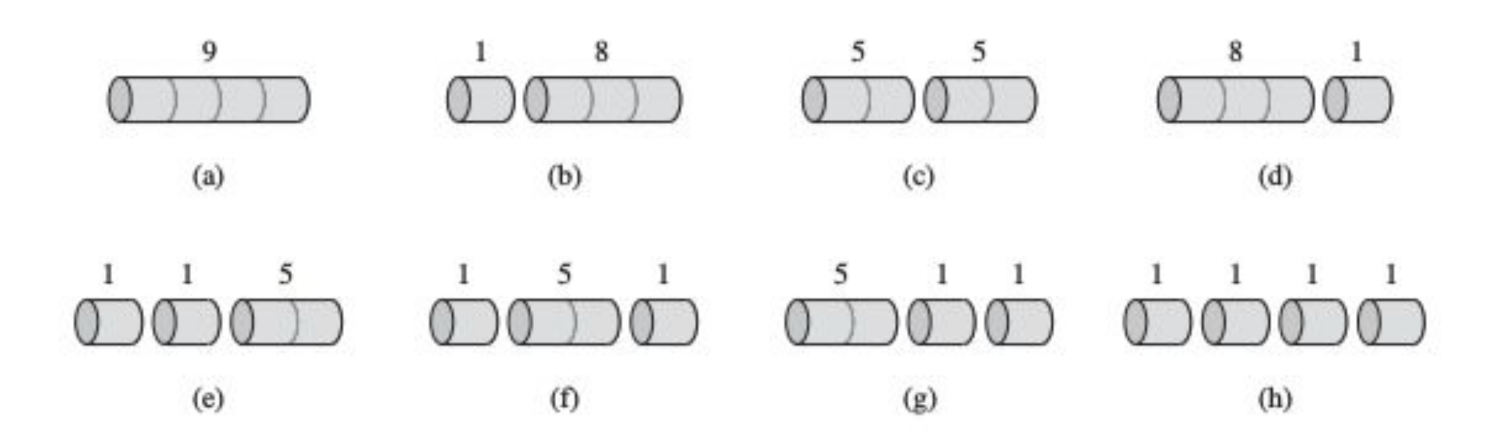
# Rod Cutting Problem

Given a rod of length n inches and a table of prices $p_i$ for $i = 1, 2, \ldots\ n$, determine the maximum revenue $r_n$ obtainable by cutting up the rod and selling the pieces.

| length $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| price $p_i$ | 1 | 5 | 8 | 9 | 10 | 17 | 17 | 20 | 24 | 30 |

# Rod Cutting Problem

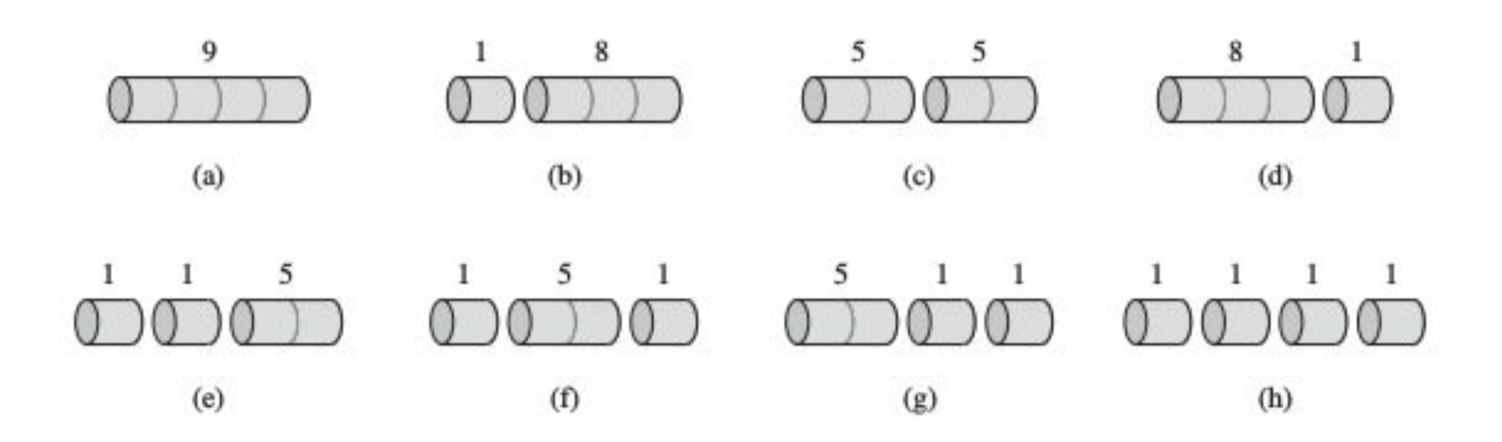
Given a rod of length n inches and a table of prices $p_i$ for $i = 1, 2, \ldots\ n$, determine the maximum revenue $r_n$ obtainable by cutting up the rod and selling the pieces.




**Brute Force:** How many different ways to cut the rod?

# Rod Cutting Problem

The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece.



(a) (b) (c) (d)
(e) (f) (g) (h)

**Brute Force:** $2^{n-1}$ different ways to cut the rod.